Optimization of Cloud Storage Costs Through Efficient Data Placement

Enrico Giorio

Thesis to obtain the Master of Science Degree in

Engenharia Informática e de Computadores

Supervisor: Prof. Paolo Romano

Examination Committee

Chairperson: Prof. José Alberto Rodrigues Pereira Sardinha
Supervisor: Prof. Paolo Romano
Member of the Committee: Prof. João Coelho Garcia

November 2022

Acknowledgments

This thesis is dedicated to my brothers Pietro and Manuele, which I love from the bottom of my heart. Even though we couldn't be as close as I wished in the last years, I have always loved you and supported you from afar. I hope I have been a good role model for you and I wish I could keep being an inspiration for your future. Be yourselves and believe in your abilities, you will get further than you can imagine. I love you.

I must thank my parents for being supportive and loving me throughout my whole journey, in good and bad times. You are by far the best role models one could have, and I wouldn't have been able to achieve any of this without you. You are my lighthouse and the people I love the most.

A special thanks goes to Rea, my girlfriend, who has been with me for most of my journey and has loved me and supported me from day one. You turned this experience into an adventure, the most beautiful of my life.

This thesis is also dedicated to my grandpa Dionisio, which is who I would have made proud and happy the most. Wherever you are now, I love you and I feel your wholehearted support. You, Alda, Natalina and Ernestino are part of the reason why I am who I am today, and your love will always remain in my heart for the rest of my life. Somehow, you managed to keep supporting me throughout the whole journey, each of you in a different way, despite all you have been going through. I admire you and love you.

I would also like to acknowledge my dissertation supervisors Prof. Paolo Romano and Prof. Gianpaolo Cugola for their insights, expertise and sharing of knowledge that made this thesis possible.

This thesis is the product of three years of exploration, commitment, sacrifice, experiences and hundreds of other things that I would never be able to put into words.

After a difficult first year at Politecnico di Milano, undeniably impacted by the pandemic, I made what turned out to be one of my best life choices by embracing the challenge of a double degree abroad. I landed in Lisbon without knowing anything or anyone and I met incredible friends right away. Some stayed, some left, but the memories will be forever etched in my mind.

This eye opening experience showed me different takes on life and work. I explored many cultures and became interested in them. I experienced new ways of learning and how to enjoy doing so.

My master thesis is not just the final artifact of three University degrees. Personally, it signifies something deeper. It represents myself becoming a grown man, with a new life ahead, who is going to make treasure of everything I collected during these years, six in fact, and bring them with me in the adult life that I will undertake.

Finally, I would like to thank all my other relatives that have always been admiring me, helping me and supporting me. This thesis was also possible because of you.

Abstract

Cloud databases are nowadays available in various deployment models. One approach stands out regarding cost efficiency: Infrastructure as a Service. Every major cloud provider offers various types of infrastructure on which customers can deploy their database software solution of choice, with different types of nodes being cost optimal for specific types of workloads. Nowadays, cloud architects tend to deploy infrastructures made of arbitrary numbers of nodes of the same type. When the data to store features diverse usage patterns, there is no clear advantage in adopting such strategy, other than simplicity of deployment. On the other hand, deploying a heterogeneous cluster made of different node types is a non-trivial task, and certainly a complex one. So complex, in fact, that it is hard, if not impossible, for human operators to consistently find infrastructure configurations that optimally minimize the costs. This thesis is framed in a larger project: PlutusDB, an autonomic system which aims at addressing the issue of using cost-wise suboptimal homogeneous IaaS clusters. PlutusDB instantiates potentially multiple database instances, each of them associated with an independent IaaS cluster. The system accurately sizes each sub-cluster to maximize resource usage and therefore minimize the overall cost. PlutusDB is also the first system that approaches data placement with cost optimization in mind, transparently analysing the data items as a whole and autonomously deciding their optimal placement. This dissertation, after providing a complete and detailed view of PlutusDB, focuses on the design and implementation of its core component: the Optimizer.

Keywords

NoSQL, IaaS, Autonomic Systems, Data Placement, Data Migration

Contents

List of Figures

List of Tables

Acronyms

AM	Analytical Modelling
CRM	Customer Relationship Management
DBaaS	Database as a Service
DSP	Data Stream Processing
EBS	Enterprise Block Storage
HDD	Hard Disk Drive
IaaS	Infrastructure as a Service
ILP	Integer Linear Programming
IOPS	I/O Per Second
JSON	JavaScript Object Notation
LP	Linear Programming
MILP	Mixed Integer Linear Programming
MIQP	Mixed Integer Quadratic Programming
MIQCP	Mixed Integer Quadratically Constrained Programming
ML	Machine Learning
OS	Operating System
PaaS	Platform as a Service
QP	Quadratic Programming
ROWA	Reat One Write All
RF	Replication Factor
RRU	Read Request Unit
SaaS	Software as a Service
SDN	Software Defined Networking

SLA	Service Level Agreement
SSD	Solid State Drive
TAS	Transactional Auto Scaler
VM	Virtual Machine
WRU	Write Request Unit

1

Introduction

Contents

The advent of cloud-based database services opened new possibilities for small businesses and enterprises. As of today, multiple cloud database technologies and paradigms are available, each of them being optimized for specific types of workloads. The more popular deployment models for cloud databases when cost efficiency is the priority is undoubtedly Infrastructure as a Service (IaaS).

IaaS is a deployment model that allows customers to independently run their own database service of choice on a rented infrastructure. All major cloud providers offer a wide variety of types of Virtual Machines (VMs) to choose from, which customers accurately select based on the properties of the applications they intend to run. Most of the times, the infrastructure for running a database system is composed by nodes of the same type, to facilitate the process of choosing and maintaining it.

The available types of nodes are usually either general purpose or specialised for certain types of workloads. When the data to store in an IaaS database system is known to uniformly produce a specific workload, choosing which node type to deploy in multiple instances is not a hard task. On the other hand, if the application does not generate uniform and comparable workload characteristics across the data set, choosing the same node type and instantiating it multiple times might not be the most cost effective choice. For instance, a throughput optimized instance might end up storing items that are infrequently accessed, and therefore being not cost efficient.

Applications nowadays need to store diverse types of data, each of them with different access patterns. For this reason, an ideal database system that is extremely cost efficient would be deployed on an heterogeneous cluster of machines, and selectively store data items in the nodes which cost characteristics are best suited for their access pattern. Such a system must tackle several key topics in order to be functional, such as how to perform decisions on the placement of the items, how to effectively track it at runtime, whether to relocate items whenever the access pattern changes and how to do it, and many more. Furthermore, when adopting IaaS clusters, the so-called “auto-scaling” feature (that automatically resizes the pool of nodes to accommodate changing workloads) is often an extra cost. To incur in lower expenses, a database manager could decide to implement its own IaaS scaling mechanism, by addressing some additional issues, such as, when adding computational nodes, which data transfers are necessary, which consistency guarantees should be in place, how the data should be transferred?

Significant amounts of research has been done on data placement, migration techniques and scaling approaches, although very few projects address a cost-oriented approach, often aiming at developing performance-oriented algorithms and systems.

In this dissertation, we present PlutusDB: an autonomic system that, from a client perspective, acts as any other commercial cloud database. Its architecture, however, includes (potentially) multiple IaaS clusters, each of them with an independent database instances. The system aims at exploiting the advantages of using diversely specialized cloud infrastructures by placing data items in the optimal nodes, based on their access frequency, therefore minimizing the overall costs. PlutusDB does so by solving a Integer Linear Optimization problem to achieve an ideal placement that minimizes the operational costs. It then performs an autonomous decision on whether or not to migrate data items and eventually transfers them without interrupting the service.

1.1 Objectives

This dissertation is developed around the following objectives:

1. Designing the architecture of PlutusDB, which includes the definition of:
   - Tasks and responsibilities of each internal component and their interaction.
   - Execution flows
2. Providing implementation proposals for the main operational functions, such as a proprietary procedure for the data transfer
3. Implementing the Optimizer, central and main building block for PlutusDB, as a component that could also be used as a standalone tool.
4. Evaluating the Optimizer through the exploration of several scenarios, with the aim of finding data sets that are cost-wise better suited to be stored in hybrid clusters. That is, clusters composed of instances of multiple node types.
5. Analyzing performance, usability and limitations of the Optimizer as a standalone tool.

1.2 Organization of the Document

This thesis is organized as follows: after the introduction in Chapter 1, a study of related work, recent publications, and state-of-the-art research is presented in Chapter 2. The architectural view of the proposed solution, PlutusDB, is presented in Chapter 3, where the vision of each the components of the system is described in details. Chapter 4 contains the proposal of a concrete implementation for the principal component of PlutusDB (the Optimizer). Subsequently, in Chapter 5, we perform a validation and testing of the proposed implementation, providing real data that demonstrate the effectiveness of such component. Finally, conclusions regarding the effectiveness of the Optimizer and, more in general, about PlutusDB's idea, will be drawn in Chapter 6.

2

Related Work

Contents

This chapter covers all the literature studied and analyzed as a foundation for PlutusDB. After describing existing cloud delivery models and their characteristics, we focus on data stores and provide an overview of the existing paradigms, consistency models, and strategies and algorithms to achieve them. We move on with the introduction of the concepts of stateful and stateless applications, briefly explaining what they are and focusing on addressing the scalability issues that must be considered for each paradigm. Data migration and data placement are analyzed in depth to prepare the reader for the thesis' contribution, which will use many concepts explored in this chapter.

2.1 Cloud Services

Cloud computing [1] features various delivery models targeting different kinds of users. The common denominator of all models is offering a paid service that allows users to delegate managing and provisioning physical machines, updating systems and drivers, dealing with hardware security and other tasks to the vendor, and focusing on productivity and business-specific tasks. Some of the benefits that all types of cloud services have in common are also related to the following:

- On-demand resources: resource allocation is quickly and transparently obtained by the customer upon need. For instance, when workload spikes occur, customers need not have to allocate resources explicitly. The operation is performed automatically by the cloud provider instead.
- Costs advantages: investing in on-premises infrastructure comes with the risks of oversizing (wasting resources when the workload is lower than average) and under-sizing (not having enough resources during workload peaks). Most cloud providers employ a "pay only for what you use" approach. Dealing with local physical infrastructure is risky from a security standpoint and can become cumbersome if the application workload increases.
- Availability: on-premises infrastructures require constant monitoring, and it can be costly to maintain backup copies that can substitute the running instances whenever failures occur. Cloud-based services are virtually "always running", and failures are handled in a short time without any additional cost to be sustained by the customer.

Although there are various cloud service types, they can all be grouped into three main categories, described next.

2.1.1 Infrastructure as a Service

IaaS is an essential service model: vendors provide the infrastructure (either physical or virtual) with little to no installations and configurations required [2]. The cost model is typically pay-per-use, allowing users to pay for the exact resources used. The most popular billing types are:

- Pay-as-you-go (per hours or minutes of usage)
- On-demand (Hourly, Daily, Monthly, or even Yearly billing, only when explicitly requested)
- Subscription-based (Time defined contracts, usually long-lasting)
- Spot market (idle machines are made available by the cloud provider. The availability is not always guaranteed, and prices fluctuate, depending on offer and demand).

The offered instances are usually VM running an arbitrary Operating System (OS) of choice, which make IaaS the lowest level, highest customizable, and most cost-efficient cloud service. The target clients are, generally, companies with internal developing teams or IT companies. Examples of IaaS resources are Computing instances (VMs), Storage Instances (volumes), and Software Defined Networking (SDN). IaaS commercial offerings are provided by Amazon Web Services [3], Google Cloud Platform [4], Microsoft Azure [5], Oracle [6], IBM [7] and many more.

2.1.2 Platform as a Service

Platform as a Service (PaaS) is a cloud service that offers whole platforms to deploy applications. Essentially, a standard IaaS solution is being offered, although with preconfigured OS, middleware, and runtime environments. It relieves developers from cumbersome configuration tasks that precede application deployment [8, 9].

2.1.3 Software as a Service

Software as a Service (SaaS) is a solution mainly used by businesses that intend to maintain always up-to-date software without incurring heavy human resources costs. SaaS delivers whole applications running on the cloud. The main advantage that SaaS application suites provide is little to no client-side performance requirements: most of the computation is performed at the cloud provider's computing instances [10, 11]. The areas covered by SaaS are multiple, such as Customer Relationship Management (CRM), email systems, project management tools, document editing tools, and many more. Examples of widely used SaaS software are: SAP Customer Experience and CRM solutions [1], Microsoft Dynamics 365 [2] and Salesforce [3].

2.1.4 Database as a Service

Database as a Service (DBaaS) is a cloud service to create, use, and manage cloud-based databases, allowing users to operate fully on the cloud without having to manage their expensive and often complex

[1] https://www.sap.com/uk/products/crm.html
[2] https://dynamics.microsoft.com/en-us/
[3] https://www.salesforce.com/eu/?ir=1

infrastructure [12].

DBaaS features distinct properties that make it a critical technology that is, in most cases, completely replacing physical databases, in particular:

- Rapid development, creation, and setup compared to traditional on-premises databases which need more attention to preoperative phases.
- Increased versatility: the fact that there is no need to select in advance which hardware to use allows developers to choose from a variety of similar but inherently different database types to make the DB tailored to the business choices.

Existing DBaaS offerings cover different database technologies (see Section 2.2) and most of the times are equipped with additional tools that show the database manager live performance and statistics insights that can help improve business processes.

Most of the DBaaS offerings are billed per access, per transaction or per set of transactions.

2.2 Models for Data Stores

Data stores, whether cloud based or on-premises, are offered by vendors in diverse types. This section glazes over the advantages and disadvantages of the most popular models, focusing on NoSQL databases, which are the main object of this study.

2.2.1 Relational

Relational databases [13] are the legacy database systems, still widely used [14, 15] due to their strong data modelling capabilities.

In relational databases, stored items are collections of values (tuples) which are grouped in tables and are usually identified by a unique key (primary key). Relationships between items can be created through links across tables, called foreign keys. Linked structures are helpful to perform complex queries that require merging tables (joins) or iterating through them. Relational databases typically support transactions and strong consistency models, explained in Section 2.3.2, although some of them provide weaker consistency guarantees to facilitate scalability [16, 17]. Some of the most widely used relational databases [18] are MySQL[4] and PostgreSQL[5].

High availability and non-structured data are only some of the required features that standard relational databases simply do not support. NoSQL databases are a possible alternative.

[4]https://www.mysql.com
[5]https://www.postgresql.org

2.2.2 NoSQL

This type of database is non-relational and tables are not linked between each other. Data items are, instead, stored as JavaScript Object Notation (JSON) documents [13, 18–20]. This paradigm allows to store high volumes of unstructured data without excessive overhead, allowing the database to scale effortlessly due to unlinked data.

NoSQL databases usually relax the consistency guarantees (see Section 2.3.2) to achieve lower access latency and together with the absence of relationships between groups of data allows to efficiently scale horizontally (scale-out) [13, 19, 21, 22] (see Sec. 2.5.1). Another advantage of NoSQL databases is that they need not define a schema for data items, allowing for the storing of unstructured data and therefore faster agile development.

NoSQL databases come in various forms, although the most popular ones that will also be used from chapter 3 onward are Key-Value Stores (KVSs).

NoSQL databases are usually deployed as IaaS, which requires performing decisions on various aspects of the system, such as the selection of nodes, the number of nodes to instantiate to sustain specific workloads, which types of storage (SSD, HDD or other) to use, and others. In order to have a solid base for building the mathematical model proposed in section 4.3.1, a decision was made to follow DataStax's whitepaper for deploying Apache Cassandra [23], which advises a number of best practices for the sizing of the underlying cloud infrastructure.

DataStax[6] is a company that, among other services, offers a custom installation of the Apache Cassandra database, called Datastax Enterprise, which offers clients extra functionalities such as a proprietary driver, support for enterprise businesses, more straightforward startup procedures and more benefits.

DataStax is also providing the community with a whitepaper for the deployment of Apache Cassandra, which includes a number of best practices that are useful for, among other things, narrowing down the choice of the infrastructure. The whitepaper is constantly being updated with changing recommendations, with the most up-to-date version at the time of the publication of this dissertation being: 24. Since constantly updating the model during the development process is not feasible and leads to potential errors, we refer to the whitepaper publicly available at the time of the start of this project [23].

The whitepaper starts by recommending the amount of RAM per node, stating that production environment should be deployed on machines with 16 to 64GB of RAM. Then, the paper states that "Insert-heavy workloads are CPU-bound in Cassandra before becoming memory-bound", as "Cassandra is highly concurrent and uses as many CPU cores as available". Therefore, 8-core CPUs are recommended. Finally, we decided to follow a DataStax training session for the sizing of a Cassandra cluster, which states that the maximum capacity for disk drives should be between 3TB and 5TB per node [25].

[6] https://www.datastax.com

For our research, we decided to choose one 4TB volume per instantiated node.

More recent recommendations are available as of today [24], in which the recommended volume capacity is slightly different and the number of volumes depends on the physical volume type (HDD/SSD).

Although we use Apache Cassandra as a reference, this model could be adopted for deployments of other commercial databases, as well.

2.2.2.A Key-Value stores

Inspired by Amazon's Dynamo [26], these systems store data as Key-Value pairs without any underlying structural schema. In each set of data, keys are unique, and values are typically either JSON objects or fundamental data types such as strings.

While simplicity is attractive for application development, it also comes with significant limitations: querying can only be performed by key since values are "opaque": the database system does not have any knowledge over them so they cannot be leveraged for complex queries.

Key-Value stores usually implement redundancy and caching for frequently used data items and typical use case is storing user session information. Some systems store items in-memory (the most widely used [27] is Redis[7]) or on disk (as it is the case for Amazon DynamoDB [26] and Apache Cassandra [28]).

2.2.2.B Column-oriented

Column NoSQL databases organize data in unlinked tables. Each tuple represents a column, the first item in each column being the unique key of each item. It derives that the first row of a table consists of all the item keys. Data is almost always stored persistently and contiguously by column.

On one hand, Column databases deliver enhanced performance for aggregation functions such as SUM, AVERAGE, etc. [29], since attributes of interest are stored contiguously. On the other hand, if a query needs to access only a single record, a column-store will have to seek several times (to all columns/files of a table) to read just this single record, resulting in significantly poor performance.

This kind of database is particularly suited for data mining and analytics applications, since most of the operations involve aggregation.

Column stores often take inspiration from Google BigTable [30].

Apache Cassandra [28] is a Key-Value store that employs some features of the column storages to enhance performance.

Other NoSQL database types exist, such as Document-oriented (an extension of Key-Value stores wiht the possibility of performing content-based queries), Graph (such as Neo4J[8], where data items are

[7]https://redis.io
[8]https://neo4j.com

linked), Time-Series (to store data in time ordered streams), Ledger (that store logs of events associated to data items), Object-Oriented, and others [19].

2.3 Consistency models in distributed cloud storages

Consistency guarantees describe a database system's expected behaviour when simultaneous operations are performed. Relational databases that support transactional semantics and strong isolation levels, usually incur in non-negligible costs in terms of performance and/or availability. Conversely, NoSQL databases adopt weaker consistency levels, such as eventual consistency, which enable the achievement of high performance and scalability - albeit at the cost of ease of programming. Following, an overview of the most commonly implemented consistency models and isolation levels, with a focus on guarantees supported by non-transactional systems.

2.3.1 Isolation levels in transactional distributed databases

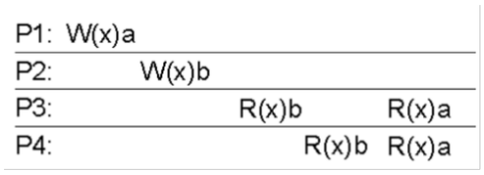
Transactions define groups of operations (reads and writes on data items) and, in any database system, must provide ACID guarantees:

- Atomic: a transaction either entirely succeeds or entirely fails
- Consistency: a transaction does not violate application correctness constraints
- Isolation: transactions do not interfere with each other
- Durability: changes are permanent

When a database system supports transactions, isolation guarantees need to be taken into consideration: they define the behaviour of the system when transactions are issued concurrently. The gold standard in isolation is "serializability" [31]. A system that guarantees serializability can process transactions concurrently but guarantees that the final result is equivalent to what would have happened if each transaction was processed individually, one after other (in a global order, as if there were no concurrency) [32].

Strict Serializability adds sequential ordering to serializability: if transaction B starts after transaction A completes, then B will be committed after A in the final execution order. Therefore, unlike serializability, Strict Serializability guarantees that if a transaction T1 writes X and commits, any subsequently started transaction T2 (in real-time) is guaranteed to observe T1's update on X.

Snapshot Isolation was proposed by Berenson et al. [33]. A transaction executing with Snapshot Isolation always reads data from a snapshot of the (committed) database taken before the transaction started. When the transaction is completed, it will only commit if the values affected have not changed since the initial snapshot, otherwise it will abort. This feature, called "First-commit-wins", prevents lost updates from happening at all [34].



P1:	W(x)a			
P2:	W(x)b			
P3:		R(x)b		R(x)a
P4:			R(x)b	R(x)a

Figure 2.1: Sequential Consistency coherent schedule.

2.3.2 Consistency guarantees in non-transactional distributed databases

An alternative to ACID is called BASE [35]:

- Basic Availability
- Soft-state
- Eventual consistency

Rather than requiring consistency after every transaction, it is enough for the database to eventually be in a consistent state. It is OK to use stale data, and it is OK to give approximate answers, if that's not the primary concern of the application and if the system aims at being responsive at any point in time [36]. Most NoSQL database systems provide single-access type operations, without support for transactions. Each of them ensures a certain level of consistency guarantees, some of them (such as Amazon DynamoDB [26]) allow a fine-grained tuning by specifying the level of consistency for every read operation. Stronger levels of consistency guarantees imply lower performance due to a higher number of messages to be exchanged between the nodes whereas lower consistency levels allow for higher performance and availability.

Strict Consistency has essentially the same properties as "Strict Serializability", except that it refers to the ordering of single operations, rather than transactions. In a Strict Consistency model, all writes are instantaneously visible, and the global order is maintained at any point in time [34, 37, 38].

Sequential Consistency is the equivalent of "Serializability" when referring to single operations. All processes (nodes) must agree upon a global ordering of operations [38] and operations performed within each process must be coherent with that global order. An example is shown in Section 2.3.2. The write performed on item x by P1 is allowed to be delayed: P3 and P4 read value 'b' before reading 'a'.

One application where sequential consistency is usually enforced is shared memory for multi-processor computing [34, 37, 39].

2.3.3 Weak Consistency in non-transactional distributed databases

Weak Consistency does not guarantee the data being the same in all the nodes of the system at any point in time [37, 40, 41]. Eventually, the state will converge, and the nodes will have the same value for all the stored data items, although this convergence status is not guaranteed to exist in any specific moment. Systems that provide Weak Consistency guarantees are availability-oriented since reads are

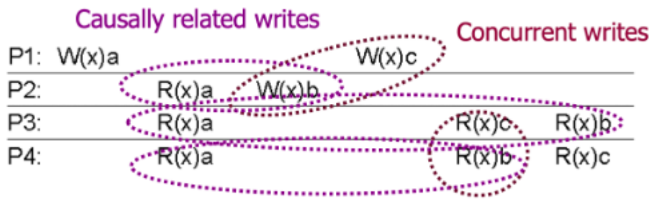


Figure 2.2: Causally consistent program schedule.

almost never blocking [42]. The term "Weak Consistency" refers to precisely one type of consistency, which is eventual consistency. However, in recent literature, [40] the following consistency models are weaker than the previous ones, are not considered "strong", and fall into the "Weak Consistency" class of guarantees.

Causal Consistency weakens Sequential Consistency based on Lamport's notion of happened-before [34, 37] and leverages the concept of causally related operations:

- Writes performed by the same process are causally related (even if they target different data)
- A read operation by a process P on a variable x is causally ordered after a previous write by P on the same variable (even if it's performed by a different process)
- Writes that are not causally related are said to be "concurrent"

Writes that are causally related must be seen by all processes in the same order, whereas concurrent writes (not causally related) may be seen in a different order by different processes. A sequentially consistent schedule is shown in Figure 2.2.

The meaning of the symbols used in fig. 2.2 is:
$W_1(x)$**a** $\equiv$ "*Write, performed by P1, of value '**a**' on item x*".
We identify the "happens-before" relation [37, 38] and represent it with a rightwards arrow ($\rightarrow$).

$W_1(x)$**a** and $W_1(x)$**c** are causally related since they are performed by the same process. Therefore, $W_1(x)$**a** $\rightarrow W_1(x)$**c**.

$R_2(x)$**a** implies that $W_1(x)$**a** $\rightarrow W_2(x)$**b** . Therefore, every process in the system must read 'a' before 'b'.

$W_2(x)$**b** and $W_1(x)$**c** are concurrent, and can be seen by other processes in any order.

FIFO Consistency is another consistency model where writes done by a single process are seen by any other process in the order that they were issued. Writes performed by different processes are, instead, not guaranteed to be ordered.

2.4 Ensuring consistency in distributed cloud storages

Ensuring certain levels of consistency in the presence of multiple copies of data items is undoubtedly challenging. Over the years, there have been numerous implementations of systems that ensure the consistency guarantees and isolation levels described in Section 2.3. This section will focus on how the guarantees are achieved in real deployment systems, with a particular focus on Amazon DynamoDB [26] and Apache Cassandra [28], which are the systems that we take inspiration from in Chapter 3 and Chapter 4.

2.4.1 Replication

Replication involves the maintenance of multiple copies of items in different nodes of a system. It is widely used for different reasons, such as performance, high availability and fault tolerance [34]. Replication enables increased performance, for instance, when caching copies of popular data items, which can be read from faster memory. Naturally, if the items are read-only, caching is trivially achievable. On the other hand, modifiable items pose different challenges, since consistency between cached values and primary copies must be guaranteed (usually with the use of ad-hoc protocols such as Write Back or Write Through [43]).

Replication is also used to achieve higher availability and fault tolerance in the presence of crashes, as copies of data are stored in other nodes, decreasing the likelihood that all the nodes containing certain items could be unavailable. To achieve replication, two are the main messaging approaches, each of them being beneficial in different scenarios: Anti-entropy and Gossiping.

A – Anti-Entropy The master replica initiates replication by choosing at random the target nodes. Eventually all replicas will receive the update. It was proposed in the Bayou system [34] and described as a one-way operation between pairs of servers to propagate writes. Through the exploitation of log files, it can maintain the order of writes.

B – Gossiping It is the dominant messaging approach used by highly available deployment systems. Each propagation triggers another one towards other replicas. If the update has already been received, the probability to gossip further is reduced. It allows the propagation to be fast but the guarantees of reaching all the replicas are only probabilistic. While the gossip architecture can be used to achieve sequential consistency, it is primarily intended to deliver weaker consistency guarantees and therefore high service availability [34]. Advantages of Gossiping include [44]:

- *Simplicity.* Often, gossip protocols require just a few lines of code and are entirely symmetric: every node runs the identical code

- *Bounded load on participants.* Many classic (non-gossip) distributed protocols are criticized because they can generate high surge loads that overload individual components. Gossip is generally used in ways that produce strictly bounded worst-case loads on each component, eliminating the risk of disruptive load surges.
- *Topology independence.* If running on a sufficiently connected networking substrate, and with sufficient bandwidth, a gossip protocol will often operate correctly on a great variety of underlying topologies.
- *Ease of local information discovery.* Many gossip protocols are used for purposes of discovery, for example to find a nearby resource. They would typically find local information with fixed costs: perhaps, a constant, or a delay that is logarithmic in the system size.

It is also called "Lazy Replication" and various implementations are available.

Ladin et al. [45] define a basic version of lazy replication: replicas maintain a local log that contains both updates performed locally, and updates gossiped by other nodes. Update operations are recorded in this log if they cannot be applied straight away (because of consistency constraints or replication constraints). Replicas also maintain a local timestamp to keep track of updates (a vector clock [37, 38] of the type $t =< t_1 \ldots t_n >$, with n being the number of replicas). The gossip architecture itself does not specify when replica managers exchange gossip messages, therefore, a robust update-propagation strategy is needed if all replicas are to receive all updates in an acceptable time. Decisions must be made regarding frequency of gossiping messages and policy for choosing a partner replica with which to exchange gossip.

Replication can be enforced through different protocols/strategies. Two approaches stand out: Primary-based protocols and Quorum-based protocols.

C – Primary-based protocols Primary-based protocols is a family of protocols that involve one master replica to exist per every data item (single-leader).

The primary-based approach to consistency protocols can further be split into two classes: remote-write and local-write. In remote-write protocols, writes are possibly executed on a remote replica and only the information needed to perform the write are sent by the master replica to the secondary replica.

In local-write, writes are always executed on the local (master) replica and the updated data is transferred afterwards, in an undefined moment in the future [37].

D – Quorum-based protocols Protocols of this family are designed to withstand network partitions (isolation of subgroups of replicas). The goal is to ensure that each operation is carried out in such a way that a majority vote among the available replicas in the subgroup where the request was issued is established. Usually, there is a distinction between read quorum and write quorum [20, 34, 37].

One of the most popular quorum-based protocols is known as Reat One Write All (ROWA), where reads do not require a quorum of votes but can be performed directly at one replica, whereas writes require an absolute quorum of all replicas [37].

In Apache Cassandra [28], every replica is responsible for a subset of the data items. The system routes write requests to the replicas responsible for the target items and waits for a quorum of replicas to acknowledge the completion of the writes. A similar quorum-based protocol is also used for reads, when the consistency guarantees required by the client are stricter than "eventual": the system routes the requests to all responsible replicas and waits for a quorum of responses.

Amazon DynamoDB [26] uses consistent hashing and quorum-based replication, as well [46]. Every node is responsible for a set of items and every item has a "preference list" (a set of nodes that must agree to create a quorum upon writing). The peculiarity of Dynamo is that it allows concurrent writes to happen, de facto allowing inconsistencies, to provide the highest possible availability. These inconsistencies are detected through vector clocks [37, 38] and anti-entropy algorithms, and exposed to the application, which resolves them. Through its fast asynchronous replication, read repair and anti-entropy mechanisms, Dynamo quickly converges to a consistent state [20].

2.5 Elastic scaling of cloud-based applications

Cloud-based applications are known for being multi-component. The more the components, the more complex the applications is to scale, since the potential bottlenecks could be of different nature (i.e., the underlying database, the web servers, the load balancers, etc.). Nowadays, cloud providers include the so-called "elastic scaling" mechanism in most of their services.

This section gives an overview on what is elastic scaling and discusses existing solutions to achieve it.

Elastic scaling is the ability of a distributed system to autonomously (without any manual intervention) add or remove new computational units in order to maintain defined performance indexes when facing workload changes. The identification of the right amount of resources to lease in order to meet the required Service Level Agreements, while keeping the overall cost low, is not an easy task, since components are of different nature and some of them are not straightforward to redimension. Consequently, after the identification of the components to scale, a fundamental issue needs to be addressed firstly: whether the component is stateful or stateless [47]. Generally speaking, stateful applications store the state (which can be any kind of information required for the correct functioning of the service) inside the computational nodes, whereas stateless applications rely on separate storage that is shared between the nodes.

Storing state locally inside the service nodes allows for faster access of information. Such an approach is, however, also prone to unavailability in case of server faults. Elastic scaling of stateful applications, also called scaling “up”, requires attention and more resources, given that data needs to be redistributed and nodes need to be sized accordingly.

In comparison, lightweight nodes of a shared-nothing architectures can be added and removed without much effort, due to their nature of not needing to communicate between each other nor being responsible for a specific subset of data. This approach is called scaling “out”. It usually sacrifices latency to guarantee high availability, since any node could process any request at any point in time.

2.5.1 Scaling UP vs scaling OUT

Stateless cloud applications store data in separate and efficient storages, shared among each server, leveraging the use of containers and microservices to keep the computing nodes as small as possible.

This approach allows to increase computational power by scaling “out” (adding additional computational nodes/servers) as opposed to scaling “up” (adding more resources to the fewer existing server machines) [48, 49]. Some of the benefits of scaling out are:

- Increased fault tolerance: a larger pool of servers is less affected by failures of single machines as availability is proportional to the number of active nodes.
- Less overhead: routing requests to specific servers requires a load balancer / proxy that has knowledge about the request-server associations. It clearly introduces a delay in processing any request. If every server can process any request, the routing decision is eliminated.
- Increased capacity: upscaling fewer machines is limited, as one cannot add an infinite number of resources and the need to add more nodes is inevitable when the workload increases dramatically.
- Easier and better scalability: Infrastructures that supports a stateless application does not require almost any data transfer when scaling out, since the servers are independent. Therefore, scalability becomes almost linear and is limited only by the capacity of the back-end storage layer hosting the application’s data.

Solutions for scaling stateful components are of interest to this thesis, as the IaaS cluster that is going to be managed and scaled is a cluster of stateful nodes. Therefore, we now discuss solutions to implement upscaling.

2.5.2 Solutions for scaling stateful components

On the other end of the spectrum, stateful services are what really make elastic scaling a challenge. If upscaling fewer machines is the more straightforward approach, scaling out is at some point inevitable, due to the aforementioned reasons. Distributed databases, given their stateful nature, are a critical system to scale [20, 48, 50, 51].

When adding nodes that store significant amounts of data, complexity increases dramatically: data needs to be redistributed and new cluster members need to be brought up to date in order to serve requests, possibly by existing members [49].

As the number of nodes in the system grows, the performance of distributed databases exhibits clear nonlinear behaviors. Such behaviors are imputable to the simultaneous, and often interdependent, effects of contention affecting both physical and logical (conflicting data accesses by concurrent transactions, resulting in increasing amounts of aborted operations) resources. As a consequence, capacity planning (which, when talking about autoscaling, is performed automatically by the cloud provider) also becomes more complex, since it depends on the forecast of resource utilization.

The following research papers address these issues with different approaches and for different types of systems.

2.5.2.A Transactional Auto Scaler

Transactional Auto Scaler (TAS) [52] is a system for automating the elastic scaling of replicated in-memory transactional data grids such Red Hat Inifinispan.

TAS introduces a novel performance prediction methodology based on the joint usage of Analytical Modelling (AM) and Machine Learning (ML) models. By exploiting the strengths of both technologies, TAS is able to accurately predict the performance of the system and consequently size the architecture.

The system collects statistics concerning load and resource utilization across the set of nodes in the data grid via a distributed monitoring system. Statistics are then aggregated and fed to the load predictor that forecasts the workload volume and its characteristics while also detecting significant workload shifts, discarding temporary and irrelevant changes.

The hybrid AM/ML performance predictor, then, takes in input the workload characteristic and the platform scale and outputs several Key Performance Indicators (KPIs) that are used by another component, called “Service Level Agreement (SLA) enforcer”, to identify the optimal platform configuration (in terms of number of nodes) based on user-specified Service Level Agreement and cost constraints. Finally, an actuator applies the SLA enforcer changes on the infrastructure.

2.5.2.B Elastic Stateful Stream Processing in Storm

Another recent work that explores elasticity in stateful services was proposed by Cardellini et al. in 2016 [53]. It is an extension of Apache Storm, a popular Data Stream Processing (DSP) system.

Data Stream Processing applications are often deployed on a fixed number of nodes and do not provide automatic scaling capabilities. Therefore, to support workload fluctuations, the user determines the number of parallel instances for the operators on the expected maximal workload, either achieving average under-utilization of the system, because load peaks can rarely occur, or being unable to manage a bursty workload (a sort of fixed capacity planning).

The autoscaling solution presented in this paper is achieved by automatically adding parallel instances according to a scaling policy. A centralized component, called Elasticity Manager, periodically increases, or decreases, the number of processing instances for a a given computing stage, improving resource utilization in relationship to the incoming data rate. Then, the Storm built-in scheduler, decides in which physical nodes the new instances will be executed. The scalability policy is threshold-based: within any computing stage, a new processing instance for each one in overload. Scale-In, instead, halves the number of processing instances if all of them are underloaded.

2.5.2.C Apache Cassandra

Another example of scaling for stateful services such as key-value stores is provided by Apache Cassandra [28], a distributed NoSQL key-value store created by Facebook to achieve availability and linear scalability, despite its stateful nature.

In Cassandra, nodes are responsible for a range of item keys (see Section 2.7). When a new node is added into the system, it splits a range of keys that some other heavily loaded node was previously responsible for, alleviating it. The node giving up the data streams the data over to the incoming new node on the fly, using kernel-kernel copy techniques.

As demonstrated [54], Cassandra scalability is linear with a correct and optimal keyspace configuration, which make capacity planning straightforward once the workload is known. Despite Cassandra not scaling autonomously, it is relatively simple to devise a system that, given a correct workload prediction, decides the correct sizing of a Cassandra cluster. For instance, if the sample workload is X requests/s, and one chooses to use nodes with computational power up to y $(y < X)$ requests/s, given the linear scalability, $n = \left\lceil \frac{X}{y} \right\rceil$ nodes will be necessary. The same reasoning should be done regarding storage capacity (despite most of the workloads being CPU-bound before becoming memory-bound [23, 54]).

Good practice parameters for the choice of the correct VM type (assuming for simplicity that a cluster is made of nodes with the same characteristics) have been presented in Section 2.2.2.

2.6 Data Migration

One additional issue with scaling stateful services regards data migration. Adding machines to a stateful component requires, as said before, transferring data to involve the new nodes in the cluster and to make them able to process requests. One of the challenges of data migration is the ability of the system to keep serving requests by having the minimum possible downtime and by using the lower possible amount of resources [55].

2.6.1 Albatross

Albatross [56] proposes a live migration technique for multitenant databases to ensure SLAs to the tenants whenever they get overloaded. Albatross's migration protocol is divided into phases.

Phase 1 starts with a snapshot of the source database cache (or of the one part that needs to be migrated) (*src*) that is then transferred to the destination VM (*dst*) without service interruption. Since *src* continues serving transactions while *dst* is initialized with the snapshot, the cached state of *dst* will lag that of *src*.

In Phase 2 (iterative phase), at every iteration, *dst* tries to "catch-up" and synchronize the state of *src*. *Src* tracks changes made to the database cache between two consecutive iterations. In iteration i, changes made to *src*'s cache since the snapshot of iteration $i-1$ are copied to *dst*. This phase is terminated when approximately the same amount of state is transferred in consecutive iterations or a configurable maximum number of iterations have completed.

Phase 3 (Atomic Handover) is where the exclusive read/write access of *src* (called ownership) is transferred from *src* to *dst*. *Src* stop serving tenant's requests and the final handover is performed before the service comes back live. The successful completion of this phase makes *dst* the owner and completes the migration.

Most importantly, Albatross guarantees serializability after migration for transactions that started before and during the migration process, by transferring the lock table to the destination, as well.

2.6.2 Zephyr

Zephyr [57] is a similar system focused on shared-nothing architectures (each node uses its own memory and does not share it with other nodes), which makes it suited to stateful services. The critical innovation of Zephyr is the "dual mode", where, during migration, both *src* and *dst* execute transactions. It also uses an iterative approach, almost identical to the Albatross one, to face the rather extended unavailability of the classic "stop and copy" approach.

2.6.3 Slacker

Both Albatross and Zephyr require modifications in the database engine. For this purpose, Barker et al. proposed Slacker [58], which approach is unique in that it operates on off-the-shelf database systems using readily available open-source tools and does not require modifications of the database engine.

Another characteristic of Slacker is the use of a middleware which is absent in the previously described data migration systems.

Slacker works with MySQL databases and leverages existing MySQL live backup tools to create a snapshot of the database without interrupting the service. Then, it starts a new MySQL instance and initializes it from the snapshot. The rest of the migration is performed with the same iterative approach explained before. Slacker takes advantage of the slack resources in a cloud VM that is hosting database tenants to perform the migration, while also making sure that the SLAs of all those tenants are not violated.

2.6.4 ShuttleDB

Another rather innovative approach to data migration is proposed in ShuttleDB [59]. It also does not introduce any added complexity to the database engine, and combines VM elasticity with lower-level, database aware elasticity to provide efficient database elasticity both within and across cloud data centers. After identifying when to initiate elastic scaling, which tenants to migrate, and where to move the tenants, ShuttleDB automatically chooses the “best” elasticity mechanism for each elastic operation on a given tenant.

2.7 Data Placement

In all the previously described systems, data is assigned to nodes to achieve optimal load balancing, lower access latency, and higher fault tolerance [60, 61]. The goal of this thesis and the innovative approach is to optimize data placement based with cost-efficiency as ultimate goal, and based on the data access patterns, which has not often been a concern to cloud providers or researchers. The principal distinction between data placement systems/algorithms that applies to most of the related work is regarding whether the workload patterns are known or not: Workload-Oblivious vs Workload-Aware data placement.

2.7.1 Workload-Oblivious Data Placement

One widely used approach to distribute data among several nodes, used by state of the art key-value stores [26,26,28,62] is consistent hashing. The system assigns ranges of keys to the nodes upon joining the cluster, by virtually placing nodes on a virtual ring. When there is an incoming request for a new key, a hash is generated for it and is mapped on the circular ring as well, in a circular distributed hash table fashion. The node that is next to the key hash is responsible for the storage of that key.

The principal advantage of consistent hashing is that the departure or arrival of a node only affects its immediate neighbours in terms of key range and other nodes remain unaffected. However, this basic form of consistent hashing can cause non uniform data assignment and uneven load distribution as there is a possibility that a majority of hashed data gets mapped to a single node. To solve this, DynamoDB and Cassandra add virtual nodes in the hash ring where one physical node can represent multiple virtual nodes by using multiple hash functions. This leads to an enriched hashed ring where data is evenly distributed across the nodes.

With consistent hashing alone, the developer is responsible for assigning similar keys to related data in order for it to be placed in the same server. This can be cumbersome, and it is the main reason more advanced systems leverage data streams and real time analysis to automatically place data in the "best" locations. Therefore, systems such as Bitgable [30] and PNUTS [63], make use of a centralized directory system. The scope of data placement is once again load balancing, but with additional goals such as geographical placement to lower the response time.

2.7.2 Workload-Aware Data Placement

When workload is known (e.g., the application has already been running for some time and an optimization of the placement is performed), the data placement problem can be informally described as follows: given a particular workflow, the current data placement, and a particular infrastructure, find the proper position(s) of data within the infrastructure to optimise one or more certain criteria, such as the cost of the data transfer [21, 61].

2.7.2.A Schism

An advancement in real time data placement has been made by MIT researchers that focused on OLTP (online transaction processing in relational SQL databases) in a distributed shared-nothing architecture. They proposed Schism [64], which leverages the fact that throughput of distributed transactions is clearly worse than if involved data were placed on the same node, due to several more messages required to avoid distributed deadlocks, implement distributed joins, etc.

Schism uses a graph-based approach to represent a database and its workload, where tuples are represented by nodes and transactions are represented by edges connecting the involved tuples. The system applies graph partitioning algorithms to find non-overlapping and (ideally) balanced partitions that minimize the weight of cut edges, Then, Schism replicates the tuples that are shared between transactions in order to maximize performance by exploiting data locality and minimizing the number of distributed transactions. This approach is, however, very workload-specific since it is only efficient for similarly characterized workloads.

2.7.2.B Ursa

A recent research presented Ursa [65], a scalable data management middleware system, which also makes use of a workload driven optimization strategy. Ursa's target systems are distributed object stores, such as Google File System[9] and its primary concern is to relieve highly utilized nodes by efficiently migrating data to less utilized ones. Ursa periodically solves an Integer Linear Programming (ILP) problem to relieve the overloaded nodes by either switching the role of the primary replica with one of the secondary replicas or by physically migrating the data to other nodes.

2.7.2.C AutoPlacer

One system that leverages real time data characteristics to move data and obtain an optimal workload-aware placement is AutoPlacer [66].

AutoPlacer works with NoSQL key value stores such as Red Hat Infinispan[10] and Apache Cassandra [28] and aims at optimizing the placement of only those items that are deemed critical for the system performance, which are the ones that generate the largest number of remote operations (the more expensive to perform). For the placement of the remaining items, an approach based on consistent hashing is used.

AutoPlacer also designs a new data structure for the directory system to track data among the nodes. It makes use of a round-based distributed optimization algorithm: in each round, the system decides on which nodes to place the top-k critical items in order to increase the correlation between the data each node is requesting and storing.

In order to be able to identify the top-k hotspots of each node with low processing cost, AutoPlacer adopts a state of the art stream analysis algorithm to identify the most frequent items of a stream. An ILP problem is then instantiated to find the optimal placement of only the top-k items, to significantly reduce the number of decision variables.
Other data placement goals are relevant, such as security-oriented placement and fault-tolerance-based placement, and mechanisms and algorithms to implement them have been explored [67]. They are, however, outside of the scope of this thesis and therefore not mentioned.

2.8 Linear Optimization

This section initially gives the reader a background on Linear Programming (LP) and ILP, extensively used in the formal modelling of the data placement problem presented in Section 4.3.1. The section will then give an overview of some of the algorithms that are used to solve such problems, and will describe

[9]https://static.googleusercontent.com/media/research.google.com/en//archive/gfs-sosp2003.pdf
[10]https://infinispan.org

available solvers that are useful to compute solutions for complex LP and ILP formulations. In particular, the section will focus on Gurobi, which has also been used in the implementation of the product of this dissertation.

Mathematical optimization seeks to find the best solution for problems in which the quality of any possible solution can be expressed as a numerical value. Such problems arise in all areas of business, sciences, engineering, economics, management, and many more, and can be highly complicated. However, linear programming aims at depicting such relationships as a set of linear inequations, thus, making it easier to analyze them [68]. A mathematical optimization model consists of an objective function and a set of constraints expressed as a system of equations or inequalities [69].

The goal of linear programming is to either maximize or minimize the value of the objective function, while at the same time ensuring that all the constraints are satisfied. Both objective functions and constraints are functions of the decision variables, i.e. those variables that are unknown at the time of instantiating the problem, and which value will be decided upon problem resolution. Linear programming applications have been seen across many business types, for example in Production and Operations Management: quite often in the process industries, a given raw material can be made into a wide variety of products. Given the profit margin on each product, the problem is to determine the quantities of each product that should be produced in order to maximize the overall profit. The decision is subject to numerous restrictions such as limits on the capacities of various operations, raw-material availability, demands for each product, and any government-imposed policies on the output of certain products [69].

Integer Linear Programming is an extension of Linear Programming where some (Mixed Integer Linear Programming (MILP)) or all (ILP) the decision variables must be integer and not continuous. The reason behind ILP is that, in some real use cases, decision variables represent quantities that cannot be expressed by non-integer numbers, such as numbers of products to output. Alternatively, variables could model decisions in the form of 0-1, such as in placement problems, where all decision variables represent possible placements and the values correspond to whether resources should be placed in such locations or not.

An example of a use case of ILP problems can be found in the field of computer science, and, in particular, in distributed system scenarios. The problem is regarding data placement, where 0-1 binary variables might represent all the possible placements of specific data items, and the goal could be to minimize resource usage or allocation cost, or even to maximize locality, as it is the case in some of the systems described in Section 2.7.2.

2.8.1 Algorithms for solving (I)LP problems

Non-approximate methods for solving LP problems, i.e. methods that are guaranteed to find an existing optimal solution, have been researched [69], here are some of the most most used ones:

- **Graphical method**: it is only suitable for small sized LP problems with 1 or 2 decision variables. The method consists of drawing the constraints as slopes on a Cartesian plane. Each slope divides the plane in a feasible and unfeasible region. The subplane which contains points that do not satisfy the constraint is unfeasible and must be blacked out. Once all the constraints are drawn, the remaining convex area of the plane is the feasible region. In order to find the best solution, it is sufficient to draw one of the infinite lines corresponding to the objective function (parametric with respect to the decision variables) and to translate it, parallel to itself, towards the edge of the feasible region. In case the problem is a minimization problem, the line should be translated in the direction that lowers the value of the function. The opposite applies for maximization problems. Once reached the end of the feasible region, its extreme point is the value of the decision variable(s). If the translating line ultimately reaches an edge instead of a vertex, any point on the edge is the optimal value of the decision variable(s). When the number of dimensions (decision variables) is at least three, working with three-dimensional and multi-dimensional planes is certainly not feasible from a human perspective. For this purpose, algorithms to solve more complex problems exist.

- **Simplex Algorithm**: The Simplex algorithm, developed by George Dantzig [70], solves LP problems by constructing a feasible solution at a vertex of the polytope (an n-dimensional polyhedron representing the feasible region) and then walking along the edges of the polytope towards vertices with non-decreasing values of the objective function, until an optimum is guaranteed to be reached. In many practical problems, the algorithm might stall or cycle around the vertices without the objective function improving in value. In practice, the simplex algorithm is quite efficient and can be guaranteed to find the global optimum if certain precautions are taken [71].

- **Interior Point Algorithms**: Many other algorithms explore the polytope by traversing it internally, rather than walking its edges. Some of the algorithms belonging to the Interior Point class are: Ellipsoid Algorithm [72], Projective algorithm of Karmarkar [73], and the most recent Current Matrix Multiplication Time algorithm [74].

Solving ILP problems incurs in increased complexity, as the feasible region is not a continuous space but rather a discrete, n-dimensional space. Similarly to LP problems, many algorithms (although more complex) have been studied in order to solve ILP and MILP problems. The most straightforward way to solve such problems is to compute the LP-relaxation, obtaining an identical problem where the decision variables are allowed to be continuous. Then, the LP problem is solved and the decision variables are eventually rounded to the nearest integer. This implies that the solution is, in most of the cases, not optimal and, in some cases, not feasible.

More complex algorithms have been researched and can be collected in two main families:

- **Exact Algorithms**: they are guaranteed to produce an exact optimal solution. The most popular algorithms of this family are:
  - Cutting Plane Methods, which initially solve the LP relaxation. Afterwards, instead of simply rounding the values to integer, these algorithms add linear constraints that drive the solution towards being integer, without excluding any integer feasible points from the analysis [75].
  - Branch and Bound Methods are based on a systematic enumeration of candidate solutions by means of state space search: the set of candidate solutions is thought of as forming a rooted tree with the full set at the root. The algorithm explores branches of this tree, which represent subsets of the solution set. Before enumerating the candidate solutions of a branch, the branch is checked against upper and lower estimated bounds on the optimal solution, and is discarded if it cannot produce a better solution than the best one found so far by the algorithm [76]. Branch and Bound Methods can be (and often are) slower than cutting plane methods. In the worst case they require effort that grows exponentially with problem size, but in some lucky cases the methods converge with much less effort [77]. This is why some algorithms combine Cutting Plane algorithms and Branch and Bound algorithms to deliver better performance.
- **Heuristic Algorithms**: Given that ILP is NP-hard, heuristic methods, which are not guaranteed to converge to a solution, nor to find the optimal solution, are often used. Some of the most used heuristic algorithms are: Hill Climbing [78], Simulated Annealing [79] and Tabu Search [80].

2.8.2 Tools for defining and solving (I)LP problems

We have seen how solving LP and, especially, ILP problems can be highly challenging, even when using advanced algorithms. Thankfully, there exist commercial programs that implement such algorithms and help us solving any problem without the need of making complex decisions ourselves.

Most of the solvers start the optimization with what is usually called the “Pre-solve” phase. During this phase, the non-linearly independent constraints are removed and other static optimizations are performed, in order to reduce the size of the problem as much as possible. This phase is suited to be implemented with multi-threading, as multiple work units can be assigned subsets of the problem to reduce. After the Pre-solve phase, solvers typically choose among the available built-in algorithms. Some of them implement their own variations and improved versions of the algorithms described in section 2.8.1.

The most popular solvers are:

- **IBM®ILOG® CPLEX®**. It is a software package developed in 1988 and actively developed by IBM [81]. It offers interfaces for common programming languages such as Java, C++, C# and

Python. CPLEX is able to solve LP, ILP, and Quadratic Programming (QP) by using algorithms like Simplex, Cutting Planes, Branch and Cut, Branch and Bound and more [82]. Although a free of charge version of CPLEX is available, it is limited to solving problems with 1000 variables and 1000 constraints, and it does not offer the mentioned APIs for common programming languages (only a proprietary language can be used). Paid subscriptions can be purchased for little less than 200 € per month and offer the possibility to run optimizations directly inside IBM Cloud, unlimited amounts of decision variables/constraints and more advanced features.

- **Gurobi**. It is, according to the official website [83], “The fastest and most powerful mathematical programming solver available for your LP, QP and MIP (MILP, Mixed Integer Quadratic Programming (MIQP), and Mixed Integer Quadratically Constrained Programming (MIQCP)) problems”. It offers APIs for a wide variety of languages and frameworks, among which: object-oriented interfaces for C++, Java, .NET, and Python, matrix-oriented interfaces for C, MATLAB® and R, and links to popular mathematical programming modeling languages such as AMPL. Gurobi offers both free and premium licenses, as well as proprietary cloud computing offerings. It also offers an academic license, which includes all the features of the paid version, but free of charge for members of the academic community. The available algorithms include the Simplex algorithm [84] and other methods, such as Parallel Barrier with Crossover and numerous heuristics [85], with Simplex being the first choice for LP problems.
- **COIN-OR**. It is a completely open source project that contains various solvers. The most renowned are CLP (for solving LP problems), CBC and Symphony (which use Branch and Cut for solving ILP problems), and PuLP (a Python framework for solving LP and ILP problems) [86].

The solver of choice for this dissertation is Gurobi, which will be described in depth next.

2.8.3 The Gurobi Solver

Given its open source nature, we initially experimented solving problems with PuLP (part of the COIN-OR project). However, after a short period of experimentation, the choice has been made to use Gurobi as the main solver for this dissertation. The reason behind this choice is purely related with performance. When solving large problems with many decision variables, Gurobi is the far superior solver, according to our personal experience. This difference in performance is also shown in recent benchmarks [87]. We implemented the mathematical formulation of the problem (see Section 4.3.1) in Python, using the Gurobi library.

The formulation of even complex problems is actually straightforward, since the transcription is facilitated, through the extensive API offering, which allows the translation of every type of mathematical expression.

Creating an ILP problem with Gurobi consists of simple and ordered steps:

1. Importing the Gurobi module
2. Instantiating a model object
3. Defining the decision variable objects.
4. Adding the objective function to the model object, as a set of linear expressions that include decision variables.
5. Adding the constraints to the model object, as a set of expressions that also include decision variables.
6. Eventually setting optimization parameters.
7. Calling the `optimize()` function on the model object.
8. Reading the decision variables' values through the X attribute, which can only be called after solving the problem. It will return the value that has been assigned by the solver.
9. Retrieving the value the objective function, directly from the model object.

Aside from the basic functionalities, Gurobi offers advanced features such as the possibility to tune numerous parameters to achieve results in either faster or more accurate ways, by putting upper bounds on solve time, reducing memory usage, and many more. Some relevant advanced features are listed below:

- When ILP problems are particularly complex or large and therefore computationally hard to solve, Gurobi can perform an automatic non-trivial relaxation of the problem. The relaxation transforms integer variables into continuous variables, and removes certain constraints [88].
- Additional parameters that can control the operation of the solver can be set by the user. Some examples are:
  - `TimeLimit`, sets an upper bound on the solve time, after which the solver will terminate and use as solution the best one found before the time expired.
  - `NodefileStart` is used when the amount of memory used for the optimization exceeds the specified parameter value. At that point, nodes are compressed and written to disk.
  - `MIPGap` sets a lower bound on the Gap (in percentage) of the solution w.r.t. the optimal solution. The Gap is non other than the difference between the current best explored value of the objective function with respect to the known optimal value. When the Gap is lower than the set threshold, the solver will stop looking for a better solution. The complete list of parameters can be found here: [89].

- An additional high level feature is the possibility to easily express non-usual constraints of the type: $z = f \implies a^T x \leq b$, also informally called “if-then constraints”.

Once an optimization is terminated, the solver stores the status of the optimization in a Status variable. Possible statuses are “OPTIMAL”, “INFEASIBLE”, “TIME_LIMIT”, for when the optimization terminated because the time expended exceeded the value specified in the `TimeLimit` parameter, and many more. Additional optimization statuses can be found on the Gurobi documentation [90].

3

Architectural View of PlutusDB

Contents

This chapter introduces PlutusDB: its architecture and internals. It then describes the execution flows of the system, along with an analysis of its internal components. Finally, we present a proposal for the implementation of a data transfer algorithm.

When deploying database systems as IaaS, the choice of the underlying infrastructure is always a critical point. Major cloud providers offer a wide range of options for nodes of a distributed infrastructure, which implies that making an optimal, cost-effective choice is a hard task.

Cloud architect often choose to deploy an underlying infrastructure composed of the same types of nodes, as choosing multiple types of nodes requires performing complex evaluations that often become an overhead. Deploying multiple physical instances of the same type is also easier. However, choosing which type offers the best price/performance ratio according to the expected application load model can still be an overwhelming task.

One downside of having the same node type across the whole infrastructure is that, in face of the need to store an unbalanced set of data where access characteristics varies greatly among data items, using only one node type might be sub-optimal regarding resource usage, and therefore cost efficiency. Consequently, an ideal solution, in presence of an unbalance data set, would be to use a non-uniform infrastructure and to place each data item in the right node type that is specialized for its usage pattern. For instance, throughput-oriented nodes should be more cost efficient for highly popular items (the ones that generate the highest amount of bandwidth).In contrast, they should be more expensive than other node types when it comes to storing low-throughput items.

To summarize, provided that the access patterns vary significantly across the data set, the choice of only one node type is likely to incur non-optimal costs for one portion of the data items.

PlutusDB is a novel NoSQL, Key-Value database system that, based on the current data set usage model, chooses the best underlying infrastructure, and then places the items in the right nodes to maximize cost efficiency. The system tries to allocate the most cost efficient infrastructure for any application workload, and, ideally, performs optimizations periodically.

PlutusDB optimizes data placement by having multiple instances of backend databases as underlying infrastructure, each with its dedicated nodes. The idea is that the generated hybrid configuration of nodes, as a whole, might be cheaper than using instances of a single node type to store the whole data set. PlutusDB aims at adopting a customizable approach by letting the end user choose the granularity of the optimisation (single data items, table fragments, semantic item groups based on similar keys/values, etc.) to improve performance and scalability.

3.1 General Architecture

We start with a high level overview of PlutusDB's architecture (illustrated in Figure 3.1).

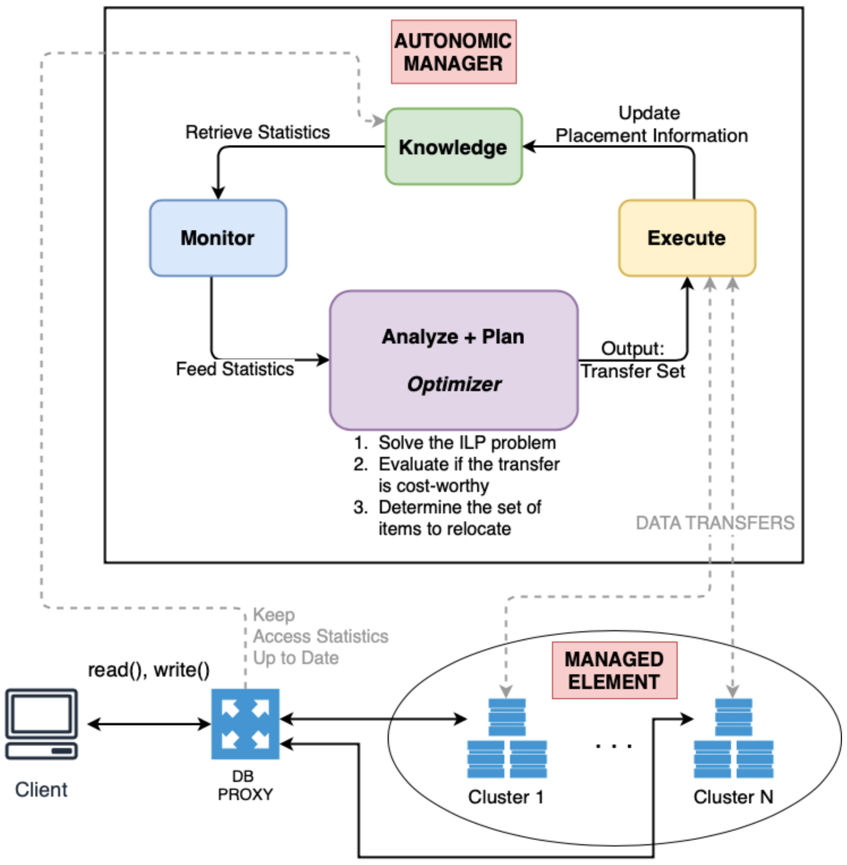


Figure 3.1: PlutusDB Architecture.

This concept represents the vision of the end product, which is a complex system that we do not aim to develop fully. Instead, this thesis will focus on the core component of PlutusDB: the Optimizer, which is the main innovation that implements the idea on top of which the whole architecture is designed.

The used terminology for the components of the system is proposed by Kephart & Chess [91]. PlutusDB comprises three high level components:

- **Managed Element**: it is the set of underlying IaaS sub-clusters, which includes all the backend database instances. Each database instance is self contained and runs on its dedicated hardware infrastructure.

- **Autonomic Manager**: This component is what allows the whole system to be (potentially) self-optimized. It includes 4 sub-components: Knowledge, Monitor, Analyze+Plan and Execute.

Additionally, a third component, the **Database Proxy**, will perform as the entry point for interacting with the database system, it has knowledge about the data placement and dispatches the requests towards the correct backend database, according to where the data is stored at the time of the request.

PlutusDB aims to be a fully autonomic system that periodically evaluates the optimal items' placement that minimizes the overall usage cost. This periodic evaluation, that could also be triggered manually by the database manager consists of a sequence of steps that involve all the following components of the Autonomic Manager:

- Knowledge: Stores the current data placement information and the up to date throughput of the data items.
- Monitor: Retrieves the throughputs from the Knowledge component and performs the necessary transformations to prepare the data (access statistics) for the optimization phase.
- Analyze + Plan: This is what we call the Optimizer. It is a standalone component that can also work on its own. It solves a pre-modelled optimization problem that produces an optimal data placement that minimizes the costs of usage of the cloud infrastructure. This phase should also be in charge of deciding whether any data should be transferred among the backends, although this functionality is not implemented at this stage.
- Execute: Takes as input the list of data to be transferred, output of the preceding stage, and executes the transfer with the algorithm described in section 3.6.

3.2 Execution flows

We can identify two main execution flows in PlutusDB: the first one is the standard usage, where clients perform read/write requests and the system works as a traditional NoSQL KVS.

The second one is where the innovation of PlutusDB relies, and it is a periodic phase aimed at minimizing the operational costs.

3.2.1 Client Requests

Being a Key-Value store, PlutusDB operates with a per-item granularity, offering support to only atomic reads and writes.

The signature of the APIs that PlutusDB offers to the Clients are straightforward and typical of NoSQL databases:

```
value = read(key)
status = write(key,newValue)
```

Where writes either insert the item if the key is not present, or overwrites the value if the key is already stored in one of the backends.

Moreover, writes behave as usual in NoSQL KVS by returning the atomic operation's success status (SUCCESS, FAILURE).

All the requests must go through the Database Proxy, which consults the Knowledge component to get information on the placement of the requested items. The main challenges of keeping an accurate placement are related to performance. Each client's request must be routed towards the right backend where data is stored. To do this, the system must intercept all the requests, consult the Knowledge component, and only then proceed with processing them. This operation implies performing an additional synchronous read previous to each request, which adds a significant delay. Even though some widely used optimisations might be used (such as caching the placement of the most popular items), the Database Proxy and the Knowledge component are crucial attention points, performance-wise. Recent literature [66] proposes advanced probabilistic techniques that leverage Bloom Filters, which might be used in a future implementation to improve the system's performance. In particular, we believe the Database Proxy is the second most crucial component of PlutusDB's architecture after the Optimizer, as its performance directly impacts the system's performance as a whole. The final goal will be for PlutusDB to perform just as well as any other database system running a single database instance.

3.2.2 Optimizations

Optimizations are the focal point of this research.

Each optimization starts with the Monitor component retrieving the most up-to-date data access statistics, stored in the Knowledge component. The Monitor component eventually performs data transformations, preparing the statistics for the next stage: the optimization. At this point, the Optimizer component, upon receiving the statistics, solves an ILP problem, which generates an optimal theoretical data placement (one that does not take into account parameters such as where the data is currently stored or the cost of the current placement). Then, the current placement and the ideal one are compared, and a calculation of the cost required to transfer data between backends (to achieve the ideal configuration) is performed.

Time and cost of items' transfer are outside of the scope of this contribution, and therefore they have not been precisely modelled in this research. However, we provide a general idea for how they should be modelled in an eventual implementation, in Section 3.6.

Transfer time is a function of:

- Total size of the items to transfer
- Network speed
- Amount of incoming requests that target items in-transfer

The reason behind the last point being relevant in estimating the transfer time will also be explained in a more detailed way in Section 3.6.

The data transfer cost is more complex to model, as it depends on the cloud provider. Essentially, the transfer cost is strictly related to the network usage and is also a function of the total size to transfer.

Once the items to transfer are identified, the optimizer hands the list off to the Execute component, which physically moves the designated items. All of the operations (including the items transfer) are ideally performed transparently with virtually zero downtime.

3.3 Knowledge component

The goal of the Knowledge component is to store information regarding the system's situation at any point in time, which includes:

- Current data placement
- Current operational costs
- Current status of the managed database clusters
- Up to date statistics of all the stored items

Widely used systems such as Apache Zookeeper [92] already implement some of these functionalities. Choosing a system as such and extending it ad-hoc by adding all the required functionalities could be another project that is required for creating PlutusDB.

3.4 Monitor component

This component is in charge of retrieving up to date statistics from the Knowledge storage and transform them to make them available for the Optimizer. Statistics must be in a specific format for the Optimizer to work correctly. Specifically, the Optimizer requires:

- An array of item IDs, which must be integers
- An array/dictionary of item sizes, indexed by item ID
- Two equally shaped arrays, containing the read and write accesses per second of each item.

If the Knowledge component does not store statistics in the formats mentioned above, the Monitor component will be in charge of processing them. It has to be noted, however, that the pre-processing of statistics could be unnecessary, depending on the implementation of the Knowledge component, making the Monitor component a trivial step of the optimization flow.

3.5 Analyze + Plan component

The optimization phase is going to be the core contribution of this dissertation. It is the “brain” of the system that orchestrates and directs the data placement and, with it, the data transfers.

Although some ML-based approaches for data placement in distributed systems have been proposed [93, 94], PlutusDB introduces a novel approach that aims at explicitly modelling the cost dynamics of cloud IaaS databases to control data placement purely based on cost efficiency.

PlutusDB’s placement engine is a program that defines and solves a complex ILP problem that, through the minimization of the cost of the overall infrastructure (objective function of the problem), achieves an optimal data placement (decision variable of the problem).

One cost optimization is ideally defined over a period of time T, which represents an estimation of a stability period: a period during which the stored data’s access characteristics are likely to be stable, without significant changes. We can then model the cost saving as the difference between the current cost, billed over T, and the cost of the database with the ideal configuration, also billed over T. However, we must subtract the one-off cost of transitioning to the ideal configuration, in order to achieve a fair estimation of the overall cost saving.

Estimating the value of T could be achieved, for instance, by adopting time-series based techniques [95] to estimate the evolution of the workload characteristics. Note that the current prototype implementation does not integrate such predictor and implicitly assumes that the tracked workload characteristics are constant/stable over time. Under these assumptions, the optimization process is a one-shot problem, it only needs to be instantiated once and the resulting placement policy is guaranteed to be optimal in the future. This is a simplifying assumption that might be lifted in the future to match the needs of dynamically shifting workloads. Numerous other workload predictors, which leverage different techniques, have been proposed over the years, such as the ones presented in [96–98], and many more.

3.6 Execute component

The Execute component is in charge of transferring the data items among sub-clusters to achieve the ideal and cost-optimal configuration, output of the Optimizer component. Our vision is that the Execute component must transfer the items between back-ends in a fully transparent way and with virtually zero downtime. To achieve this, PlutusDB aims at using an incremental transfer procedure, inspired by the algorithms described in Section 2.6, whose formalization and implementation are outside of the scope of this dissertation. However, a proposal of a possible algorithm that might be implemented to achieve the desired goal is depicted in Figure 3.2.

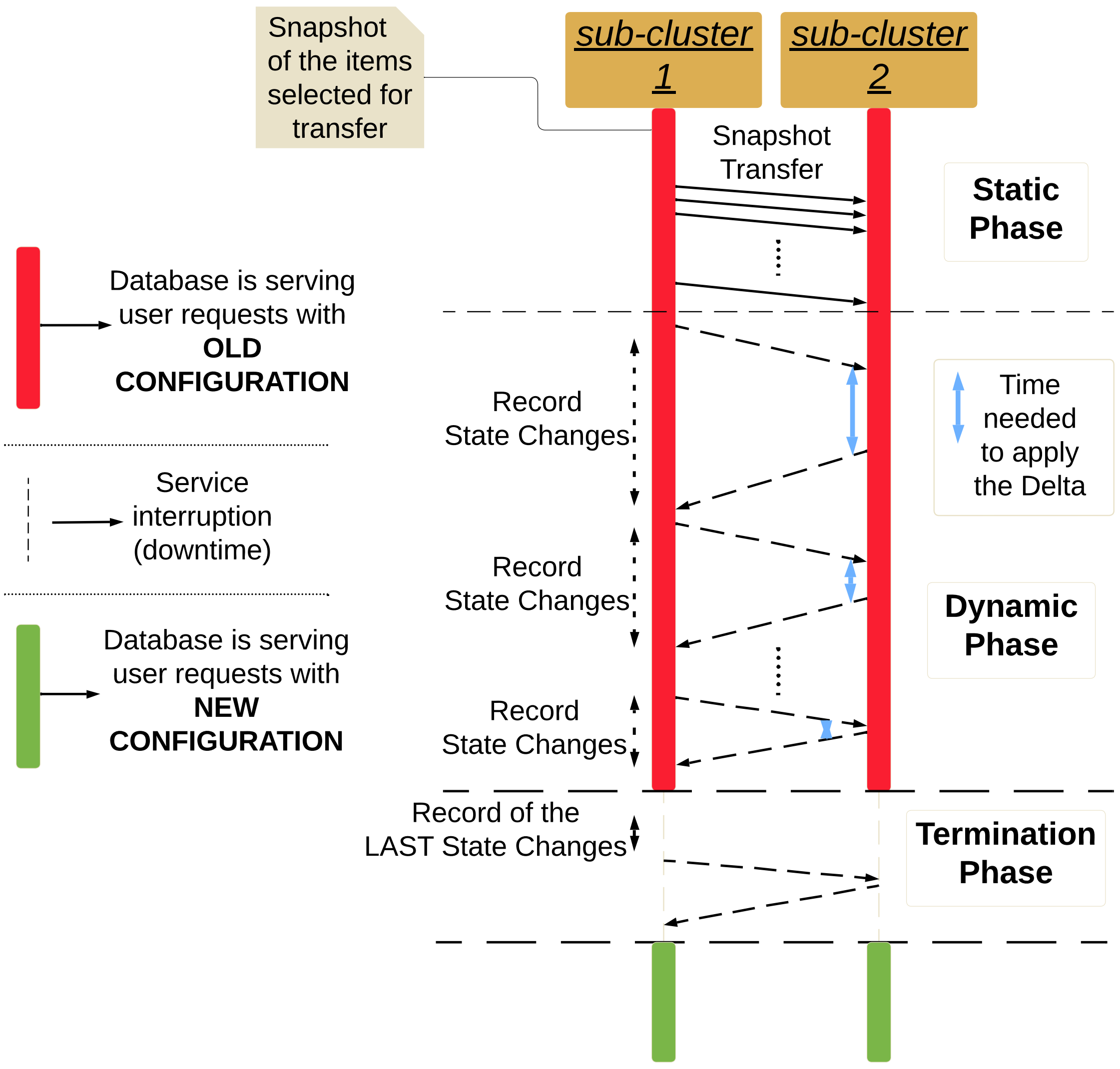

Snapshot of the items selected for transfer
sub-cluster 1
sub-cluster 2
Snapshot Transfer
Static Phase
Database is serving user requests with OLD CONFIGURATION
Record State Changes
Time needed to apply the Delta
Service interruption (downtime)
Record State Changes
Dynamic Phase
Database is serving user requests with NEW CONFIGURATION
Record State Changes
Record of the LAST State Changes
Termination Phase

Figure 3.2: PlutusDB data transfer procedure.

The following subsections describe the phases of the proposed data transfer algorithm for PlutusDB. The algorithm itself is completed in three phases and is described in one direction, i.e. where a sub-cluster sends items to another receiving sub-cluster. However, in a scenario where a subset of data must be exchanged between sub-clusters, the algorithm should be executed in both directions (and the implementation might execute two “sending” and a “receiving” processes in parallel).

Another notable remark is that at this stage we do not specify any implementation detail, although we envision the Execute component performing the orchestration of the algorithm, i.e. reading and writing from the targeted source and destination sub-clusters and updating the placements details stored in the

Knowledge component.

The algorithm tries to transfer a subset of data items incrementally, to reduce downtime. This can be achieved through iteratively transferring data while serving and logging the operations performed on them, with the idea of applying these operations at the destination database in the subsequent round, until no operations target in-transfer items.

A – Static Phase The algorithm begins with the Execute component taking a snapshot of the status of the data that must be transferred (at the source database). The snapshot, when compared (in the next phases) with a future operations log, allows tracking which operations targeted in-transfer items during the transfer itself.

All the involved data items are then transferred (possibly with the help of known batching techniques [99] to optimize transfer time and resource usage), while the sending cluster keeps serving requests and logging them to a local operations log.

At the end of this phase, the destination database will store all the necessary data items, although some of them will be potentially not up to date, as some operations might have targeted them at the source database while the transfer was in progress. Thankfully, these operations can be easily reproduced by comparing the commit log at the end of the transfer with the snapshot that the source database took at the beginning of the Static Phase.

B – Dynamic Phase The second phase is an iterative procedure which goal is to progressively transfer the operations recorded during the first phase, trying to bring the destination database up to date.

Each round starts with comparing the current status of the items that have been transferred during the previous round, with the first round analysing the status of all the items transferred during the Static Phase. The sets of operations that targeted the in-transfer items, which we call "Deltas" are defined as the difference between the active operations log of the sending database at the start of the round and the operations log snapshot taken at the start of the previous round. In other words, a Delta consists of all the operations targeting items that were being transferred during the previous round.

While a Delta is transferred, the sending database will keep serving and recording requests on the log. Once the destination database has finished applying the delta, the active log is once again compared with the log at the start of the round and, in case some operations targeting in-transfer items were performed, another round must start. The Deltas will be transferred until the number of operations in the last Delta is $\varnothing$, declaring the end of the Dynamic Phase.

As proved in [56], the Deltas typically decrease in size, eventually becoming empty. In practice, when considering real deployment scenarios, a minimum threshold for the Deltas' size, (number of operations) could be set to guarantee the termination of the Dynamic Phase. The algorithm is guaranteed to terminate since every round will take less time (or, equivalently, contain less operations) than the previous round and will eventually be below the threshold, as stated in [56].

This phase opens up the possibility for further optimizations, such as dropping writes that have been overwritten by subsequent writes, reordering operations for faster transfer and others [99], as described in Chapter 6. These optimizations might be applied at this stage to decrease the size of Deltas, which directly impacts the time needed to apply them at the destination database, reducing the overall number of rounds.

C – Termination Phase At this point, both sending and receiving databases must briefly stop serving user requests, in order to allow the destination to apply the last Delta, ensuring at the same time that the items that no operations target any items that are part of the last Delta.

Due to the nature of the Dynamic Phase, the last Delta is guaranteed to be minimal in size, if not empty, meaning the time required to apply it will also be minimal, if not null. The last Delta's size directly impacts the downtime being imperceptible by the end user or even absent.

After the last Delta has been applied, the data placement information contained in the Knowledge component is updated, and the items that are part of the transfer set are deleted from the source database. The system now resumes serving client requests and the new configuration is effective: operations on the transferred items will be routed towards the destination database, instead.

One additional relevant benefit of the iterative approach is that the size of the last Delta is independent of the transfer set size, as stated in [56]. The minimum size threshold of the Delta is what ultimately defines the duration of the downtime, which, if present, is expected to be constant (estimated to be in the order of a few seconds).

4

PlutusDB's Optimizer

Contents

After describing PlutusDB as a whole, we now focus on the main contribution of this thesis: the Optimizer. This chapter begins with two introductory sections describing in detail the cost models of DBaaS and IaaS. Although the formulation is primarily defined for IaaS, we show the cost model of DBaaS since it is possible to extend the model to consider DBaaS as an additional storage location for the items. We then describe the Optimizer in detail: from the mathematical model to the implementation details.

4.1 Database as a Service on AWS

This section describes the AWS offering for DBaaS, which has been hinted at throughout the document: Amazon DynamoDB [26].

Although DBaaS is currently not competitive regarding cost per performance when compared to IaaS, it could still represent a possible placement for the items in case the cost characteristics will change in the future. For this purpose, the possibility of adding DBaaS is concrete and almost seemingly supported, as described in Section 4.4.

The cost model of a typical DBaaS offering is almost standard throughout all major vendors, albeit we focus on Amazon DynamoDB to have a simple and consistent model. Moreover, to present numerical and actual costs, we refer to the AWS Region: “Europe: Milan”.

We can represent DynamoDB's billing model as the sum of the following three main components:

- Cost per Read Request Unit (RRU) (0.2968 € per million RRU)
- Cost per Write Request Unit (WRU) (1.4842 € per million WRU)
- Cost of Storage (0.29715 € per GB per Month)

, where RRU is DynamoDB's internal billing unit. The amount of RRU billed per read depend on the size of the read item, in the following way: 0.5 RRU corresponds to reading 4KB. Therefore, reading a 16KB item requires using 2 RRU. WRUs follow a similar logic with a different ratio: 1 WRU corresponds to writing 1KB.

Finally, the Cost of Storage is self-explanatory, as it represents the baseline cost of storing data items: it depends on their size and is constant over time.

The cost of storing an item i in DBaaS is then modelled in Equation (4.1), where:

- s_i is the size of item i [KB].
- $iops_i^r$ and $iops_i^w$ are the, respectively, read and write, access ratios of item i [op/s].
- c^R, c^W and c^S are the aforementioned costs per million RRUs [€], per million WRUs [€] and of Storage [€/h].

- $\ulcorner n \urcorner$ is a custom function that rounds any number $n \in \mathbb{R}$ to the next multiple of 0.5, similarly to the classical ceiling $\lceil n \rceil$ function, but with half integer outputs, instead. For instance, $\ulcorner 7.7 \urcorner = 8$, $\ulcorner 8.00 \urcorner = 8$, and $\ulcorner 8.1 \urcorner = 8.5$

$$C = \ulcorner iops_i^r \cdot 3600 \cdot s_i/4 \urcorner \cdot \frac{c^R}{10^6} + \lceil iops_i^w \cdot 3600 \cdot s_i \rceil \cdot \frac{c^W}{10^6} + \frac{s_i}{10^6} \cdot \frac{c^S}{30 \cdot 24} \tag{4.1}$$

4.2 Infrastructure as a Service on AWS

This section describes Amazon's IaaS offering in detail, along with its cost model. In order to achieve concrete results from real-world scenarios, we decided to use AWS's IaaS offerings in early development stage as a reference for the cost models of a generic IaaS cluster.

4.2.1 IaaS cluster definition

Any IaaS cluster, at its most basic configuration, is a set of nodes. Each of them consists of a set of computational instances (called EC2, in the case of AWS), along with a set of volumes (Enterprise Block Storage (EBS)). Although volumes could be shared among multiple machines, DataStax's advice [25] is to use a shared-nothing architecture, which means that each VM should have its own attached volume that is not shared with other computing nodes. For clarity, we will call "node" the combination of a VM and its attached volume. We will also use "node type" to define a unique combination of VM type and volume type.

NoSQL Key-Value stores enforce replication, effectively storing multiple copies of each data item to improve performance and fault tolerance. Multiple replication strategies exist across commercially available and widely used NoSQL Key-Value stores. Therefore, we assume that whichever installed database will also enforce replication, and in particular, the ROWA replication strategy. This strategy, as described in section 2.4.1, consists of having multiple consistent copies of each data item. Every write operation is also performed at all secondary replicas, while read operations are performed only at one replica (not necessarily the Master Replica). The implementation of the Optimizer allows for an extension to support other replication strategies. However, adding such functionality requires changing the formulation of the problem, wholly or partially, depending on the complexity of the strategy one wants to implement.

4.2.2 Types of cluster nodes

Virtual Machines of different families (i.e., optimized for different types of workloads) were considered, once again referring to the offering of AWS. While other cloud providers offer different types of VMs with different nomenclatures, the same families of machines (optimized for similar workloads) are generally available. Furthermore, the set of machines can be changed to include other vendors' offerings without difficulty, as explained in Section 4.4.

All the machine types and their parameters chosen for the analysis are listed in Table 4.1. Henceforth, we will refer to "VMs" as the set of Instance Types listed in the first column of the table.

Table 4.1: Types of virtual machines considered in the analysis

Instance Type	**Cost €**	**Max Throughput (MB/s, 128 KiB I/O)**	**Max IOPS (16 KiB I/O)**
c4.2xlarge	0.398	125	8000
c4.4xlarge	0.796	250	16000
c4.8xlarge	1.591	500	32000
c4.large	0.1	62.5	4000
c4.xlarge	0.199	93.75	6000
m4.10xlarge	2	500	32000
m4.16xlarge	3.2	1250	65000
m4.2xlarge	0.4	125	8000
m4.4xlarge	0.8	250	16000
m4.large	0.1	56.25	3600
m4.xlarge	0.2	93.75	6000
r4.16xlarge	4.256	1750	75000
r4.2xlarge	0.532	212.5	12000
r4.4xlarge	1.064	437.5	18750
r4.8xlarge	2.128	875	37500
r4.large	0.133	53.13	3000
r4.xlarge	0.266	106.25	6000
x1.16xlarge	6.669	875	40000
x1.32xlarge	13.338	1750	80000

The choice of such machine types allows us to consider a wide range of workload-specific instance families, namely:

- **c4 family**: Compute optimized (optimized for compute-intensive workloads).
- **m4 family**: General purpose (provides a balance of computing power, memory, and network resources. Usually, it is a good choice for many applications).
- **r4 family**: Memory optimized (optimized for memory-intensive workloads).
- **x1 family**: Memory optimized (optimized for enterprise-class databases and in-memory applications).

While different families of VMs provide different combinations of thresholds for supported throughput and IOPS, and different cost ratios, there is usually no difference in the billing model between VMs, as VMs are only billed per time of usage (per hour), regardless of their family.

The other varying factor that characterizes node types is the type of volume the VMs are attached to, which introduce additional billing components. The volume types we considered are listed in Table 4.2, whereas their costs are listed in Table 4.3 for readability.

Table 4.2: Parameters of the volumes considered in the analysis

Volume Type	Max IOPS	Max Throughput (MB/s)	Max Storage (GB)
gp2	16000	250	4 000
gp3	16000	1000	4 000
io1	64000	1000	4 000
st1	500	500	4 000
sc1	250	250	4 000

Table 4.3: Costs of the volumes considered in the analysis

Volume Type	Cost of Storage (€/GB per MONTH)	Cost of IOPS (€/IOPS per MONTH)	Cost of Throughput (€/MB/s per MONTH)
gp2	0.1155	0	0
gp3	0.0924	0.0058	0.0462
io1	0.1449	0.0756	0
st1	0.525	0	0
sc1	0.01764	0	0

Once again, we tried to be as comprehensive as possible with the choice of volume types, as highly specialized volumes combined with similarly specialized VMs could be tailored solutions for applications with specific workload patterns. This possibility is enhanced by some volumes being optimized for high throughput (usually Solid State Drive (SSD)s), while others being optimized for colder workloads (usually Hard Disk Drive (HDD)s).

As previously mentioned, the billing of volumes is characterized by three components:

- Cost of storage: it is the cost of allocated (also called "provisioned" on AWS) storage space, which does not consider how much storage is effectively used.
- Cost of bandwidth usage.
- Cost of I/O Per Second (IOPS) usage (defined as a function of the bandwidth usage).

On AWS, the IOPS of an item are calculated by multiplying its bandwidth by a constant factor, that from now on we call σ. The concept of calculating the IOPS from the bandwidth is important since the IOPS that will be part of the cost formulation are not the "nominal" IOPS generated by the application, but rather the "billed" IOPS calculated by AWS. Certain volume types, such as "gp3" feature all three cost components, whereas others, such as "io1", do not.

More detailed information on AWS EBS pricing can be found on the official page [100].

As explained in section 2.2.2, we follow DataStax's recommendations, and we consider each instance's dedicated volume to have a fixed 4TB capacity, to reduce the scope of the problem. Therefore, provisioned storage costs will be the same per every instantiated node.

Adapting the Optimizer to adopt the newer cluster models is a minor modification, and the models can be trivially adapted to comply with the newest guidelines.

4.2.3 Cost model of a typical Database deployed on IaaS

Given that we focus on modeling clusters without additional components such as Load Balancers, confined to a single Availability Region, and with non-shared volumes, the cost model is identified by the sum of the cost components of instantiated VMs and volumes:

1. Cost of usage of the instantiated VMs.
2. Cost of volumes storage.
3. Cost of volumes bandwidth usage.
4. Cost of volumes IOPS usage (defined as a function of the bandwidth usage).

4.3 Optimizer

The Optimizer is the core part of this dissertation and its main contribution. It enables an eventual implementation of the rest of the system (proposed in Chapter 3) to later stages. The implementation of a complete system is left as a proposal for further work, hoping that other researchers in a near future would be eager to embrace the project and fully develop it to achieve the idea from which it was born.

As described in Section 3.5, the Optimizer, implementation of the Analyze+Plan component, aims at generating a data placement that minimizes the operational costs of the overall database. The Optimizer is designed to define a possibly hybrid cluster, and an according placement of the data items across multiple sub-clusters. In fact, there might be situations in which the access characteristics of a specific subset of the data items entail a type of node to be more cost-efficient when storing them. At the same time, another cluster made of a different node type might be more cost-efficient when storing the remaining items. In these situations, the Optimizer will instantiate the two sub-clusters and generate a data placement that reflects the expectations.

Naturally, we do not expect hybrid clusters to be the most cost efficient configuration for every type of workload. However, in these cases, the Optimizer could still be an extremely valuable tool, since the optimal configuration that the Optimizer produces will directly tell the user which is the best machine that suits the input workload, making the choice of the infrastructure straightforward. We could then say that the Optimizer is a tool that helps users choose the best IaaS cluster that minimizes the operational costs in all possible scenarios. Additionally, we might observe that a hybrid cluster could provide additional cost savings w.r.t. traditional cluster configurations.

This aspect enhances the importance and effectiveness of such tool, which can also be used independently from PlutusDB to help cloud architects to efficiently decide which infrastructure to deploy.

The optimal placement is achieved through the solving of an ILP problem, whose main decision variables represent the placement of data items. The Optimizer finds the optimal placement of multiple data items, each with a size and a nominal (application-generated) throughput. Formulating the problem in this way enables the grouping of an indefinite number of physical items, which can be represented as a single logical item whose size and throughput amounts to the sum of the individual items' sizes and throughputs. This approach allows the Optimizer to scale up and optimize more extensive databases, as the number of decision variables is the only real limiting factor, as we will show in Chapter 5. Naturally, the more items are grouped together, the lower the precision of the optimisation.

The way in which items are grouped directly impacts the precision, as well, although in this dissertation we do not analyze methods and implications of items' grouping. Overall, known methods can be applied, which can be application-level ("semantic") or based on the access characteristics (having higher throughput items grouped together). Data Placement methods have been studied in Section 2.7.

We initially modeled the ILP problem with a mathematical, formal formulation, which inputs are the up-to-date statistics of the stored data items. Afterward, we implemented the Optimizer as a program that can be run on demand, although in PlutusDB it will be run periodically. Finally, we evaluated it against numerous data sets.

4.3.1 Mathematical Formulation

The node types that can be instantiated are all possible combinations of VMs and volumes. Any given node type will inherit properties and costs from the VM and volume that compose it.

In a way, we are merging Tables 4.1, 4.2 and 4.3, so that every node -(a,b) pair- features:

- The lowest Max IOPS between the Max IOPS of VMtype a and the Max IOPS of volume type b.
- The lowest Max Throughput between the Max throughput of VMtype a and the Max Throughput of volume type b.
- The standard and constant maximum storage of 4TB.
- The cost per hour of VMtype a.
- The costs of Storage, IOPS and Throughput of volume type b.

The goal is to potentially have multiple database instances, each one associated with a sub-cluster, and each sub-cluster made of a single node type. Each item will be stored and replicated in one sub-cluster, which implies that each database instance must be deployed across a minimum of Replication Factor (RF) nodes. RF is then a key parameter in the choice of a cluster because of two reasons. Firstly, it impacts the number of required machines due to the need to store multiple copies of the items, and secondly because the actual read and write throughput of the items that will be billed are a function of the nominal throughputs and of RF, since RF copies of each item are accessed.

The mathematical formulation of the problem requires precise tracking of the data items. The placement of data items must be controlled in terms of

- In which sub-cluster items are stored
- In which node of the sub-cluster each item is stored

Controlling the placement of items in terms of sub-clusters is essential due to the need of ensuring an exact Replication Factor for each data item, since, as said before, we must ensure that the number of instances in which an item is stored is exactly RF.

Under these circumstances, the main decision variable (or, rather, set of decision variables), which we call $\mathbf{X}$, is modeled as a 3-dimensional matrix in which:

- The x-axis represents data items (indexed by a unique ID)
- The y-axis represents all the possible node types, which we identify with the set P: the Cartesian product of VM types and volume types.
  $P = \{(a, b) \mid a \in VM\_types, b \in volume\_types\}$.
- The z-axis represents all the possible instances that can be instantiated for a given node type.

As is usual in ILP problems whose goal is to identify an optimal data placement in a distributed context [101–103], the decision variables in the matrix are binary. One limitation of modeling the primary decision variable in the way mentioned above is that a maximum number of logical instances that can be allocated per type (which, in other words, is the dimension corresponding to the z-axis) must be fixed in advance. The reason being the matrix cannot be dynamically sized at runtime, given the static nature of optimization problems (the number of items and the dimension of the set P are already fixed). We then call K the maximum number of instances that might be allocated per type.

The value of K directly impacts the performance of the solver since increasing one dimension of a 3D matrix increases the total number of decision variables significantly. For this purpose, we optimize the formulation by introducing the concept of *logical* instances, which are groups of RF physical machines. We state that each sub-cluster can be made of $[0..K]$ *logical* instances. This approach effectively reduces the size of the matrix by RF times since we need not track each instantiated node singularly but rather groups of RF machines. On the other hand, this approach implies that the cluster can only be composed of multiples of RF machines, which we believe is an acceptable limitation that, on the other hand, significantly reduces the number of decision variables.

The meaning of each bit in the 3-dimensional placement matrix is explained in equation 4.2.

$$X_{ijk} = \begin{cases} 1 & \text{Item } i \text{ is stored in logical instance } k \text{ of type } j \\ 0 & \text{Otherwise} \end{cases} \tag{4.2}$$

Following, a series of constants that will be used in the mathematical formulation:

- $\vec{s}$: item sizes, s_i is the size of item i [MB]
- $\vec{t^r}$ and $\vec{t^w}$: item IOPS, t_i^r is the access ratio (reads/s) of item i , while t_i^w is the access ratio (writes/s) of item i [OPS/s]
- max_size is the size of each node's attached volume.
- RF is the replication factor, which is also the minimum number of nodes for the existence of any database instance (each sub-cluster must be made of at least RF machines)
- $\vec{iops}$ and $\vec{tp}$, defined $\forall j \in P$ are vectors that contain, respectively, maximum IOPS and maximum throughput of each node type.
- $\vec{cost}$ is also a vector that contains the costs per hour of each node type's VM.
- $\vec{cost}^{storage}$, $\vec{cost}^{iops}$ and $\vec{cost}^{tp}$ are vectors that contain, respectively, cost of storage, cost of IOPS and cost of throughput of each node type's volume.
- σ is a constant to transform the bandwidth in billed IOPS according to AWS's specifications (described in Section 4.2.3).

Calculating the cost per hour of the overall infrastructure requires counting the total number of instantiated nodes per type (since part of the infrastructure cost is the hourly usage of the VMs).

This number can be inferred through expression 4.3, where we compute a normalization of the total number of items stored in every logical instance k. In other words, the argument of the sum in Equation (4.3) means "Instance k of node j is instantiated at least once", and then, by summing all of the normalized values over k, we obtain the total number of instantiated machines of type j.

$$\text{\# of nodes of type } j = \sum_k \frac{|\sum_i \mathbf{X}_{ijk}|}{\sum_i \mathbf{X}_{ijk}} \tag{4.3}$$

Normalizing decision variables is non linear and therefore not permitted in ILP problems. Thus, expressing the sum above requires the introduction of a new utility binary matrix, $\mathbf{z}$.

The matrix $\mathbf{z}$ tracks whether each logical instance k of each node type j is instantiated or not (its value represents exactly the normalized sum of eq. (4.3), and is defined per every possible (j, k) pair).

The meaning of $\mathbf{z}$ is explained in Equation (4.4):

$$z_{jk} = \begin{cases} 1 & \text{Logical Instance } k \text{ of node type } j \text{ is instantiated} \\ 0 & \text{Otherwise} \end{cases} \tag{4.4}$$

Equation (4.5) helps visualizing $\mathbf{z}$. In eq. (4.5), logical instance $K-1$ of type "m4.large-gp2" is instantiated, as well as logical instance 0 of type "m4.large-sc1", whereas no instances of type "x1.32xlarge-sc1" are instantiated.

$$\mathbf{z} = \begin{array}{c|cccc|l} & \text{“m4.large-gp2”} & \text{“m4.large-sc1”} & \dots & \text{“x1.32xlarge-sc1”} & \\ \hline & 0 & 1 & \dots & 0 & \text{Logical Instance 0} \\ & \vdots & \vdots & \ddots & \vdots & \vdots \\ & 1 & 0 & \dots & 0 & \text{Logical Instance K-1} \end{array} \tag{4.5}$$

The correlation between $\mathbf{X}$ and $\mathbf{z}$ is defined for each node type $j \in P$ and for each logical instance $k < K$ as in equations 4.6a and 4.6b:

$$\sum_{i=0}^{N-1} \mathbf{X}_{ijk} = 0 \iff \mathbf{z}_{jk} = 0 \tag{4.6a}$$

$$\sum_{i=0}^{N-1} \mathbf{X}_{ijk} > 0 \iff \mathbf{z}_{jk} = 1 \tag{4.6b}$$

With the definition of $\mathbf{z}$, The number of instantiated logical nodes per type can now be expressed in a simple way, shown in Equation (4.7).

$$m_j = \sum_{k=0}^{K-1} z_{jk} \tag{4.7}$$

Finally, the cost formulation (objective function of the optimization problem) is shown in equations 4.8, while the constraints are shown in equations 4.9. The decision variables are highlighted in red for better readability.

$$C = \sum_{j \in P} \sum_{k=0}^{K-1} \quad \vec{z}_{jk} \cdot RF \cdot cost_j + \tag{4.8a}$$

$$\sum_{j \in P} \sum_{k=0}^{K-1} \quad \vec{z}_{jk} \cdot RF \cdot max\_size \cdot cost_j^{storage} + \tag{4.8b}$$

$$\sum_{i=0}^{N} \sum_{j \in P} \sum_{k=0}^{K-1} \quad X_{ijk} \cdot (t_i^r + t_i^w \cdot RF) \cdot s_i \cdot cost_j^{tp} + \tag{4.8c}$$

$$\sum_{i=0}^{N} \sum_{j \in P} \sum_{k=0}^{K-1} \quad X_{ijk} \cdot (t_i^r + t_i^w \cdot RF) \cdot s_i \cdot \sigma \cdot cost_j^{iops} \tag{4.8d}$$

$$\min_{\mathbf{X},\vec{z}} \quad C \tag{4.9a}$$

subject to:

$$\sum_{i=0}^{N-1}(X_{ijk}\cdot s_i) \le max\_size \qquad \forall j\in P,\ \forall k<K \tag{4.9b}$$

$$\sum_{i=0}^{N-1}(X_{ijk}\cdot(t_i^r + t_i^w\cdot RF)\cdot s_i) \le tp_j\cdot RF \qquad \forall j\in P,\ \forall k<K \tag{4.9c}$$

$$\sum_{i=0}^{N-1}(X_{ijk}\cdot(t_i^r + t_i^w\cdot RF)\cdot s_i)\cdot\sigma \le iops_j\cdot RF \qquad \forall j\in P,\ \forall k<K \tag{4.9d}$$

$$\sum_{i=0}^{N-1} X_{ijk} \le M\cdot z_{jk} \qquad \forall j\in P,\ \forall k<K \tag{4.9e}$$

$$\sum_{i=0}^{N-1} X_{ijk}\cdot M \ge z_{jk} \qquad \forall j\in P,\ \forall k<K \tag{4.9f}$$

$$\sum_{j\in P}\sum_{k=0}^{K-1} X_{ijk} = 1 \qquad \forall i<N \tag{4.9g}$$

The objective function features the following subequations:

- 4.8a models the cost per hour of instantiated nodes. We must instantiate RF physical nodes per logical node.
- 4.8b models the cost of the allocated storage, with each physical node having an attached volume with capacity *max_size*. Remark: AWS bills the provisioned storage, regardless of how much of it is actually used.
- 4.8c models the cost of throughput usage.
- 4.8d models the cost of IOPS usage as a function of the throughput.

The meaning of each constraint is the following:

- 4.9b ensures that each logical instance of each node type must store at most max_size GB.
- 4.9c ensures that each logical instance of each node type must store items whose generated throughput amounts to at most tp_j, which is the maximum bandwidth provided by node type j.
- Similarly, 4.9d ensures that each logical instance of each node type must store items whose iops amount to at most $iops_j$, the maximum IOPS provided by node type j.
- 4.9e and 4.9f are used to establishing the correlation between decision variables, expressed in inequations 4.6a and 4.6b. A common way [104] to formulate "if-then" constraints requires a large

auxiliary number, M, orders of magnitude larger than any other coefficient in the problem (10^5 or more). It can be proven that these constraints are effective by reductio ad absurdum: since we want to force that if $\sum_{i=0}^{N-1} X_{ijk} = 0$, then $z_{jk} = 0$ and if $\sum_{i=0}^{N-1} X_{ijk} > 0$, then $z_{jk} > 0$, we first ensure that both constraints 4.6a and 4.6b are satisfied in those two cases. Afterwards, we analyze the cases in which the constraints should NOT be satisfied, in particular:

- Suppose that $\sum_{i=0}^{N-1} X_{ijk}{=}0$ and $z_{jk}{>}0$. This results in constraint 4.6b to be violated.
- Similarly, if $\sum_{i=0}^{N-1} X_{ijk}{>}0$ and $z_{jk}{=}0$, instead, violates constraint 4.6a.

Whereas both of the cases that should be allowed do not violate any of the two constraints.

- 4.9g ensures that each item is placed in only one logical instance, and therefore replicated across its RF physical instances.

Note that the data presented in Table 4.1, Table 4.2 and Table 4.3 is shown precisely as displayed on the official AWS website [100]. In order to correctly calculate the cost per hour of the solution, we must ensure every cost is homogenized according to the time of reference, e.g. converted from "per MONTH" to "per HOUR", and converted to the same measurement unit, as well (e.g. MiB, GB, etc.). We found the best choice given the problem size was to transform everything to GB. These conversions are omitted for readability in the mathematical formulation, although they are implemented directly in the Python code, publicly available on GitHub [105].

After defining the optimization problem, we must be able to compare the solver's solution (which, in some cases, might be hybrid) with the best homogeneous solution (an IaaS cluster with a single database instance of nodes of the same type), to calculate the cost savings, if any. By doing so, we can achieve one of the goals of this thesis: prove that there exist some data sets in which a hybrid placement is advantageous from a cost perspective with respect to a standard IaaS solution.

To do this, it is possible to reuse the solver with an additional constraint that forces the number of allocated node types to be 1. We then introduce a new array of binary decision variables, which we call $\mathbf{y}$, which represents whether any node type is instantiated at least once:

$$y_j = \begin{cases} 1 & \text{Node type } j \text{ is instantiated at least once} \\ 0 & \text{Otherwise} \end{cases} \tag{4.10}$$

Whether node type j is instantiated at least once can be inferred by evaluating the sum over k of the allocated logical instances for a given node type j: $\sum_k z_{jk} > 0$, and therefore by defining a correlation between this sum and y_j in a similar way as we did to generate $\mathbf{z}$ from $\mathbf{X}$ in constraints 4.6a and 4.6b. Statements 4.11a and 4.11b show the correlation in a formal mathematical formulation.

$$\sum_k \mathbf{z}_{jk} = 0 \iff \mathbf{y}_j = 0 \tag{4.11a}$$

$$\sum_k \mathbf{z}_{jk} > 0 \iff \mathbf{y}_j = 1 \tag{4.11b}$$

The constraints that must be added to the problem make once again use of the variable M and are expressed in constraints 4.12a and 4.12b.

$$\sum_k z_{jk} \leq M \cdot y_j \tag{4.12a}$$

$$\sum_k z_{jk} \cdot M \geq y_j \tag{4.12b}$$

Finally, we can insert the final constraint that guides the solver towards choosing only one node type, as shown in Equation (4.13).

$$\sum_{j \in P} y_j = 1 \tag{4.13}$$

This completes the ILP problem, that can now be formulated with any language (AMPL or any solver's APIs for programming languages) and implemented in any framework or tool (Gurobi, CPLEX, etc.) that one desires to use.

4.3.2 Software Choice

The mathematical model must be implemented in some programming language to be able to create a tool that allows an existing solver to process it and to parsing and manipulating the results in a meaningful way afterward. A choice has been made to use Python[1] due to its flexibility, since it allows easy retrieval of data from excel sheets and, subsequently, dictionaries manipulation through simple APIs and without too much overhead.

A second reason for choosing Python lies in the possibility of writing very high-level and understandable code. This will be useful in the future, as other researchers will need to use and modify the tool to complete PlutusDB. It is also possible to create numerous expressions simultaneously, just like the mathematical formulation does when specifying a constraint $\forall j \in P$, for instance. It is a useful functionality when defining sets of constraints in large problem formulations.

Finally, what ultimately directed our choice through Python, is the possibility to use Gurobi[2] [106,107].

As described in Section 2.8.3, Gurobi is a robust solver that, thanks to the academic license offering, and in combination with Python, allows for highly expressive problem modelling and excellent solving performance without incurring in any costs.

[1]https://www.python.org
[2]https://www.gurobi.com

The Optimizer has been made publicly available on GitHub as a fully open-source project [105].

4.4 Support for DBaaS and other cloud vendors

After the model finalization, we must point out that it is possible for the Optimizer to support DBaaS as an additional option for the data placement by modeling DBaaS as an additional node type, with no constraints on throughput, IOPS or size.

In order to introduce DBaaS, and given that its billing is generally per-access (see Section 4.1), we must make the following modifications to the formulation:

1. Redefine variables $\mathbf{X}$ and $\mathbf{z}$ to be defined over an extended version of the P set, which we call P_D, which is essentially $P \cup \{\text{“}\textit{DBaaS}\text{”}\}$.

2. Add a constraint that implies that only one “logical instance” of DBaaS can be instantiated (since there are no multiple machines among which to choose for this specific case).

$$\sum_{k=0}^{K-1} X_{ijk} = 1 \quad \forall i < N,\ j = \text{“}\textit{DBaaS}\text{”}$$

3. Add two expressions to the objective function that are only valid for $j = \text{“}\textit{DBaaS}\text{”}$, which represent the costs of storing and accessing item stored in DBaaS. We use $cost_D^S$ to identify the cost of storing an item in DBaaS. We use $cost_D^W$ and $cost_D^R$ to identify, respectively, the costs of writing and reading a data item stored in DBaaS, as described in Section 4.1.

$$\sum_{i=0}^{N-1} \sum_{k=0}^{K-1} X_{ijk} \cdot s_i \cdot cost_D^S$$

$$\sum_{i=0}^{N-1} \sum_{k=0}^{K-1} X_{ijk} \cdot (t_i^r \cdot cost_D^R + t_i^w \cdot cost_D^W)$$

Note that no RF is involved in these equations because DBaaS usually handles replication internally, as it is the case in Amazon DynamoDB (see Section 2.4.1). Therefore, the cost of storing and accessing multiple replicas is already reflected in the costs per access and storage.

This variant is not specified in the main mathematical problem and is not implemented in the current version of the Optimizer (although trivial to insert) since we preferred to concentrate the efforts on a pure IaaS solution.

However, this is a concrete possibility for cloud architects that would like to explore whether there would be any cost advantage in employing DBaaS as cloud storage.

Integrating this option with PlutusDB would result in changes in other components, too:

- The Knowledge component must be extended to support DBaaS as an additional option for the data placement
- The Execute component must be extensively changed to enable performing data migrations between DBaaS and IaaS, and vice-versa.
- The data transfers within PlutusDB would incur additional costs, as reading and writing data from DBaaS must be added to the cost of achieving a new configuration.

Therefore, we believe integrating DBaaS as an additional supported backend storage for PlutusDB could be another project on its own.

5 Evaluation and Data Collection

Contents

This chapter presents the results collected from various experiments, where we used the Optimizer to find the best IaaS configuration of numerous data sets. The chapter concludes with some comments on the tool's computational requirements and scalability.

5.1 Data collection

The next step of the dissertation consists of gathering experimental data to demonstrate the purpose of the solution in real-world scenarios. Several approaches for creating sample workloads have been used extensively to test the load and effectiveness of widely used commercial solutions. YCSB was used as a reference for the workload characterizations since we believed it was the better tool for benchmarking NoSQL databases. YCSB contains six core workloads, and we selected three of them to test the Optimizer and collect data. The three workloads that we considered are:

- **Workload A**: 50% Reads - 50% Writes
- **Workload B**: 95% Reads - 5% Writes
- **Workload C**: 100% Reads

Since YCSB normally generates vast amounts of individual data items, we must operate with a coarser granularity. As discussed in Chapter 4, this can be done by logically mapping n physical data items, each of them with size s, throughput tp and IOPS io to a larger logical data item with size $s \cdot n$, throughput $tp \cdot n$ and IOPS $io \cdot n$. The reason behind the use of logical items is scalability, which will become clearer later in the Chapter.

We then take inspiration from the three selected workloads to generate data that is suited for this specific study and that can be fed into the Optimizer. We formally approach data collection: we model a set of experiments with several parameters and multiple levels each. The goal is to explore a vast space of different data sets and eventually find scenarios that yield hybrid clusters as the optimal (cheapest) solution.

The varying parameters for this experiment are:

- Size of individual items (constant across all data items)
- Total throughput (Throughput, and therefore IOPS, of individual items vary according to a zipfian distribution, as it is the case in YCSB)
- Reads %, which further characterizes the access pattern, as reads and writes have different impacts on the infrastructure constraints and on costs.

Note that the total throughput is not a parameter present in YCSB but rather a parameter specific to this study. Other parameters, which value has been fixed in order to reduce the scope of the space search, are:

- skew of the distribution, which we set to 1,
- Replication Factor, which we set to 3 given the best practices presented in section 2.2.2,
- N (the total number of logical items), which we set to 300.

The levels for each varying parameter were selected empirically, with the aim of, once again, covering potentially real scenarios:

- Items' size: 0.01GB to 10GB. Every step multiplies the size of the items by 10.
- Total Throughput: 0.001 GB/s to 10 GB/s. Every step multiplies the total throughput by 10.
- Reads %: according to YCSB's Workload A (50%), Workload B (95%) and Workload C (100%).

To better visualize which combinations of parameters yield hybrid clusters as the best IaaS solution, we visualize the cost saving of all the executed experiments through the heat maps presented in Figures 5.1 to 5.3. Note that the cost saving amounts to zero in case the solution is not hybrid. Each experiment has been executed with 300 logical items.

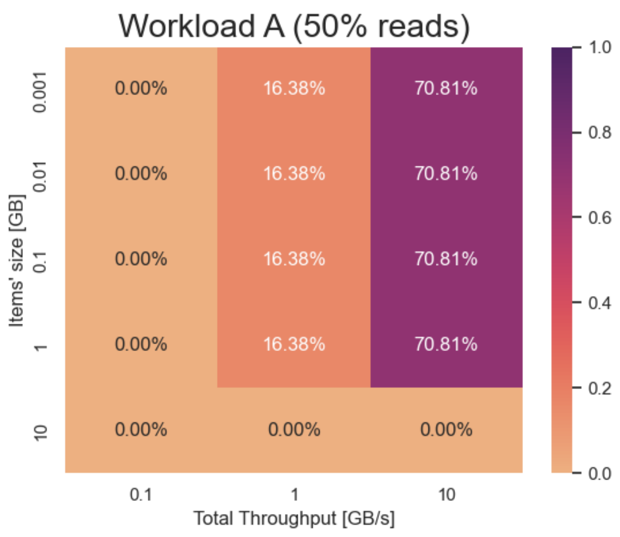


Figure 5.1: Cost savings for different throughputs and items' sizes, 50% reads - 50% writes

These results suggest that data sets with higher total throughputs are more suited for hybrid clusters, as the overall throughput variability among items is more significant when the popularity is modeled according to a zipfian distribution. We can also infer that the size of individual items does not affect the cost savings, probably due to the experiment modeling with each item having the same size.

Let us now look at the hybrid cluster configurations that result in cost savings. We start from the analysis of the Workload A experiment with 300 logical items, each of them with a 0.1 GB size, and a total throughput of 1 GB/s. We will call this "**Experiment 1**". The cost-optimal multi-node-type configuration is displayed in the first two lines of Table 5.1 and Table 5.2, whereas the homogeneous configuration

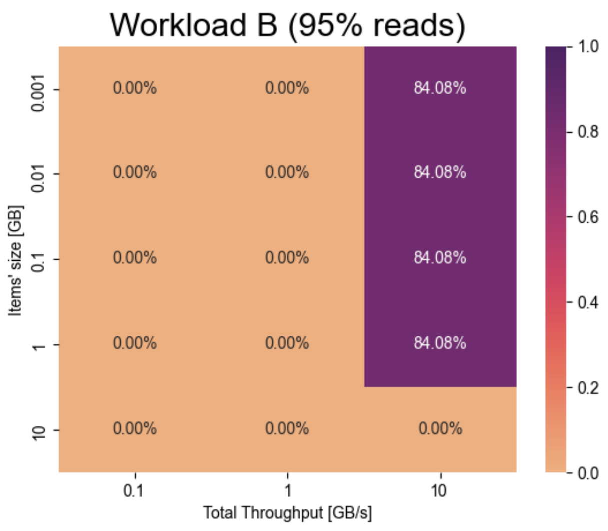


Figure 5.2: Cost savings for different throughputs and items' sizes, 95% reads - 5% writes

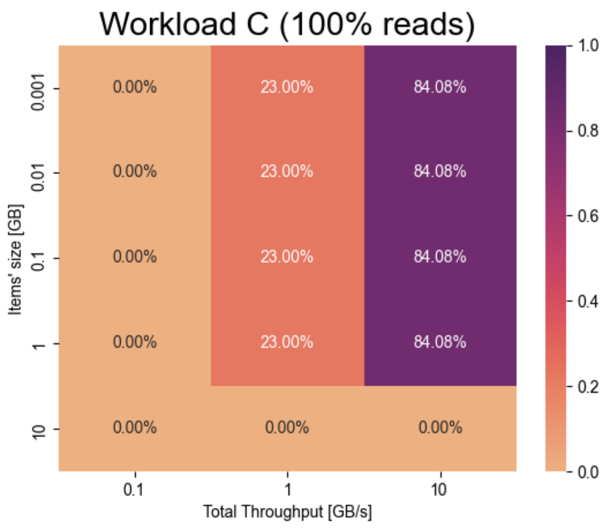


Figure 5.3: Cost savings for different throughputs and items' sizes, 100% reads

corresponds to the last line in both tables. This particular experiment yields a cost saving of 16.38% with respect to the best single-node-type cluster configuration (18 instances of "c4.xlarge-gp2" nodes). For better readability, we divide the data into two tables. The values of **Total Cost per hour**, **Max Throughput**, and **Max IOPS** are obtained by multiplying the parameters of the corresponding node (in Table 4.1) by the number of allocated instances.

The second experiment (which we call "**Experiment 2**") that we are going to analyze is from Workload B (95% reads), with items sized 0.1GB and a total throughput of 10GB/s. Tables 5.3 and 5.4 represent the allocated nodes in Experiment 2, which we also divide into two tables for readability. The first lines

Table 5.1: Used nodes in "**Experiment 1**", part 1

Node Type	Allocated Instances	Total Cost per hour (€)	Max Bandwidth MB/s	Max IOPS ops/s
c4.large-gp2	18	1.8	1 125	72 000
c4.xlarge-gp2	6	1.194	562.5	36 000
c4.xlarge-gp2	18	3.582	1 687.5	108 000

Table 5.2: Used nodes in "**Experiment 1**", part 2

Node Type	Allocated Instances	Bandwidth Saturation	IOPS Saturation
c4.large-gp2	18	99.67 %	95.06 %
c4.xlarge-gp2	6	99.08%	94.49 %
c4.xlarge-gp2	18	99.47%	94.87 %

are the nodes allocated for the hybrid cluster, whereas the last line refers to the homogeneous cluster configuration for the same experiment. The hybrid cluster resulting from Experiment 2 results in a cost

Table 5.3: Used nodes in "**Experiment 2**", part 1

Node Type	Allocated Instances	Total Cost per hour (€)	Max Bandwidth MB/s	Max IOPS ops/s
c4.2xlarge-gp2	3	1.194	375	24 000
c4.4xlarge-gp2	3	2.388	750	48 000
c4.8xlarge-io1	3	4.773	1500	96 000
c4.large-gp2	78	0.78	4 875	312 000
c4.xlarge-gp2	9	1.791	843.75	54 000
m4.large-gp2	3	0.3	168.75	10 800
r4.2xlarge-gp2	3	1.596	637.5	36 000
r4.xlarge-gp2	3	0.798	318.75	18 000
m4.16xlarge-io1	9	28.8	9000	576 000

saving of 84.08%.

To better understand the dynamics that lead the Optimizer to choose the optimal configuration, we analyze the saturation of the three quantities that constrain the choice of nodes: IOPS, bandwidth and storage. The saturation of experiments 1 and 3, for both hybrid and non-hybrid best configuration is presented in Figures 5.4 to 5.6.

The saturation of the hybrid clusters are obtained as an average of each individual sub-cluster's saturation, weighted by the number of instances allocated per sub-cluster (respectively, columns "Bandwidth Saturation" and "Allocated Instances" of Tables 5.2 and 5.4). This way, we can get a fair estimate of the average saturation of each sub-cluster.

It is evident that in both hybrid and non-hybrid clusters, the solver aims at saturating storage, IOPS and bandwidth as much as possible without violating feasibility. **Experiment 1** shows the potential of hybrid clusters, as, despite the maximum supported bandwidth and IOPS being the same for both hybrid and non-hybrid configurations, the hybrid configuration's cost per hour is lower, making it a more cost-

Table 5.4: Used nodes in "**Experiment 2**", part 2

Node Type	Allocated Instances	Bandwidth Saturation	IOPS Saturation
c4.2xlarge-gp2	3	99.86%	95.23%
c4.4xlarge-gp2	3	97.97%	93.43%
c4.8xlarge-io1	3	97.97%	93.43%
c4.large-gp2	78	98.18%	93.64%
c4.xlarge-gp2	9	97.65 %	93.12 %
m4.large-gp2	3	95.59 %	91.16 %
r4.2xlarge-gp2	3	92.25 %	99.71 %
r4.xlarge-gp2	3	92.20 %	99.66 %
m4.16xlarge-io1	9	82.07%	97.83%

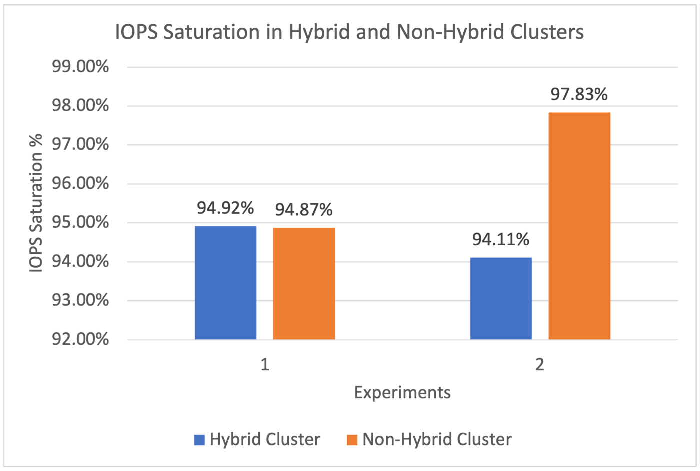


Figure 5.4: IOPS saturation comparison of hybrid and non-hybrid clusters in experiments 1 and 2

efficient choice than a traditional homogeneous cluster.

Experiment 2, on the other hand, shows how a hybrid cluster can be more cost-efficient due to better saturation of the bottleneck parameter, in this case: bandwidth. In fact, as it is evident in Figure 5.5, the bandwidth saturation related to Experiment 2 is higher for the hybrid cluster, resulting in the hybrid cluster being a better fit for the storage of such data set.

Moreover, note how the storage saturation is dramatically lower than the saturation of the other two quantities. This is mainly due to these scenarios being high throughput, hence IOPS and bandwidth saturating first, making it impossible to store additional items in the saturated nodes. Additionally, the best practice of using 4TB per node published in DataStax's whitepaper (see Section 2.2.2) leaves wide margins for storing data even in low throughput scenarios, which implies saturating storage space to be unlikely and only seen in rare cases with large amounts of data with extremely low access ratios. Although we are aware of the fact that databases might need to store additional data, and therefore it is good practice to maintain some unused space, we believe that 4TB per machine is an upper limit which, given the extremely low measured saturation, could be reduced when facing high throughput data sets,

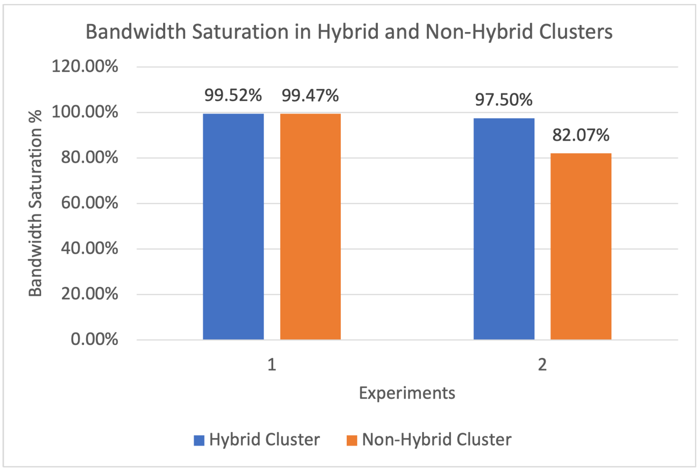


Figure 5.5: Bandwidth saturation comparison of hybrid and non-hybrid clusters in experiments 1 and 2

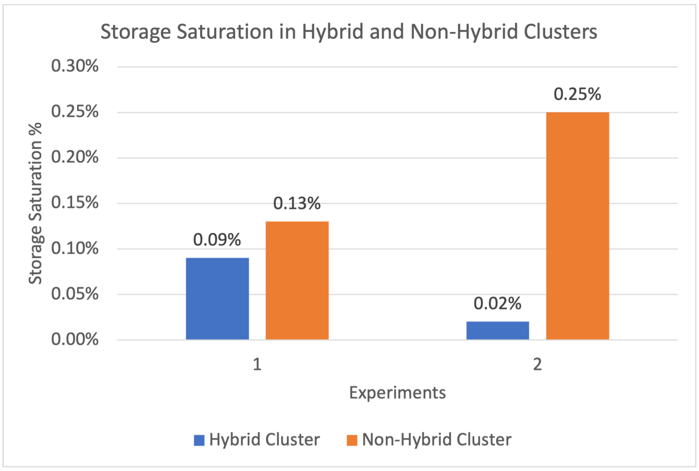


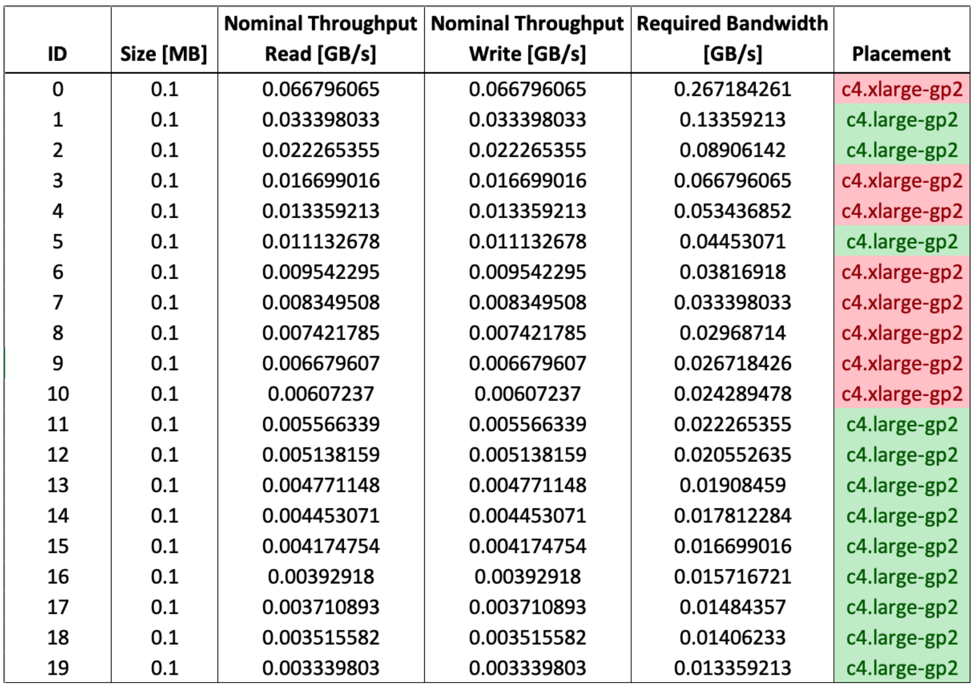
Figure 5.6: Storage saturation comparison of hybrid and non-hybrid clusters in experiments 1 and 2

in order to decrease the overall billing.

Cost savings, then, hardly originate from better saturation of the cluster's capabilities, but rather from a more cost-efficient composition of the clusters, which exploit the access characteristics of the items (their generated bandwidth and IOPS).

To demonstrate this further, we analyze the placement of the items in the experiments that yielded hybrid placements. As it is not feasible in terms of space to present the placement of all the logical items, we show the placement of only the first items, ordered by popularity (number of accesses per second).

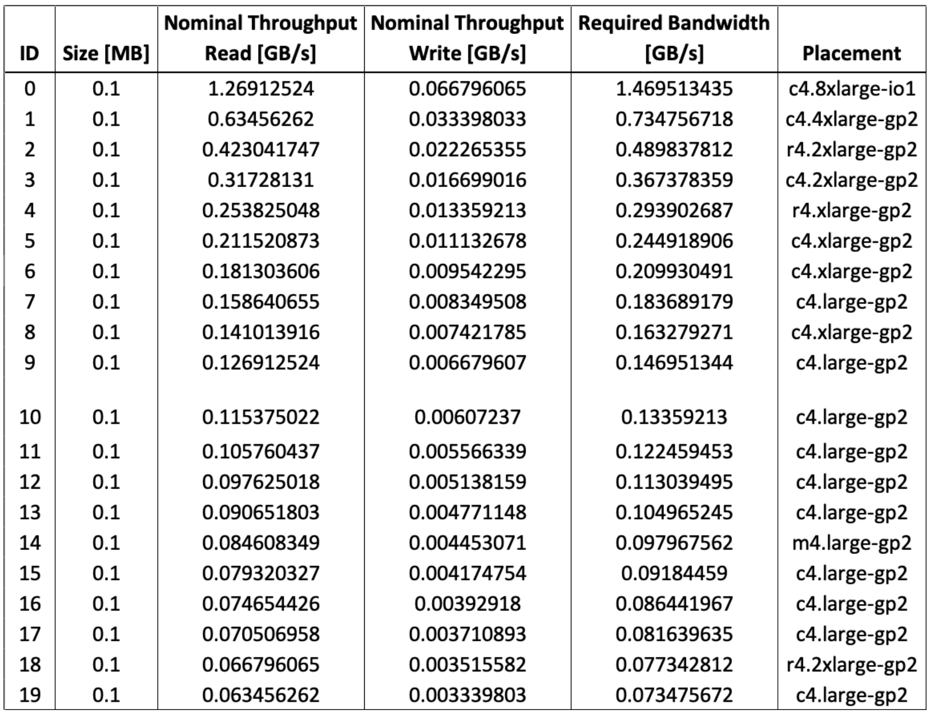
Figure 5.7 shows the items' placement of the 20 most popular items in Experiment 1. The two node types that form the hybrid cluster are highlighted in different colors for clarity. The remaining items that are not displayed are mostly placed in the "c4.large-gp2" node type (in green), with just a few exceptions. In this case, the solver is allocating the most popular items in the more capable machines (in red) and the colder items in cheaper and less capable machines (in green). However, there are some "outliers" (i.e. popular items that are stored in cheap machines and some cold items, not displayed, that are stored in capable machines). A possible explanation for this is that the solver tries to optimize the placement to reach the saturation of one of the three parameters mentioned above, to optimize the cost-efficiency pf the placement. The outliers are then used to fill the unused IOPS/capacity/bandwidth and exploit each machine fully. This behavior is in line with the saturation percentages seen before.



ID	Size [MB]	Nominal Throughput Read [GB/s]	Nominal Throughput Write [GB/s]	Required Bandwidth [GB/s]	Placement
0	0.1	0.066796065	0.066796065	0.267184261	c4.xlarge-gp2
1	0.1	0.033398033	0.033398033	0.13359213	c4.large-gp2
2	0.1	0.022265355	0.022265355	0.08906142	c4.large-gp2
3	0.1	0.016699016	0.016699016	0.066796065	c4.xlarge-gp2
4	0.1	0.013359213	0.013359213	0.053436852	c4.xlarge-gp2
5	0.1	0.011132678	0.011132678	0.04453071	c4.large-gp2
6	0.1	0.009542295	0.009542295	0.03816918	c4.xlarge-gp2
7	0.1	0.008349508	0.008349508	0.033398033	c4.xlarge-gp2
8	0.1	0.007421785	0.007421785	0.02968714	c4.xlarge-gp2
9	0.1	0.006679607	0.006679607	0.026718426	c4.xlarge-gp2
10	0.1	0.00607237	0.00607237	0.024289478	c4.xlarge-gp2
11	0.1	0.005566339	0.005566339	0.022265355	c4.large-gp2
12	0.1	0.005138159	0.005138159	0.020552635	c4.large-gp2
13	0.1	0.004771148	0.004771148	0.01908459	c4.large-gp2
14	0.1	0.004453071	0.004453071	0.017812284	c4.large-gp2
15	0.1	0.004174754	0.004174754	0.016699016	c4.large-gp2
16	0.1	0.00392918	0.00392918	0.015716721	c4.large-gp2
17	0.1	0.003710893	0.003710893	0.01484357	c4.large-gp2
18	0.1	0.003515582	0.003515582	0.01406233	c4.large-gp2
19	0.1	0.003339803	0.003339803	0.013359213	c4.large-gp2

Figure 5.7: Placement of the 20 most popular items, Experiment 1

Figure 5.8 shows the items' placement of the 20 most popular items in Experiment 2. Although the hybrid cluster is made of more node types with respect to the previous experiment, we can see how the Optimizer allocates items in the same way: the more popular items are stored in more capable machines, whereas the allocated machine type becomes less expensive and capable as the items decrease in popularity. Some non-displayed outliers, that are less popular than the shown items, are once again possibly used to fill the unexploited capabilities of the allocated machines, to achieve the highest possible saturation of either size, IOPS, or bandwidth.



ID	Size [MB]	Nominal Throughput Read [GB/s]	Nominal Throughput Write [GB/s]	Required Bandwidth [GB/s]	Placement
0	0.1	1.26912524	0.066796065	1.469513435	c4.8xlarge-io1
1	0.1	0.63456262	0.033398033	0.734756718	c4.4xlarge-gp2
2	0.1	0.423041747	0.022265355	0.489837812	r4.2xlarge-gp2
3	0.1	0.31728131	0.016699016	0.367378359	c4.2xlarge-gp2
4	0.1	0.253825048	0.013359213	0.293902687	r4.xlarge-gp2
5	0.1	0.211520873	0.011132678	0.244918906	c4.xlarge-gp2
6	0.1	0.181303606	0.009542295	0.209930491	c4.xlarge-gp2
7	0.1	0.158640655	0.008349508	0.183689179	c4.large-gp2
8	0.1	0.141013916	0.007421785	0.163279271	c4.xlarge-gp2
9	0.1	0.126912524	0.006679607	0.146951344	c4.large-gp2
10	0.1	0.115375022	0.00607237	0.13359213	c4.large-gp2
11	0.1	0.105760437	0.005566339	0.122459453	c4.large-gp2
12	0.1	0.097625018	0.005138159	0.113039495	c4.large-gp2
13	0.1	0.090651803	0.004771148	0.104965245	c4.large-gp2
14	0.1	0.084608349	0.004453071	0.097967562	m4.large-gp2
15	0.1	0.079320327	0.004174754	0.09184459	c4.large-gp2
16	0.1	0.074654426	0.00392918	0.086441967	c4.large-gp2
17	0.1	0.070506958	0.003710893	0.081639635	c4.large-gp2
18	0.1	0.066796065	0.003515582	0.077342812	r4.2xlarge-gp2
19	0.1	0.063456262	0.003339803	0.073475672	c4.large-gp2

Figure 5.8: Placement of the 20 most popular items, Experiment 2

5.2 Resource utilization and Scalability

Resource utilization and performance of solvers are naturally bound by the number of decision variables of the problem. The total number of decision variables can be expressed as: the cardinality of $\mathbf{X}$ $(N \cdot |P| \cdot K)$ plus the cardinality of $\mathbf{z}$ $(|P| \cdot K)$, which does not depend on N. The expression can then be simplified to $(N+1) \cdot |P| \cdot K$. In the specific case of this analysis and with these considered node types, given that the number of instantiable nodes $|P|$ is 76 and that we fix a maximum number of logical instances (K) to 20, the total number of decision variables depends directly on N. When using $N = 500$ logical data items, for instance, the total number of variables is 761 520.

We performed all the following experiments with the "NodefileStart" setting of the solver to a value of 4GB. Setting this value is necessary, otherwise the solver would require overwhelming amounts of RAM, which cause the program to crash indefinitely. As described in Section 2.8.3, this parameter limits the maximum RAM usage and forces the solver to write information on files, although still counting this as RAM usage.

We will now show data collected from multiple studies, each of them with an increasing number of items. For each study, we will include two experiments: one with low total throughput and one with high total throughput, to show the impact that high throughput data sets have on optimization times.

Figure 5.9 shows the runtime of the Optimizer for increasing numbers of logical items. All the data points have been collected by performing experiments with the corresponding N, a total throughput of either 0.001GB/s (blue line) or 1 GB/s (orange line), a constant items' size of 0.1 GB, and the read/write ratio of Workload A (50% Reads, 50% Writes).

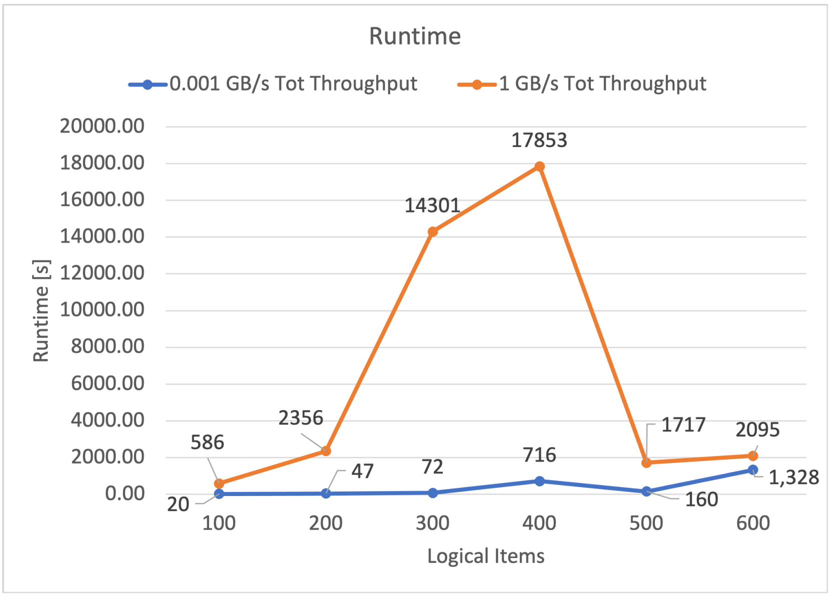


Figure 5.9: Optimizer processing time for high and low throughput datasets, increasing numbers of logical items

In both high throughput and low throughput scenarios, the first four experiments until N=400 yielded a hybrid placement, while the last ones, with 500 and 600 items, did not. This explains a runtime decrease with N=500, as the problem is easier to solve. The runtime of the experiment with 600 items is slightly higher than the one with 500 items, since, although both experiments yielded a homogeneous cluster, the number of variables is higher.

Higher throughputs result in longer total runtimes for the problem to be solved, as the complexity is higher with higher coefficients. Moreover, the runtime increase is steeper in high throughput data sets. Despite this, the increasing runtime trend with respect to the total number of logical items is somehow comparable between the two datasets since, in both cases, it features an increase that is correlated to the increase of logical items and a drop once the experiments do not yield hybrid placements anymore. This study suggests that controlling the number of items is critical to achieve lower runtimes.

The downside of grouping more physical items into fewer logical items is that, given that each logical item must be stored in a single logical node, the constraints on size, throughput and IOPS that a single logical node can sustain effectively limit the scenarios that the solver could operate on.

For instance, suppose that the stricter constraint for a specific cluster is on bandwidth. The upper limit on the analyzable scenarios is represented by the logical item that generates the highest bandwidth. This item must be stored in precisely one logical instance to guarantee feasibility. Therefore, it must generate a bandwidth that is at most equal to the max bandwidth of the most capable node type. Otherwise, when its generated bandwidth is higher than the supported bandwidth of the most capable logical instance (RF times the bandwidth of the most capable physical node), there would be no configuration of nodes that could store the item without violating feasibility.

To formally define a threshold for the usability of a data set for optimization, we define:

- $\hat{t}$ as the highest throughput among all logical items,
- $\hat{r}$ as the percentage of reads of the item with the highest throughput,
- $\hat{io}$ as the highest IOPS among all logical items (although this value could be calculated by multiplying the maximum throughput by σ, as described in section 4.2.2),
- $\hat{s}$ as the largest size among all logical items,
- t^{max}, io^{max}, s^{max} as the maximum supported throughput, IOPS and size among all available nodes.

All three inequalities expressed in 5.1 must be respected to guarantee feasibility.

$$\hat{t} \cdot \hat{r} + \hat{t} \cdot (1 - \hat{r}) \cdot RF \leq t^{max} \tag{5.1a}$$

$$\hat{io} \leq io^{max} \tag{5.1b}$$

$$\hat{s} \leq s^{max} \tag{5.1c}$$

In case either of the three constraints is violated, programmers are forced to use a finer granularity when grouping physical data items to logical ones. By doing so, the the value of the parameter(s) that cause the constraint(s) to be violated will decrease and the largest item could be stored into the most capable machine. Nevertheless, this operation inevitably increases the hardware requirements and the optimization time, as the number of logical items will be lager.

Next, we analyze the peak resident set size (the total physical RAM used, also called "High Water Mark") of the Optimizer's process for increasing values of N, to understand the Optimizer's hardware requirements. This data has been gathered by reading the value of "VmHWM" in the `/proc/{PID}/status` file (where PID is the Optimizer's Process ID), which represents the peak physical memory usage of the process. After gathering data from multiple runs with increasing values of N, we present the data in Figure 5.10. Once again, we distinguish experiments with increasing N between high throughput and low throughput to have a more complete view of the trends. We perform experiments with constant items' sizes of 0.1 GB and balanced reads/writes ratio.

This study shows a correlation between the runtime and peak memory usage trends: increasing the number of logical items directly impacts both values. Higher total throughput implies a steeper growth for both peak memory usage and runtime.

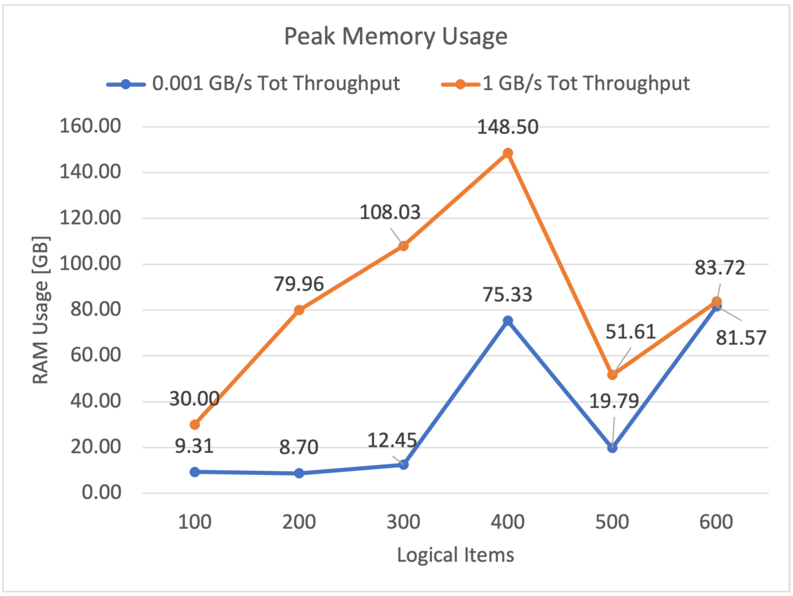


Figure 5.10: Peak Resident Set Size of the Optimizer process for increasing values of logical data items

According to the studies in this section, we can infer that the grouping of physical items into logical items is a significant requirement for the use of this tool. To keep runtime and memory usage sufficiently low, programmers must define groupings in a way that results in the number of logical items to be below a certain threshold. The threshold depends on the desired values of runtime and memory usage. Therefore, when dealing with many data items, one logical item must represent numerous physical items (without incurring in the feasibility issues mentioned earlier) if one wants to decrease the Optimizer runtime/resource usage.

6

Conclusion

Contents

This Chapter concludes the thesis by starting with an overview of the problem statement. It then addresses how PlutusDB can contribute to solving it. Finally, the document will conclude with a comprehensive evaluation of the Optimizer's implementation, along with considerations regarding margins of improvement and real-world applications.

6.1 Conclusions

This thesis approached the problem of optimizing the costs of cloud databases. The dissertation aims at helping users navigate the overwhelming diversity of storage offers and, specifically, choose the right cloud infrastructure for deploying an IaaS database system. One of the criteria for deciding which infrastructure to deploy, which has not been deeply explored yet, is choosing the architecture that minimizes cloud costs. After an overview of existing cloud database systems, we studied recent research on topics of interest, such as data migration and data placement.

The thesis then presented the architecture of PlutusDB, a system that aims at reducing the operational cost of cloud-based applications by:

i) Automatically identifying the most cost-efficient infrastructure definition in cloud-oriented data storages and the corresponding placement of data items

ii) Dynamically migrating data across different types of cloud storage platforms to achieve and maintain the most cost-efficient configuration

PlutusDB is a complex autonomic system that, from the user's perspective, behaves like a standard IaaS database. It uses most of the concepts analyzed in Chapter 2, such as replication, data migration, and data placement. The system is a set of interdependent components that exchange information and work together in a pipeline executed periodically to achieve a common goal: a cluster configuration and an optimal placement of the items that minimize the costs on the user side. PlutusDB periodically self-optimises and transfers data among the multiple back-ends, intending to reduce the operational costs to a minimum, and always aiming at keeping the data items in a cost-optimal configuration.

PlutusDB is a complex system, and the focus of this dissertation was on its key component: the Optimizer, which implementation is presented in Chapter 4. The Optimizer analyzes the data statistics and produces an optimal IaaS configuration, along with the according placement of the items. The design of the Optimizer enables its use as a standalone tool, as well, which helps cloud architects to effortlessly perform a decision on which type and size of the cluster to instantiate for any given workload.

Chapter 5 analyzes the potential benefits of using the Optimizer, highlighting which real-world scenarios could benefit from instantiating hybrid clusters. It then concludes by tackling the scalability aspects of the solution.

We believe the architecture of PlutusDB to be a solid base for an innovative system that could significantly improve the high costs of managing the database systems of modern applications. PlutusDB's framework leaves wide margins for implementation decisions and strategies of the single components, only defining their general behavior and ultimate goal. It also gives some significant and precious suggestions regarding possible implementation strategies, facilitating the work of future researchers.

Finally, we must highlight how the Optimizer, which has been made publicly available on GitHub, could be an extremely functional tool for deciding which cluster(s) to instantiate according to the data items' characteristics. This capability is useful aside from the potential cost gains arising from hybrid clusters, and therefore regardless of its function in PlutusDB. This is probably the most important contribution of the dissertation, which shows how the Optimizer, other than being framed in a more complex architecture and being its core component, is a solid tool for modern cloud architects that can be used right out of the box and instantly support business-aware, critical decision making.

6.2 System Limitations and Future Work

In this last section of the dissertation, we would like to focus on analyzing the limitations of the current implementation, along with a detailed list of the unimplemented parts of PlutusDB.

First and foremost, let us address the implementation of the Optimizer. The current status is an optimization tool that could be leveraged by cloud architects, cloud application developers, or even cloud providers who may want to provide it as a service or use it internally to optimize their DBaaS offering.

The major limitation of the Optimizer is scalability. As we have seen in Chapter 5, the number of decision variables significantly impacts the runtimes, as well as the RAM usage, while at the same time influencing the cost savings. The task of the application programmers is then to define items groupings in a smart way, to achieve the best tradeoff between cost saving and optimization time and resource usage. Another possible approach to improve scalability could be revisiting the formulation to try to reduce complexity. Moreover, approximate methods could be used to accelerate it and enable optimizations for higher numbers of logical items. Finally, since it is possible to limit the execution time, a comparative analysis could be performed between accurate solutions and faster, nonoptimal solutions, evaluating tradeoffs between execution times and accuracy.

Let us now address the cost savings obtained from hybrid clusters. As shown in Chapter 5, 44% of the evaluated scenarios yield hybrid clusters with cost savings w.r.t. non-hybrid "traditional" cluster configurations. Additionally, specific scenarios might benefit from cost savings of up to around 84%.

Although this might seem appealing at first glance, storing data in hybrid clusters raises non-trivial issues, partially described in Chapter 3, related to how to efficiently implement two main mechanisms:

i) Migrating data across storage back-ends

ii) determining their placement in real-time

More in detail, there are some components of the PlutusDB architecture that need to be fully designed ad implemented, namely:

- An efficient data placement structure, which can store the placement of data without too much overhead. Examples of similar components are distributed or centralized directories, and approximated data structures leveraging technologies such as Bloom Filters.
- The Execute component, which should be able to access all the sub-clusters, and, through an efficient implementation of the proposed algorithm, transfer the items to achieve any given placement. This component must be carefully designed to avoid data corruption while maintaining efficiency and incur little to no downtime. As hinted in Section 3.6, this component is highly suited for parallel implementations.
- Additionally, since the problem does not currently account for workload changes, the integration of a workload stability predictor is required, and the problem formulation must be updated accordingly.

Finally, it is worth mentioning that although YCSB is a benchmark that aims at accurately reproducing real-world applications, it is still a synthetic benchmark. For this reason, we started exploring other datasets that could elevate the study of the Optimizer by implementing a tool to evaluate the tool with other workloads. In particular, we focused on existing database traces to test the tool's effectiveness in real workloads from existing datasets. However, due to time constraints, we could not complete the testing with real traces and provide solid optimization results. We used object store traces from IBM[1]. According to IBM [7]: "The traces that IBM contributed include read and write requests made against objects in a cloud-based object storage. These traces can help us understand the behavior of cloud workloads and drive new research and insight into enhancing the cloud". We present, in Appendix A, a working example of how to parse such real traces to be able to use the Optimizer against them. This script is an example of how the Monitor component could implement the parsing of statistics from the Knowledge component.

[1] `https://www.ibm.com/cloud/blog/object-storage-traces`

Bibliography

[1] P. Mell and T. Grance, “The NIST Definition of Cloud Computing,” Sep. 2011.

[2] I. , “What is IaaS (Infrastructure-as-a-Service).” [Online]. Available: https://www.ibm.com/topics/iaas

[3] “Amazon web services.” [Online]. Available: https://aws.amazon.com/

[4] “Google cloud platform.” [Online]. Available: https://cloud.google.com/

[5] “Cloud computing services: Microsoft azure.” [Online]. Available: https://azure.microsoft.com/en-us/

[6] “Discover the next generation cloud platform: Oracle cloud infrastructure.” [Online]. Available: https://www.oracle.com/cloud/

[7] “Ibm cloud.” [Online]. Available: https://www.ibm.com/cloud

[8] “What is PaaS?” [Online]. Available: https://www.redhat.com/en/topics/cloud-computing/what-is-paas

[9] “What is PaaS? Platform as a Service Definition [Examples Included].” [Online]. Available: https://searchcloudcomputing.techtarget.com/definition/Platform-as-a-Service-PaaS

[10] M. Cusumano, “Cloud computing and saas as new computing platforms,” *Communications of the ACM*, vol. 53, no. 4, pp. 27–29, 2010.

[11] “What is Software as a Service (SaaS): A Beginner’s Guide.” [Online]. Available: https://www.salesforce.com/in/saas/

[12] “What is DBaaS (Database-as-a-Service).” [Online]. Available: https://www.ibm.com/topics/dbaas

[13] M. Mohamed, O. Altrafi, and M. Ismail, “Relational Vs. NoSQL databases: A survey,” *International Journal of Computer and Information Technology (IJCIT)*, vol. 03, p. 598, May 2014.

[14] E. F. Codd, “Relational database: A practical foundation for productivity,” in *Readings in Artificial Intelligence and Databases*. Elsevier, 1989, pp. 60–68.

[15] “DB-Engines Ranking.” [Online]. Available: https://db-engines.com/en/ranking/relational+dbms

[16] P. N. Weinberg, J. R. Groff, A. J. Oppel, and J. R. Groff, *SQL, the complete reference*, 3rd ed. New York: McGraw-Hill, 2010, oCLC: ocn233549686.

[17] D. P. A. N. Lopes, “We aq l: Scaling relational databases through weak consistency,” Ph.D. dissertation, Universidade de Lisboa, 2016.

[18] Jing Han, Haihong E, Guan Le, and Jian Du, “Survey on NoSQL database,” in *2011 6th International Conference on Pervasive Computing and Applications*. Port Elizabeth, South Africa: IEEE, Oct. 2011, pp. 363–366. [Online]. Available: http://ieeexplore.ieee.org/document/6106531/

[19] A. Nayak, A. Poriya, and D. Poojary, “Type of nosql databases and its comparison with relational databases,” *International Journal of Applied Information Systems*, vol. 5, no. 4, pp. 16–19, 2013.

[20] F. Gessert and N. Ritter, “Scalable data management: NoSQL data stores in research and practice,” in *2016 IEEE 32nd International Conference on Data Engineering (ICDE)*. Helsinki, Finland: IEEE, May 2016, pp. 1420–1423. [Online]. Available: http://ieeexplore.ieee.org/document/7498360/

[21] S. Mazumdar, D. Seybold, K. Kritikos, and Y. Verginadis, “A survey on data storage and placement methodologies for Cloud-Big Data ecosystem,” *Journal of Big Data*, vol. 6, no. 1, p. 15, Dec. 2019. [Online]. Available: https://journalofbigdata.springeropen.com/articles/10.1186/s40537-019-0178-3

[22] Ontotext, “What is a NoSQL Graph Database?” [Online]. Available: https://www.ontotext.com/knowledgehub/fundamentals/nosql-graph-database/

[23] “Selecting hardware for enterprise implementations | Apache Cassandra 2.2.” [Online]. Available: https://docs.datastax.com/en/cassandra-oss/2.2/cassandra/planning/planPlanningHardware.html

[24] “Capacity planning and hardware selection for apache cassandra implementations.” [Online]. Available: https://docs.datastax.com/en/ossplanning/docs/ossCapacityPlanning.html

[25] J. Chu, DataStax, and DataStax Academy, “How to size up an apache cassandra cluster (training),” pp. 20–40. [Online]. Available: https://www.slideshare.net/planetcassandra/201404-cluster-sizing

[26] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, “Dynamo: amazon’s highly available key-value store,” *ACM SIGOPS Operating Systems Review*, vol. 41, no. 6, pp. 205–220, Oct. 2007. [Online]. Available: https://dl.acm.org/doi/10.1145/1323293.1294281

[27] “DB-Engines Ranking.” [Online]. Available: https://db-engines.com/en/ranking/key-value+store

[28] A. Lakshman and P. Malik, “Cassandra: a decentralized structured storage system,” *ACM SIGOPS Operating Systems Review*, vol. 44, no. 2, pp. 35–40, Apr. 2010. [Online]. Available: https://dl.acm.org/doi/10.1145/1773912.1773922

[29] D. Abadi, P. Boncz, S. Harizopoulos, S. Idreos, S. Madden *et al.*, “The design and implementation of modern column-oriented database systems,” *Foundations and Trends® in Databases*, vol. 5, no. 3, pp. 197–280, 2013.

[30] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, “Bigtable: A Distributed Storage System for Structured Data,” *ACM Transactions on Computer Systems*, vol. 26, no. 2, pp. 1–26, Jun. 2008. [Online]. Available: https://dl.acm.org/doi/10.1145/1365815.1365816

[31] “Serializability vs “Strict” Serializability: The Dirty Secret of Database Isolation Levels.” [Online]. Available: https://fauna.com/blog/serializability-vs-strict-serializability-the-dirty-secret-of-database-isolation-levels

[32] C. H. Papadimitriou, “The serializability of concurrent database updates,” *Journal of the ACM*, vol. 26, no. 4, pp. 631–653, Oct. 1979. [Online]. Available: https://dl.acm.org/doi/10.1145/322154.322158

[33] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O’Neil, and P. O’Neil, “A critique of ANSI SQL isolation levels,” *ACM SIGMOD Record*, vol. 24, no. 2, pp. 1–10, May 1995. [Online]. Available: https://dl.acm.org/doi/10.1145/568271.223785

[34] G. F. Coulouris, Ed., *Distributed systems: concepts and design*, 5th ed. Boston: Addison-Wesley, 2012.

[35] D. Pritchett, “Base: An acid alternative: In partitioned databases, trading some consistency for availability can lead to dramatic improvements in scalability.” *Queue*, vol. 6, no. 3, pp. 48–55, 2008.

[36] John, “ACID versus BASE for database transactions,” Jul. 2009. [Online]. Available: https://www.johndcook.com/blog/2009/07/06/brewer-cap-theorem-base/

[37] A. S. Tanenbaum and M. van Steen, *Distributed Systems: Principles and Paradigms*, 2nd ed. Upper Saddle River, NJ: Pearson Prentice Hall, 2007.

[38] Massachusetts Computer Associates, Inc. and L. Lamport, “Time, clocks, and the ordering of events in a distributed system,” in *Concurrency: the Works of Leslie Lamport*, D. Malkhi, Ed. Association for Computing Machinery, Oct. 2019. [Online]. Available: https://dl.acm.org/citation.cfm?id=3335934

[39] Lamport, “How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs,” *IEEE Transactions on Computers*, vol. C-28, no. 9, pp. 690–691, Sep. 1979. [Online]. Available: http://ieeexplore.ieee.org/document/1675439/

[40] P. Viotti and M. Vukolić, “Consistency in non-transactional distributed storage systems,” *ACM Computing Surveys (CSUR)*, vol. 49, no. 1, pp. 1–34, 2016.

[41] D. Bermbach and J. Kuhlenkamp, “Consistency in distributed storage systems,” in *International Conference on Networked Systems*. Springer, 2013, pp. 175–189.

[42] W. Vogels, “Eventually consistent: Building reliable distributed systems at a worldwide scale demands trade-offs? between consistency and availability.” *Queue*, vol. 6, no. 6, pp. 14–19, 2008.

[43] R. Koller, L. Marmol, R. Rangaswami, S. Sundararaman, N. Talagala, and M. Zhao, “Write policies for host-side flash caches,” in *11th USENIX Conference on File and Storage Technologies (FAST 13)*, 2013, pp. 45–58.

[44] K. Birman, “The promise, and limitations, of gossip protocols,” *ACM SIGOPS Operating Systems Review*, vol. 41, no. 5, pp. 8–13, 2007.

[45] R. Ladin, B. Liskov, L. Shrira, and S. Ghemawat, “Providing high availability using lazy replication,” *ACM Transactions on Computer Systems*, vol. 10, no. 4, pp. 360–391, Nov. 1992. [Online]. Available: https://dl.acm.org/doi/10.1145/138873.138877

[46] U. Sharma, “Internals of DynamoDB,” Jun. 2020. [Online]. Available: https://medium.com/@uditsharma/internals-of-dynamodb-b3b7912256ae

[47] T. Lorido-Botran, J. Miguel-Alonso, and J. A. Lozano, “A review of auto-scaling techniques for elastic applications in cloud environments,” *Journal of grid computing*, vol. 12, no. 4, pp. 559–592, 2014.

[48] “Scaling Stateful Services.” [Online]. Available: https://www.infoq.com/news/2015/11/scaling-stateful-services/

[49] C. McCaffrey, “CraftConf 2016 - Building Scalable Stateful Services,” Apr. 2016. [Online]. Available: https://speakerdeck.com/caitiem20/craftconf-2016-building-scalable-stateful-services

[50] S. Das, F. Li, V. R. Narasayya, and A. C. König, “Automated Demand-driven Resource Scaling in Relational Database-as-a-Service,” in *Proceedings of the 2016 International Conference on Management of Data*. San Francisco California USA: ACM, Jun. 2016, pp. 1923–1934. [Online]. Available: https://dl.acm.org/doi/10.1145/2882903.2903733

[51] Hasgeek TV, “Why it is harder to scale with stateful, data-driven systems - Regunath B,” Jul. 2018. [Online]. Available: https://www.youtube.com/watch?v=0RFOPNIro1A

[52] D. Didona, P. Romano, S. Peluso, and F. Quaglia, “Transactional Auto Scaler: Elastic Scaling of Replicated In-Memory Transactional Data Grids,” *ACM Transactions on Autonomous and Adaptive Systems*, vol. 9, no. 2, pp. 1–32, Jul. 2014. [Online]. Available: https://dl.acm.org/doi/10.1145/2620001

[53] V. Cardellini, M. Nardelli, and D. Luzi, “Elastic stateful stream processing in storm,” in *2016 International Conference on High Performance Computing & Simulation (HPCS)*. Innsbruck, Austria: IEEE, Jul. 2016, pp. 583–590. [Online]. Available: http://ieeexplore.ieee.org/document/7568388/

[54] “Benchmarking Cassandra Scalability on AWS — Over a million writes per second | by Netflix Technology Blog | Netflix TechBlog.” [Online]. Available: https://netflixtechblog.com/benchmarking-cassandra-scalability-on-aws-over-a-million-writes-per-second-39f45f066c9e

[55] C. Lu, “Aqueduct: Online data migration with performance guarantees,” in *Conference on File and Storage Technologies (FAST 02)*, 2002.

[56] S. Das, S. Nishimura, D. Agrawal, and A. El Abbadi, “Albatross: lightweight elasticity in shared storage databases for the cloud using live data migration,” *Proceedings of the VLDB Endowment*, vol. 4, no. 8, pp. 494–505, May 2011. [Online]. Available: https://dl.acm.org/doi/10.14778/2002974.2002977

[57] A. J. Elmore, S. Das, D. Agrawal, and A. El Abbadi, “Zephyr: live migration in shared nothing databases for elastic cloud platforms,” in *Proceedings of the 2011 international conference on Management of data - SIGMOD ’11*. Athens, Greece: ACM Press, 2011, p. 301. [Online]. Available: http://portal.acm.org/citation.cfm?doid=1989323.1989356

[58] S. Barker, Y. Chi, H. J. Moon, H. Hacigümüş, and P. Shenoy, “”Cut me some slack”: latency-aware live migration for databases,” in *Proceedings of the 15th International Conference on Extending*

Database Technology - EDBT '12. Berlin, Germany: ACM Press, 2012, p. 432. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2247596.2247647

[59] S. Barker, Y. Chi, H. Hacigümüs, P. Shenoy, and E. Cecchet, "Shuttledb: Database-aware elasticity in the cloud," in *11th International Conference on Autonomic Computing (ICAC 14)*, 2014, pp. 33–43.

[60] J. Wang, P. Shang, and J. Yin, "Draw: A new data-grouping-aware data placement scheme for data intensive applications with interest locality," in *Cloud Computing for Data-Intensive Applications*. Springer, 2014, pp. 149–174.

[61] J. Paiva and L. Rodrigues, "On Data Placement in Distributed Systems," *ACM SIGOPS Operating Systems Review*, vol. 49, no. 1, pp. 126–130, Jan. 2015. [Online]. Available: https://dl.acm.org/doi/10.1145/2723872.2723890

[62] DataStax, "Apache cassandra™ architecture." [Online]. Available: https://www.datastax.com/resources/whitepaper/apache-cassandratm-architecture

[63] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H.-A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni, "PNUTS: Yahoo!'s hosted data serving platform," *Proceedings of the VLDB Endowment*, vol. 1, no. 2, pp. 1277–1288, Aug. 2008. [Online]. Available: https://dl.acm.org/doi/10.14778/1454159.1454167

[64] C. Curino, E. Jones, Y. Zhang, and S. Madden, "Schism: a workload-driven approach to database replication and partitioning," *Proceedings of the VLDB Endowment*, vol. 3, no. 1-2, pp. 48–57, Sep. 2010. [Online]. Available: https://dl.acm.org/doi/10.14778/1920841.1920853

[65] G.-w. You, S.-w. Hwang, and N. Jain, "Scalable Load Balancing in Cluster Storage Systems," in *Middleware 2011*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, vol. 7049, pp. 101–122, series Title: Lecture Notes in Computer Science. [Online]. Available: http://link.springer.com/10.1007/978-3-642-25821-3_6

[66] J. Paiva, P. Ruivo, P. Romano, and L. Rodrigues, "AUTOPLACER: Scalable Self-Tuning Data Placement in Distributed Key-value Stores," in *10th International Conference on Autonomic Computing (ICAC 13)*. San Jose, CA: USENIX Association, Jun. 2013, pp. 119–131. [Online]. Available: https://www.usenix.org/conference/icac13/technical-sessions/presentation/paiva

[67] A. Kaur, P. Gupta, M. Singh, and A. Nayyar, "Data Placement in Era of Cloud Computing: a Survey, Taxonomy and Open Research Issues," *Scalable Computing: Practice and Experience*, vol. 20, no. 2, pp. 377–398, May 2019. [Online]. Available: https://www.scpe.org/index.php/scpe/article/view/1530

[68] “Linear programming - definition, formula, problem, examples.” [Online]. Available: https://www.cuemath.com/algebra/linear-programming/

[69] H. Arsham, “Deterministic modeling: linear optimization with applications,” *Web site (University of Baltimore): http://home. ubalt. edu/ntsbarsh/opre640a/partIII. htm*, vol. 428, 2014.

[70] G. B. Dantzig, A. Orden, P. Wolfe *et al.*, “The generalized simplex method for minimizing a linear form under linear inequality restraints,” *Pacific Journal of Mathematics*, vol. 5, no. 2, pp. 183–195, 1955.

[71] “Linear programming,” Oct 2022. [Online]. Available: https://en.wikipedia.org/wiki/Linear_programming#CITEREFDantzigThapa2003

[72] R. G. Bland, D. Goldfarb, and M. J. Todd, “The ellipsoid method: A survey,” *Operations research*, vol. 29, no. 6, pp. 1039–1091, 1981.

[73] J. Hooker, “Karmarkar’s linear programming algorithm,” *Interfaces*, vol. 16, no. 4, pp. 75–90, 1986.

[74] M. B. Cohen, Y. T. Lee, and Z. Song, “Solving linear programs in the current matrix multiplication time,” *Journal of the ACM (JACM)*, vol. 68, no. 1, pp. 1–39, 2021.

[75] J. E. Kelley, Jr, “The cutting-plane method for solving convex programs,” *Journal of the society for Industrial and Applied Mathematics*, vol. 8, no. 4, pp. 703–712, 1960.

[76] “Branch and bound,” Aug 2022. [Online]. Available: https://en.wikipedia.org/wiki/Branch_and_bound

[77] S. Boyd and J. Mattingley, “Branch and bound methods,” *Notes for EE364b, Stanford University*, pp. 2006–07, 2007.

[78] B. Selman and C. P. Gomes, “Hill-climbing search,” *Encyclopedia of cognitive science*, vol. 81, p. 82, 2006.

[79] D. Bertsimas and J. Tsitsiklis, “Simulated annealing,” *Statistical science*, vol. 8, no. 1, pp. 10–15, 1993.

[80] F. Glover and M. Laguna, “Tabu search,” in *Handbook of combinatorial optimization*. Springer, 1998, pp. 2093–2229.

[81] “Cplex optimizer.” [Online]. Available: https://www.ibm.com/analytics/cplex-optimizer

[82] R. Lima and E. Seminar, “Ibm ilog cplex-what is inside of the box,” in *Proc. 2010 EWO Seminar*, 2010, pp. 1–72.

[83] “The fastest solver,” Oct 2022. [Online]. Available: https://www.gurobi.com/

[84] J. A. Nelder and R. Mead, “A simplex method for function minimization,” *The computer journal*, vol. 7, no. 4, pp. 308–313, 1965.

[85] Gurobi Optimization, Inc., “Gurobi Optimizer Algorithms I - Basics.” [Online]. Available: https://assets.gurobi.com/pdfs/user-events/2017-frankfurt/Algorithms-I.pdf

[86] “Coin-or,” Aug 2022. [Online]. Available: https://en.wikipedia.org/wiki/COIN-OR

[87] Gurobi Optimization LLC, “Gurobi 8 Performance Benchmarks.” [Online]. Available: https://assets.gurobi.com/pdfs/benchmarks.pdf

[88] “Gurobi Constraints,” 1 2019. [Online]. Available: https://www.gurobi.com/documentation/9.5/refman/constraints.html

[89] “Gurobi Solver Parameters,” 1 2019. [Online]. Available: https://www.gurobi.com/documentation/9.5/refman/parameters.html

[90] “Gurobi Status Codes,” 1 2019. [Online]. Available: https://www.gurobi.com/documentation/9.5/refman/constraints.html

[91] J. O. Kephart and D. M. Chess, “The vision of autonomic computing,” *Computer*, vol. 36, no. 1, pp. 41–50, 2003.

[92] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed, “Zookeeper: Wait-free coordination for internet-scale systems,” in *2010 USENIX Annual Technical Conference (USENIX ATC 10)*, 2010.

[93] Z. Liu, Y. Zhang, E. Y. Chang, and M. Sun, “Plda+ parallel latent dirichlet allocation with data placement and pipeline processing,” *ACM Transactions on Intelligent Systems and Technology (TIST)*, vol. 2, no. 3, pp. 1–18, 2011.

[94] J. Ren, X. Chen, D. Liu, Y. Tan, M. Duan, R. Li, and L. Liang, “A machine learning assisted data placement mechanism for hybrid storage systems,” *Journal of Systems Architecture*, vol. 120, p. 102295, 2021.

[95] A. Khan, X. Yan, S. Tao, and N. Anerousis, “Workload characterization and prediction in the cloud: A multiple time series approach,” in *2012 IEEE Network Operations and Management Symposium*. IEEE, 2012, pp. 1287–1294.

[96] J. Kumar and A. K. Singh, “Workload prediction in cloud using artificial neural network and adaptive differential evolution,” *Future Generation Computer Systems*, vol. 81, pp. 41–52, 2018.

[97] R. N. Calheiros, E. Masoumi, R. Ranjan, and R. Buyya, “Workload prediction using arima model and its impact on cloud applications’ qos,” *IEEE transactions on cloud computing*, vol. 3, no. 4, pp. 449–458, 2014.

[98] K. Cetinski and M. B. Juric, “Ame-wpc: Advanced model for efficient workload prediction in the cloud,” *Journal of Network and Computer Applications*, vol. 55, pp. 191–201, 2015.

[99] J. Saltzer and M. F. Kaashoek, *Principles of computer system design: an introduction*. Morgan Kaufmann, 2009.

[100] “High-Performance Block Storage– Amazon EBS Pricing – Amazon Web Services.” [Online]. Available: https://aws.amazon.com/ebs/pricing/

[101] B. Gou, “Generalized integer linear programming formulation for optimal pmu placement,” *IEEE transactions on Power Systems*, vol. 23, no. 3, pp. 1099–1104, 2008.

[102] Ž. Popović, B. Brbaklić, and S. Knežević, “A mixed integer linear programming based approach for optimal placement of different types of automation devices in distribution networks,” *Electric Power Systems Research*, vol. 148, pp. 136–146, 2017.

[103] A. Rosich, R. Sarrate, and F. Nejjari, “Optimal sensor placement for fdi using binary integer linear programming,” in *20th International Workshop on Principles of Diagnosis*, 2009, pp. 235–242.

[104] “How to specify an if-then constraint with an integer linear programming (ilp) solver,” May 2009. [Online]. Available: http://www.yzuda.org/Useful_Links/optimization/if-then-else-02.html

[105] E. Giorio, “Plutusdb,” https://github.com/EnricoSteez/HybridDB, 2021.

[106] B. Bixby, “The gurobi optimizer,” *Transp. Re-search Part B*, vol. 41, no. 2, pp. 159–178, 2007.

[107] J. P. Pedroso, “Optimization with gurobi and python,” *INESC Porto and Universidade do Porto, Porto, Portugal*, vol. 1, 2011.

[108] “Object storage traces: A treasure trove of information for optimizing cloud workloads.” [Online]. Available: https://www.ibm.com/cloud/blog/object-storage-traces

A

Trace parser script

This appendix contains the code of the parser for the IBM traces [108].

Listing A.1: Parser for IBM Traces

```python
from cmath import inf
import re

objects = dict()  # object_id : [ size, tp_read, tp_write ]
min_ts = inf
max_ts = 0
pattern = re.compile("^[0-9]+ (\w)+\.(\w)+\.(\w)+ [0-9]+( [0-9]+ [0-9]+)?")
with open("IBMObjectStoreTrace000Part0", "r") as file, open(
    "traces.txt", "w"
) as output:
    for line in file:
        if not re.match(pattern, line):
            print(f"line {line} doesn't match")
            continue

        line = line.split()
        try:
            ts = int(line[0])
        except ValueError:
            continue
        op = line[1].split(".")[1]  # REST.GET.OBJECT -> GET
        object_id = line[2]
        object_size = line[3]
        # print(f"Found object {object_id}")
        if object_id in objects:
            value = objects[object_id]
            if op == "GET" or op == "HEAD":  # READ
                new_value = (value[0], value[1] + 1, value[2])
            else:  # WRITE
                new_value = (value[0], value[1], value[2] + 1)

            objects[object_id] = new_value
        else:
            # print(f"First seen, op={op}")
            if op == "GET" or op == "HEAD":  # READ
                objects[object_id] = (object_size, 1, 0)
            else:
                objects[object_id] = (object_size, 0, 1)

        if ts > max_ts:
            max_ts = ts
        if ts < min_ts:
            min_ts = ts

    time = max_ts - min_ts
    for (size, tp_r, tp_w) in objects.values():
        output.write(f"{size} {tp_r/time:.3f} {tp_w/time:.3f}\n")
```