Optimization of Cloud Storage Costs Through Efficient Data Placement

Enrico Giorio
enrico.giorio@tecnico.ulisboa.pt

Instituto Superior Técnico, Lisboa, Portugal

November 2022

Abstract

Cloud databases are nowadays available in various deployment models. When deciding which type of storage to employ for an application, 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 tendencies see cloud architects deploying homogeneous infrastructures, i.e. made of an arbitrary number of nodes of the same type. However, when the data to be stored 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. Let alone periodically modifying the infrastructure to maintain cost-efficiency when facing significant workload changes. 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, Data Placement, Data Migration, Linear Optimization, Cost Efficiency

1. Introduction

The advent of cloud-based database services opened new possibilities for small businesses and enterprises. Currently, multiple cloud database technologies and paradigms are available, each 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).

All major cloud providers offer a wide variety of types of Virtual Machines (VMs) for customers to choose from, based on the properties of the applications they intend to run. Most of the time, the infrastructure for running a database system is composed of nodes of the same type, to facilitate the process of choosing and maintaining it.

When the data to store in an IaaS database system is known to uniformly generate a specific workload pattern, choosing which node type to deploy in multiple instances is not a challenging task. However, modern applications need to store unbalanced sets of data where access characteristics vary significantly among data items. In these scenarios, there is no clear advantage in adopting such a strategy other than the 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 hard, in fact, that it is hard, if not impossible, for human operators to consistently find infrastructure configurations that optimally minimize the costs. Let alone periodically modify the infrastructure to maintain cost efficiency when facing significant workload changes.

An ideal cost-efficient database system should be deployed on a heterogeneous cluster of machines, and selectively store data items in (possibly different) nodes whose cost characteristics are best suited for their access pattern. Such a system must tackle several critical 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.

Data placement, scaling approaches, and migration techniques have been extensively researched in recent years. However, very few projects address a cost-oriented approach, often aiming at developing performance-oriented algorithms and systems.

This thesis presents 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 with an independent database instance. 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 solves an Integer Linear Optimization (ILP) problem to achieve an ideal placement that minimizes 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 and Contributions

This thesis is developed around three main objectives:

1. Presenting PlutusDB in its entirety, through an overview of its internal model, a detailed description of each component, and additional design proposals for the main operational functions.

2. At this point, the reader should have a complete idea of the concept behind the design of PlutusDB and the tasks that each of its components must accomplish. The thesis can then presents the implementation of the Optimizer. This component is the main building block for PlutusDB, around which all other elements are designed.

3. Showing the reader performance, usability, and limitations of the Optimizer as a standalone tool.

This master thesis contributes with presenting the challenging project of PlutusDB: its architecture, a clear definition of the execution flows, a detailed explanation of the interactions among the internal components, and finally, the proposal of a proprietary algorithm for data transfer.

The second and main contribution is the implementation of the Optimizer.

The Optimizer has been tested and evaluated with datasets inspired by a popular benchmark for NoSQL databases, YCSB[7], with the aim of finding datasets that, given their workload characteristics, are cost-wise better suited to be stored in hybrid clusters. That is, clusters composed of instances of multiple machine types.

This study proves that there exist datasets that could benefit from instantiating a database in hybrid clusters, with hourly cost-saving up to 84% compared to the cost of the best non-hybrid, traditional cluster configuration.

2. Methodology

This master thesis proposes a distributed system deployed in the cloud, which leverages scaling mechanisms, data placement techniques, and a data migration algorithm. After an overview of existing cloud delivery models, the three topics have been explored in detail. From each of them, we report recent publications and state-of-the-art research.

2.1. Scaling cloud components

Given the stateful nature of distributed databases, we focus on solutions for scaling stateful components in the cloud. Scaling stateful components is a complex task. Data must be redistributed, and new cluster members need to be brought up to date, possibly by existing members, to serve requests. Capacity planning is another challenging aspect of scaling stateful components: as the number of nodes in the system grows, the performance of distributed databases exhibits explicit nonlinear behaviors. Such behaviors are attributed to the effects of contention of physical and logical resources. Studied papers address these issues with different approaches and for different types of systems. Some of them are described next.

Transactional Auto Scaler Transactional Auto Scaler (TAS) [12] is a system for automating the elastic scaling of replicated in-memory transactional data grids such as Red Hat Inifinispan.

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. The performance predictor, through the joint usage of Analytical Modeling and Machine Learning models, using the workload characteristic and the platform scale, outputs several indices which are used to identify the optimal platform size.

Apache Cassandra An example of scaling for stateful services such as key-value stores is provided by Apache Cassandra [17]. In Cassandra, nodes are responsible for a range of item keys.

When a new node is added to the system, it splits a range of keys that some other heavily loaded node was previously responsible for, alleviating it.

As demonstrated [2], Cassandra's scalability is linear with a correct and optimal keyspace configuration, which makes capacity planning straightforward once the workload is known.

2.2. Data Migration

One additional issue with scaling stateful services regards data migration. Adding machines to a stateful component requires 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 [18]. Following, is an overview of some of the analyzed papers that address data migration.

Albatross. Albatross [9] proposes a live migration technique for multitenant databases to ensure Service Level Agreements (SLAs) to the tenants whenever the machines they are deployed on get overloaded. Albatross's migration protocol is divided into phases.

Phase 1 starts with a snapshot of the source database cache (*src*) that is then transferred to the destination VM (*dst*). *src* continues serving transactions while *dst* is initialized with the snapshot. Therefore, 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 terminates 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* is transferred from *src* to *dst*. *Src* stops 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.

Zephyr. Zephyr [13] is a similar system focused on shared-nothing architectures, which makes it more suited to migrate data in stateful services. The critical innovation of Zephyr is the "dual mode" where, during migration, both *src* and *dst* execute transactions.

ShuttleDB. Another rather innovative approach to data migration is proposed in ShuttleDB [5]. The system combines VM elasticity with lower-level, database-aware elasticity. 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.

2.3. Data Placement

Most existing distributed systems assign data to nodes to achieve optimal load balancing, lower access latency, and higher fault tolerance [24, 19]. This thesis' goal and innovative approach is to optimize data placement based on cost-efficiency as the primary objective, which has not often been a concern to cloud providers or researchers.

One basic, although widely used, approach to distributing data among several nodes, used by state-of-the-art key-value stores [11, 17, 11, 10] is consistent hashing. Systems leveraging consistent hashing assign 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 same circular ring, in a distributed hash table way. The node whose ring placement is after the key hash is responsible for storing that key.

More advanced systems leverage data streams and real-time analysis to automatically place data in the "best" locations, according to more meaningful information, such as generated bandwidth.

Schism. Schism [8] focuses on the throughput of distributed transactions being 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.

It 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. Schism replicates the tuples that are shared between partitions in order to maximize performance by exploiting data locality and minimizing the number of distributed transactions.

AutoPlacer. AutoPlacer [20] works with NoSQL key-value stores such as Red Hat Infinispan and Apache Cassandra [17] 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 uses a round-based distributed optimization algorithm: in each round, the system decides on which nodes to place the top-k critical items to increase the correlation between the data each node is requesting and storing.

AutoPlacer adopts a state-of-the-art stream analysis algorithm to identify the top-k most frequent items of a stream. It then instantiates an ILP problem to find the optimal placement of only the top-k items, to significantly reduce the number of decision variables.

3. Proposed Solution

When deploying database systems as IaaS, the choice of the underlying infrastructure is always a critical point. Cloud architects often choose to deploy an underlying infrastructure composed of the same types of nodes, as selecting multiple types of nodes requires performing complex evaluations that often become an overhead. However, choosing which single type offers the best price/performance ratio according to the expected application load model can still be an overwhelming task.

In face of the need to store an unbalanced set of data, where access characteristics vary significantly among data items, using only one node type might be sub-optimal regarding resource usage, and therefore cost efficiency. Consequently, an ideal solution 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 less cost-efficient than other node types when storing low-throughput 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 performs optimizations periodically to maintain cost-optimality.

PlutusDB uses multiple instances of backend databases as underlying infrastructure, each with its dedicated nodes. The idea is that, in specific conditions, the generated hybrid configuration of nodes 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 optimization (grouping single physical data items into fewer logical ones) to improve performance and scalability.

3.1. PlutusDB Architecture

We start with a high-level overview of PlutusDB's architecture (illustrated in Figure 1).

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 and main innovation of PlutusDB: the Optimizer, implementation of the idea on top of which the whole architecture is designed.

Managed Element. The Managed Element is the set of underlying IaaS sub-clusters. Each corresponding to a database instance, which is self-contained.

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

Database Proxy. This is the proxy of the system, which will serve as the entry point for interacting with the database system.

PlutusDB aims at being a fully autonomic system that periodically evaluates the optimal items' placement to maintain cost-efficiency when facing workload changes. This periodic evaluation, which 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. Through the solving of an ILP problem, it produces an optimal data placement that minimizes the costs of usage of the cloud infrastructure. This phase should also decide whether any data should be transferred among the backends, although this functionality is not implemented in this thesis.
- Execute: Takes as input the list of data to be transferred, the output of the preceding stage, and executes the transfer with the algorithm described in section 3.1.

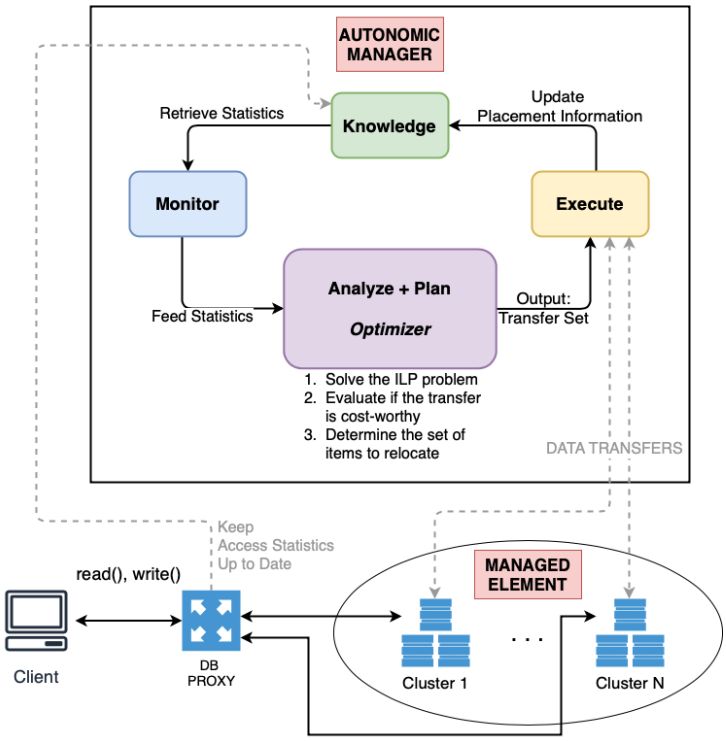

Figure 1: PlutusDB Architecture.

Being a Key-Value store, PlutusDB operates with a per-item granularity, and it offers support to atomic reads and writes in the form of:

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

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.

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 have not been precisely modeled in this research.

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.

The internal components of PlutusDB are:

Knowledge component. The goal of the Knowledge component is to store information regarding the system's situation at any point in time, which includes placement and up-to-date statistics of currently stored data items, operational costs, and status of the managed database clusters.

Monitor component. This component is in charge of retrieving up-to-date statistics from the Knowledge storage and feed them to the Optimizer. Statistics must be in a specific format for the Optimizer to work correctly. If the Knowledge component does not store statistics in such format, the Monitor component is in charge of processing them.

Analyze + Plan component. The optimization phase is going to be the core contribution of this thesis. It is the "brain" of the system that orchestrates and directs the data placement and, with it, the data transfers.

PlutusDB introduces a novel approach that explicitly models 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 an ILP problem that, through the minimization of the cost of the overall infrastructure (objective function), achieves an optimal data placement (decision variables).

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. The current prototype implementation 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.

Execute component. The Execute component is in charge of transferring the data items among subclusters to achieve the ideal and cost-optimal configuration, output of the Optimizer component.The Execute component should 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 2.2. A proposal of an algorithm that might achieve the desired goal is now presented. 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.

In the first phase of the algorithm (**Static Phase**), the Execute component takes a snapshot of the status of the data that must be transferred (at the source database). All the involved data items are then transferred to the destination database while the sending database keeps serving requests and logging them to a local operations log.

The second phase (**Dynamic 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 i starts with comparing the present status of the items that have been transferred during round $i-1$. We call this difference "Delta". The Delta, if not empty, is then transferred while the sending database keeps serving requests and recording them in the local log. Once the destination database has finished applying the Delta, round $i+1$ can start, comparing snapshots at the start and at the end of round i. The iterative procedure continues until the number of operations in the last Delta is $\varnothing$ (or containing only a few operations), declaring the end of the Dynamic Phase. As proved in Albatross [9], the Deltas typically decrease in size, eventually becoming empty and therefore terminating the algorithm.

At this point, both sending and receiving databases must briefly stop serving user requests (**Termination Phase**) to allow the destination to apply the last Delta, ensuring at the same time that no operations target any items that are part of the last Delta. After the last Delta has been applied, the system resumes serving client requests and the new configuration is effective.

3.2. The Optimizer

Any IaaS cluster, at its most basic configuration, is a set of computing nodes. Each node consisting of a computational instance with an attached volume. The term "node type" defines a unique combination of VM type and volume type. For this implementation, we refer to AWS [1] as the provider of reference for the cost models. Virtual Machines of different families (i.e., optimized for different types of workloads) and multiple volume types were considered, to explore a wide range of workload-specific node-types. Moreover, for this analysis, we assume that the managed database is deployed following DataStax's recommendations[3, 15], where each instance has a dedicated volume with a 4TB capacity.

The cost model of an IaaS cluster is identified by the sum of the cost components of instantiated VMs and volumes:

i) Cost of usage of the instantiated VMs.
ii) Cost of volumes storage.
iii) Cost of volumes bandwidth usage.
iv) Cost of volumes IOPS usage (defined as a function of the bandwidth usage).

As described in section 3.1, the Optimizer, implementation of the Analyze+Plan component, aims at generating a possibly hybrid cluster and according data placement that minimize the operational costs of the overall database. There might be situations, in fact, 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 such 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 that directly tells the user which is the best machine that suits the input workload, making the choice of the infrastructure straightforward.

The optimal placement is achieved through the solving of an ILP problem, whose main decision variables represent the placement of data items. The goal is to potentially have multiple database instances, each 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 RF nodes. Therefore, each item's generated bandwidth will be a function of the nominal, application-generated throughput, and RF. Under these circumstances, we define the problem as a mathematical formulation that precedes the implementation.

The main set of decision variables, which we call $\mathbf{X}$, is modeled as a 3-dimensional matrix. As is usual in ILP problems whose goal is to identify an optimal data placement in a distributed context [16, 22, 23], the decision variables in the matrix are binary. $\mathbf{X}$ is defined over:

- 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.
  $P = \{(a, b) | a \in VM\_types, b \in volume\_types\}$.
- The z-axis represents all the possible instances that can be instantiated for a given node type, which go from 0 to a predefined maximum of K.

We optimize the formulation by introducing the concept of *logical* instances, which are groups of RF physical machines. This approach effectively reduces the size of the matrix by RF times. We state that each sub-cluster can be made of $[0..K-1]$ *logical* instances.

The meaning of each bit in the 3-dimensional placement matrix is explained in equation 1.

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

The constants used in the mathematical formulation are:

- $\vec{s}$: item sizes, $s_i \Rightarrow$ size of item i [GB]
- $\vec{t^r}$ and $\vec{t^w}$: IO Per Second, $t_i^r \Rightarrow$ read access ratio of item i, $t_i^w \Rightarrow$ write access ratio of item i, both in [OPS/s]
- max_size: size of attached volumes.
- RF: Replication Factor
- $\vec{iops}$ and $\vec{tp}$, defined $\forall j \in P$: maximum IOPS and maximum throughput per each node type.
- $\vec{cost}$: costs per hour of each node type.
- $\vec{cost}^{storage}$, $\vec{cost}^{iops}$ and $\vec{cost}^{tp}$: Cost of Storage, Cost of IOPS and Cost of Throughput of each node type.
- σ is a constant to transform the bandwidth in billed IOPS according to AWS's specifications.

The other set of decision variables, bi-dimensional matrix $\mathbf{z}$, tracks whether each logical instance k of each node type j is instantiated or not, and is defined per every possible $(j, k) | j \in P, k \in [0..K-1]$ pair).

$$z_{jk} = \begin{cases} 1 & \text{Logical Instance } k \text{ of node type } j \text{ is} \\ & \text{instantiated} \\ 0 & \text{Otherwise} \end{cases} \tag{2}$$

The correlation between $\mathbf{X}$ and $\mathbf{z}$ is defined in eqs. (3a) and (3b):

$$\sum_{i=0}^{N-1} \mathbf{X}_{ijk} = 0 \iff \mathbf{z}_{jk} = 0 \tag{3a}$$

$$\sum_{i=0}^{N-1} \mathbf{X}_{ijk} > 0 \iff \mathbf{z}_{jk} = 1 \tag{3b}$$

The problem formulation is shown in Appendix A. The decision variables are highlighted in red for better readability. Following, an overview of the objective function and constraints:

- eq. (8a) models the cost per hour of instantiated nodes.
- eq. (8b) models the cost of the allocated storage. AWS bills the provisioned storage, regardless of how much of it is actually used.
- eqs. (8c) and (8d) model the cost of bandwidth and IOPS usage.
- eqs. (9b) to (9d) set an upper bound on the maximum allocated, respectively, storage, IOPS, and bandwidth per node.
- eqs. (9e) and (9f) establish the correlation between decision variables, expressed in inequations eqs. (3a) and (3b).
- eq. (9g) ensures that each item is placed in only one logical instance.

The unit conversions are omitted for readability in the mathematical formulation, although they are implemented directly in the Python code, publicly available on GitHub [14].

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, to calculate the cost savings, if any. We achieve this by reusing 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: $\mathbf{y}$, which meaning is shown in eq. (4).

$$y_j = \begin{cases} 1 & \text{Node type } j \text{ is instantiated at least once} \\ 0 & \text{Otherwise} \end{cases} \tag{4}$$

eqs. (5a) and (5b) show the correlation between $\mathbf{y}$ and $\mathbf{z}$.

$$\sum_{k=0}^{K-1} z_{jk} = 0 \iff y_j = 0 \tag{5a}$$

$$\sum_{k=0}^{K-1} z_{jk} > 0 \iff y_j = 1 \tag{5b}$$

The constraints that must be added to the problem are expressed in eqs. (6a) and (6b).

$$\sum_{k=0}^{K-1} z_{jk} \leq M \cdot y_j \tag{6a}$$

$$\sum_{k=0}^{K-1} z_{jk} \cdot M \geq y_j \tag{6b}$$

Finally, we can insert the final constraint that guides the solver towards choosing only one node type (eq. (7)).

$$\sum_{j \in P} y_j = 1 \tag{7}$$

3.3. Software Choice

The mathematical model must be implemented in a 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 due to its flexibility and the possibility of using Gurobi [4, 6, 21]. Gurobi is a robust solver that, due to the academic license offering, and in combination with Python, allows for highly expressive problem modeling and excellent solving performance without incurring any costs. The implementation of the Optimizer has been made publicly available on GitHub as a fully open-source project [14].

4. Results & discussion

We now show experimental data to demonstrate the effectiveness of the solution in real-world scenarios. YCSB[7] was used as a reference for the characterization of the chosen workloads. Since YCSB normally generates vast amounts of individual data items, we must operate with a coarser granularity. 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. We take inspiration from YCSB's workloads and we extend them to create data sets that can be fed into the Optimizer to extract valuable results. In particular, each data set is composed by N data items, all of which with equal size. The total throughput of the data set is tunable and the items' access ratios are distributed according to a zipfian distribution, with skew 1. Finally, the ratio between reads and writes is also customizable. Each experiment consists of a run of the Optimizer according to a different input data set.

The first study aims at exploring in which data sets the Optimizer yields hybrid clusters as the best configuration. Each experiment has been executed with 300 logical items, varying items' sizes, varying total throughput, and reads % according to YCSB's Workload A (50% Reads/Writes). The levels for the two varying parameters (total throughput and items' sizes) have been selected empirically. The data is presented as a heatmap in fig. 2, where each cell corresponds to a different data set and contains the cost saving achieved by instantiating the hybrid configuration over the best non-hybrid configuration. Naturally, the cost saving is zero when the Optimizer yields a homogeneous cluster as the best outcome. These results suggest that

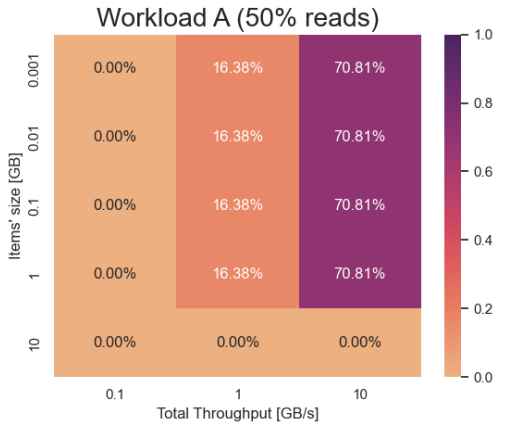

Figure 2: Cost savings for different throughputs and items' sizes, 50% reads - 50% writes

data sets with higher total throughputs are more suited for hybrid clusters, as the overall throughput variability among items is more significant (probably since 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 data sets containing items with the same size.

We now analyze the scalability of the Optimizer. For this purpose, we present two studies (figs. 3 and 4) that track, respectively, the runtime and the peak Resident Set Size (RSS, or total required RAM) of two experiments for increasing numbers of total items. In particular, one experiment will be characterized by a low total throughput and one with a high total throughput, to show the impact that this parameter has on optimization times. 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).

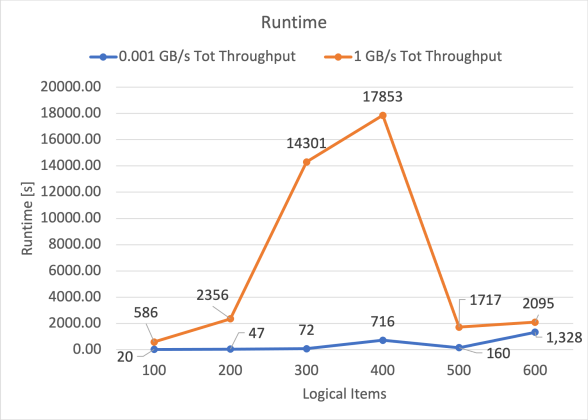

Figure 3: Optimizer processing time for high and low throughput datasets, increasing numbers of logical items

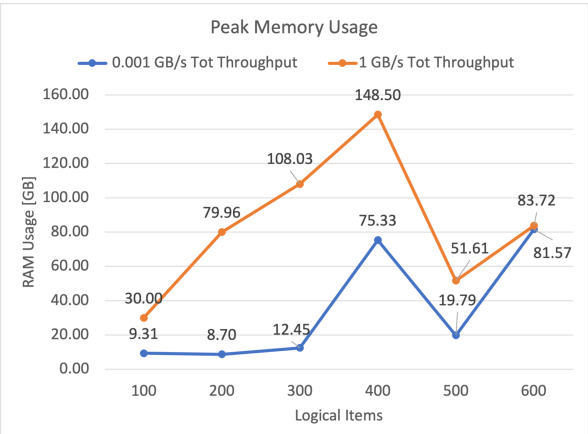

Figure 4: Peak Resident Set Size of the Optimizer process for increasing values of logical data items

In both high throughput and low throughput scenarios, experiments until N=400 yielded a hybrid placement, while experiments with 500 and 600 items did not. This explains a runtime and peak required memory decrease with N=500, as the problem is easier to solve. Moreover, the runtime and RSS increasing trends are steeper with high throughput data sets.

These studies suggest that controlling the number of logical items is critical to achieve better performance and lower hardware requirements. Therefore, we can infer that the grouping of physical items into logical items is a significant requirement for the performance of the Optimizer. Programmers must define groupings in a way that results in the number of logical items to be below a specific threshold, which depends on the desired values of what we can call Key Performance Indicators (KPIs): runtime and pear Resident Set Size.

5. Conclusions

This thesis approaches the problem of optimizing the costs of cloud databases and aims at helping users choose the right cloud infrastructure for deploying an IaaS database system.

The thesis presents 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 an autonomic system that from the user's perspective behaves like a standard IaaS database. The system, however, 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 user costs. The goal is achieved by using some of the concepts analyzed in section 2. PlutusDB is a complex system, and the focus of this thesis is primarily on its key component: the Optimizer.

The Optimizer analyzes the data statistics and produces an optimal IaaS configuration, along with the according placement of the items. Its implementation is presented in section 3 and is publicly available on GitHub [14]. Section 4 analyzes the potential benefits of using the Optimizer, highlighting some real-world scenarios that could benefit from instantiating hybrid clusters. It then concludes by tackling some 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 thesis, 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.

5.1. System Limitations and Future Work

The major limitation of the Optimizer is scalability. As we have seen in section 4, the number of decision variables significantly impacts the runtimes. 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. 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.

Let us now address the cost savings obtained from hybrid clusters. As shown in fig. 2 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, 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. Examples of similar components are distributed or centralized directories, and approximated data structures as proposed in [20].
- The Execute component, which, through an efficient implementation of the proposed algorithm, transfers 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.
- The integration of a workload stability predictor is required to account for workload changes, and the problem formulation must be updated accordingly.

References

[1] Amazon web services.

[2] Benchmarking Cassandra Scalability on AWS — Over a million writes per second | by Netflix Technology Blog | Netflix TechBlog.

[3] Selecting hardware for enterprise implementations | Apache Cassandra 2.2.

[4] The fastest solver, Oct 2022.

[5] S. Barker, Y. Chi, H. Hacıgumus, P. Shenoy, and E. Cecchet. ShuttleDB: Database-Aware Elasticity in the Cloud. page 12, June 2014.

[6] B. Bixby. The gurobi optimizer. *Transp. Research Part B*, 41(2):159–178, 2007.

[7] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 143–154, 2010.

[8] C. Curino, E. Jones, Y. Zhang, and S. Madden. Schism: a workload-driven approach to database replication and partitioning. *Proceedings of the VLDB Endowment*, 3(1-2):48–57, Sept. 2010.

[9] 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*, 4(8):494–505, May 2011.

[10] DataStax. Apache Cassandra™ Architecture.

[11] 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*, 41(6):205–220, Oct. 2007.

[12] 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*, 9(2):1–32, July 2014.

[13] 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*, page 301, Athens, Greece, 2011. ACM Press.

[14] E. Giorio. Plutusdb. `https://github.com/EnricoSteez/HybridDB`, 2021.

[15] f. given i=JC, DataStax, and DataStax Academy. How to size up an apache cassandra cluster (training).

[16] B. Gou. Generalized integer linear programming formulation for optimal pmu placement. *IEEE transactions on Power Systems*, 23(3):1099–1104, 2008.

[17] A. Lakshman and P. Malik. Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review*, 44(2):35–40, Apr. 2010.

[18] C. Lu, G. A. Alvarez, and J. Wilkes. Aqueduct: online data migration with performance guarantees. page 12.

[19] J. Paiva and L. Rodrigues. On Data Placement in Distributed Systems. *ACM SIGOPS Operating Systems Review*, 49(1):126–130, Jan. 2015.

[20] 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)*, pages 119–131, San Jose, CA, June 2013. USENIX Association.

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

[22] 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*, 148:136–146, 2017.

[23] 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*, pages 235–242, 2009.

[24] 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*, pages 149–174. Springer, 2014.

Appendices

A. Mathematical Formulation

$$C = \sum_{j \in P} \sum_{k=0}^{K-1} z_{jk} \cdot RF \cdot cost_j + \tag{8a}$$

$$\sum_{j \in P} \sum_{k=0}^{K-1} z_{jk} \cdot RF \cdot max\_size \cdot cost_j^{storage} + \tag{8b}$$

$$\sum_{i=0}^{N} \sum_{j \in P} \sum_{k=0}^{K-1} X_{ijk} \cdot (t_i^r + t_i^w \cdot RF) \cdot s_i \cdot cost_j^{tp} + \tag{8c}$$

$$\sum_{i=0}^{N} \sum_{j \in P} \sum_{k=0}^{K-1} X_{ijk} \cdot (t_i^r + t_i^w \cdot RF) \cdot s_i \cdot \sigma \cdot cost_j^{iops} \tag{8d}$$

$$\min_{\mathbf{X}, z} \quad C \tag{9a}$$

subject to:

$$\sum_{i=0}^{N-1} (X_{ijk} \cdot s_i) \leq max\_size \qquad \forall j \in P,\ \forall k < K \tag{9b}$$

$$\sum_{i=0}^{N-1} (X_{ijk} \cdot (t_i^r + t_i^w \cdot RF) \cdot s_i) \leq tp_j \cdot RF \qquad \forall j \in P,\ \forall k < K \tag{9c}$$

$$\sum_{i=0}^{N-1} (X_{ijk} \cdot (t_i^r + t_i^w \cdot RF) \cdot s_i) \cdot \sigma \leq iops_j \cdot RF \qquad \forall j \in P,\ \forall k < K \tag{9d}$$

$$\sum_{i=0}^{N-1} X_{ijk} \leq M \cdot z_{jk} \qquad \forall j \in P,\ \forall k < K \tag{9e}$$

$$\sum_{i=0}^{N-1} X_{ijk} \cdot M \geq z_{jk} \qquad \forall j \in P,\ \forall k < K \tag{9f}$$

$$\sum_{j \in P} \sum_{k=0}^{K-1} X_{ijk} = 1 \qquad \forall i < N \tag{9g}$$