UNIVERSITAT POLITÈCNICA DE CATALUNYA

DOCTORAL THESIS

A framework for multidimensional indexes on distributed and highly-available data stores

Author:
Cesare CUGNASCO

Supervisor:
Dr. Yolanda BECERRA
Co-supervisor:
Dr. Jordi TORRES

A thesis submitted in fulfillment of the requirements for the degree of Doctor of Philosophy

in the

Departament d'Arquitectura de Computadors

February 1, 2019

"Je n'ai fait celle-ci plus longue que parce que je n'ai pas eu le loisir de la faire plus courte. "
"I would have written a shorter letter, but I did not have the time. "

Blaise Pascal, Provincial Letters: Letter XVI (4 December 1656)

Abstract

Cesare CUGNASCO

A framework for multidimensional indexes on distributed and highly-available data stores

Spatial Big Data is considered an essential trend in future scientific and business applications. Indeed, research instruments, medical devices, and social networks generate hundreds of petabytes of spatial data per year. However, as many authors have pointed out, the lack of specialized frameworks dealing with such kind of data is limiting possible applications and probably precluding many scientific breakthroughs. In this thesis, we describe three HPC scientific applications, ranging from molecular dynamics, neuroscience analysis, and physics simulations, where we experience first hand the limits of the existing technologies. Thanks to our experience, we define the desirable missing functionalities, and we focus on two features that when combined significantly improve the way scientific data is analyzed.

On one side, scientific simulations generate complex data sets where multiple correlated characteristics describe each item. For instance, a particle might have a space position (x,y,z) at a given time (t). If we want to find all elements within the same area and period, we either have to scan the whole dataset, or we must organize the data so that all items in the same space and time are stored together. The second approach is called Multidimensional Indexing (MI), and it uses different techniques to cluster and to organize similar data together. On the other side, approximate analytics has been often indicated as a smart and flexible way to explore large data sets in a short period. Approximate analytics includes a broad family of algorithms which aims to speed up analytical workloads by relaxing the precision of the results within a specific interval of confidence. For instance, if we want to know the average age in a group with $\pm$ 1-year precision, we can consider just a random fraction of all the people, thus reducing the amount of calculation. But if we also want less I/O operations, we need *efficient data sampling*, which means organizing data in a way that we do not need to scan the whole data set to generate a random sample of it.

According to our analysis, combining Multidimensional Indexing with efficient data Sampling (MIS) is a vital missing feature not available in the current distributed data management solutions. This thesis aims to solve such a shortcoming and it provides novel scalable solutions. At first, we describe the existing data management alternatives; then we motivate our preference for NoSQL key-value databases. Secondly, we propose an analytical model to study the influence of data models on

the scalability and performance of this kind of distributed database. Thirdly, we use our experience and the analytical model to design two novel multidimensional indexes with efficient data sampling: the D8tree and the AOTree. Our first solution, the D8tree, improves state of the art for approximate spatial queries on static and mostly read dataset. Later, we enhanced the data ingestion capability or our approach by introducing the AOTree, an algorithm that enables the query performance of the D8tree even for HPC write-intensive applications. We compared our solution with PostgreSQL and plain storage, and we demonstrate that our proposal has better performance and scalability.

Finally, we describe Qbeast, the novel distributed system that implements the D8tree and the AOTree using NoSQL technologies, and we illustrate how Qbeast simplifies the workflow of scientists in various HPC applications providing a scalable and integrated solution for data analysis and management.

Acknowledgements

There are plenty of people I want to thank for their patience and help over these last few years, but I will not be able to thank you all.

First, I want to thank my supervisors, Yolanda and Jordi, for supporting me through the thousands of changes of plan, delays and the, sometimes heated, discussions. Can you believe it? It's over! Just kidding, tomorrow is business as usual.

Thanks to the reviewers David, Josep Lluís, Lena, Nico, and Toni for their precious feedback. I know it may seem a formal requirement, but your inputs really helped me reorganize my thoughts and work in a clear and structured way.

Thanks to the colleagues from the CASE department, Antoni, Beatriz, Fernando, Guillaume, Hadrien y Mariano for the engaging use cases and for having listened to me all those times I was praising the virtue of multidimensional indexing.

Thanks to Pol for making the dirty C++ work, and to Eloy for the over-killing framework for running the tests. Without you my work would have been harder and surely more boring.

Thanks to all the colleagues and friends I crossed paths with in room C6E201. We shared many days, lunches, discussions on various topics (independence, Bitcoin, and gossip, to name a few) and most importantly the coffee machine. Seriously though, who left the capsule inside?

Thanks to Raül, that made the project Quake possible and, most importantly, accepts my travel expenses. Thanks to Daniele, for the beautiful Qbeast logo, that sometimes even steals the show from the algorithms I created.

Thanks to all the the people who heard me say time and time again "I can't, I have to work on my thesis!". Damn, now I have to find another excuse!

Thanks to my father, for giving me the ability to ingest thousands of random and mostly useless information, and to my mother, for giving me a positive outlook on life and teaching me to joke even in the darkest hours. I dedicate this thesis to you. To my grandparents, for being a moral example and inspiration. To my sister, for always giving me an opportunity to bicker, and to my brother for bringing peace, usually with a clever pun. To my uncles and aunties for always gathering all of us as one big family. They even included Brando!

Research says that half of the Ph.D. students suffer from some kind of mental disorder, and I must thank my wife Ambra if I didn't fly over the cuckoo's nest. Basically, I didn't have her permission. Jokes aside, she is the best thing that ever happened to me. She also read all my works in the crazy attempt to improve my

writing and to understand what I am doing, so much so that she probably deserves a Ph.D. in computer science as well.

Contents

List of Figures

List of Tables

List of Abbreviations

2PC	**T**wo **P**hase **C**ommit
2PL	**T**wo **P**hase **L**ocking
Aeneas	**A**n **E**xtensible **N**oSQL **E**nhancing **A**pplication **S**ystem
AOtree	**A**symptotic **O**utlook **T**ree
BSC	**B**arcelona **S**upercomputing **C**enter
D8tree	**D**enormalized **O**cta **T**ree
GC	**G**arbage **C**ollection
HPC	**H**igh **P**erformance **C**omputing
HTAP	**H**ybrid **T**ransactional **A**nalytical **P**rocessing
JVM	**J**ava **V**irtual **M**achine
MD	**M**olecular **D**ynamics
MI	**M**ultidimensional **I**ndexing
MIS	**M**ultidimensional **I**ndexing with uniform data **S**ampling
MVCC	**M**ulti**V**ersion **C**oncurrency **C**ontrol
NoSQL	**N**o **SQL** or **N**ot only **SQL**
OLAP	**O**nline **A**nalytical **P**rocessing
OS	**O**perative **S**ystem
RDBMS	**R**elational **D**ata **B**ase **M**anagement **S**ystem
SQL	**S**tructured **Q**uery **L**anguage
XML	**G**arbage **C**ollection
XSD	**X**ML **S**chema **D**efinition

Per la mia Mamma

Chapter 1

Introduction

Technology development is a human process, and as such it does not follow a strict path. We are social animals; we communicate best with people that share our language, background, and view of the word; with people from our community.

Communities are a powerful force, and the thousands of ongoing open-source projects are living proof. The Linux kernel project, with more than 15 thousand developers, is a striking example. The downside is that the stronger is the sense of belonging to a group, the harder it gets to communicate and understand "outsiders" [87].

It is easy to see how this type of tribalism influences modern politics, where increasingly polarized factions are unable to agree on any topic, whether it is dealing with global warming, emigration, economy or sovereignty.

The situation in computer science is not nearly as grim, yet the acronym NIH - Not Invented Here - [10] is well known by anybody working in the sector. The "*NIH syndrome*," also known as "*the tendency of reinventing the wheel*," is something we all have experienced, if not perpetuated, in our daily work.

While continuously re-implementing similar software can eventually (or casually) improve the technology stack, it is arguably an inefficient way.

The focus of our work in the Barcelona Supercomputing Center (BSC) is data management and distributed computing. We looked into different technological communities and studied how similar problems have been solved in various fields. We tested, analyzed and adapted "foreign" technologies for the HPC scientific research. As a positive collateral effect, we have also brought HPC know-how to other sectors, improving and extending the capability of a widely used database.

In the middle of the 2000s, the rise of the "**Social Web**" emphasized the importance of user-generated content and interaction. **Web 2.0** companies had to adapt to deal with a much larger flux of data, but existing solutions were unfit to the job.
In particular, Relational DataBase Management Systems took the hit. RDBMS, while rich in features, they lacked the speed and the horizontal scalability capabilities required. Indeed, business-oriented features such as relational integrity constraints, strong transactional consistency, and flexible declarative query language, which are

undoubtedly nice to have, are also complex and expensive to implement in a total peer-to-peer fashion and thus reduce the capability of such systems to linearly increase their performance by simply adding more computers.

The outcome was that a new generation of databases with reduced functionalities, but improved performance and scalability started to be popular. The new and extremely heterogeneous data storage systems were named with the buzzword NoSQL. Eased from SQL architectural constraints, NoSQL databases were put to deal with increasingly larger data sets. For example, in 2014 Apple reported using the NoSQL database Apache Cassandra to manage data in the order of the tens of petabytes with millions of concurrent transactions. Such increase in performance opened new applications in sectors where databases were historically considered too slow to be used. In our case, it made it possible to use a database to store the results of HPC scientific applications.

Dealing with storage, the HPC and NoSQL communities may share workloads of comparable orders of magnitude, but they tend to have different priorities and focuses. For NoSQL, the principal goal is availability, which means that a system must always work, despite hardware or software faults as the slightest downtime could lead to millions of euro of lost revenue.
Diversely, in HPC the focus is more on the bare performance, trying to push the hardware to the boundaries, even if it means sacrificing functionalities or system availability. In other words, any internet company that uses NoSQL is generally willing to sacrifice the performance per machine, or performance per dollar, aiming to more scalable and stable systems. Instead, in research we try to squeeze all the computation we can get for the same budget, with less care if the system has some downtime or requires low-level programming skills. Generalizing, on the one hand, we see fleets of thousands of commodity hardware servers, while on the other fewer high-end machines.

Therefore, it is not surprising that the two sectors have developed differently, focusing on what is considered more important. In NoSQL, we have databases of thousands of nodes distributed in different countries and continents working together, while in HPC we have master-slave file systems integrated into the high-speed network with RDMA capabilities.

Nevertheless, with data centers estimated to have consumed 1.3% of world energy in 2010 [66] and expected to rise to 20% in 2025 [9], it is imperative to increase computing energy efficiency, which also means bringing HPC know-how to the whole ICT industry.

In the meantime, as HPC simulations grow larger, scaling up to thousands of cores, there is an emerging need of fault-tolerance and highly-available architectures, as with these numbers the likelihood that one or more nodes fail during an executions increases. Also, in our experience, developers tend to re-implement existing solutions or use inefficient architectures when dealing with low-level interfaces.

Such a tendency, as we will discuss later in this thesis, leads to poor performance that can nullify the advantage of high-end HPC hardware.

In this thesis, we will analyze which features of existing key-value database technology can be beneficial to HPC, what is missing, and how it is possible to improve the working methodology of researchers. At first, we will describe the migration of three scientific applications, ranging from molecular dynamics, neuroscience and physics simulations to NoSQL solutions, and we study how the way we store the data influences the performance and scalability of the system. Then, we analyze which desirable features are missing, and we focus on two which we find extremely important but somehow at times underestimated, and that significantly improve the way scientific data is analyzed when combined.

Scientific simulations generate complex data sets where multiple correlated characteristics describe each item. For instance, a particle might have a space position (x,y,z) at a given time (t). If we want to find all elements within the same area and period, we either have to scan the whole dataset, or we must organize the data so that all items in the same space and time are stored together. The second approach is called Multidimensional Indexing (MI), and it uses different techniques to cluster and to organize similar data together.

While both ICT and HPC widely use single-dimension indexing on large data sets, MI is different because multidimensional points lack of an intrinsic natural order. While it is easy to determine that 3 comes before 5, we cannot say that the 2D point (1, 3) comes before point (2,2). One might be closer to a specific location, like point (0,0), but there is not a universal order, and therefore all indexing techniques which rely on ordering data cannot be directly applied. An alternative approach is to reduce the granularity of the dimensions and combine them in a unique, distinct value. This approach is used in many databases and indexing systems, but it only works well when the distribution of the value is mostly uniform and does not change in time. For instance, we could split a map of a city into quadrants of one km squared size, and create a file for each quadrant containing the name of the restaurants and shops in that area. Some files will be probably larger than others, but no file will grow up to the point it is unmanageable. Now, let us suppose that we use the same approach to track the position of people in the city. In this case, we will see that some files, the one coinciding with stadiums, concert halls, and shopping malls will be much larger than others. Even worse, such distribution is likely to change over time, or between day and night. To overcome these limitations, Multidimensional Indexes take care of adapting the way data is partitioned following the data distribution, even when it changes over time.

Many authors [37] [33] [101] pointed out how multidimensional and spatial big data are going to be an essential part of future scientific and business applications. In particular, Eldawy et al. [37] describe how we are entering the "*Era of Big Spatial Data*", with space telescopes generating up to 150 GB of spatial data per week [61],

medical devices producing spatial images at a rate of 50 PB per year and social networks that are managing billions of geotagged events per day. However, the lack of specialized frameworks to deal with such kind of information is limiting possible application and probably precluding many scientific breakthroughs. For instance, Simion et al. [100] discussed in their work *"The Price of Generality in Spatial Indexing"* how re-using existing solutions for one-dimensional indexing in spatial applications leads to sub-optimal performance in PostgreSQL. Kornacker et al. [67] reached a similar conclusion analyzing the use of generalized indexes in DB2/Common Server. Again, Eldawhy and Mokbel[37] elaborated a comprehensive survey of the existing solutions for big spatial data, and they described all existing approaches and their relative limitations. In particular, they showed that few solutions target dynamic indexing, and they only work for small point queries. Finally, they conclude that there are several open research problems that must still be addressed in this area.

On the other side, approximate analytics has been often indicated [48][4] as a smart and flexible way to interactively explore large data sets in a short period, as it allows to test and try different hypothesizes rapidly. Approximate analytics includes a broad family of algorithms which aim to speed up analytical workloads by relaxing the precision of the results within a specific interval of confidence. For instance, if we want to know the average age in a group with $\pm$ 1-year precision, we can consider just a random fraction of all the people, thus reducing the amount of calculation. But if we also want less I/O operations, we need *efficient data sampling*, which means organizing data in a way that we do not need to scan the whole data set to generate a random sample of it.

To our knowledge, there are no existing solutions that support scalable Multi-dimensional Indexing combined with efficient data Sampling (MIS), a feature that we consider fundamental when dealing with large data sets, as it enables more agile data pipelines, and new types of interactive analysis and data exploration.

These two features when combined enable interactive exploration of massive simulations, allowing scientists to explore the outcomes of their experiment with the desired level of detail and to run distributed analysis with either arbitrary precision or precise timing deadlines.

For these reasons, this Ph.D. thesis focuses on the incremental development of novel distributed MIS solutions that can be applied both to HPC and data analytics workloads.

The research described in this thesis brought to the invention of 2 novel MIS algorithms and the development of a peer-to-peer distributed indexing architecture that we think is going to influence the architecture of the future data storage systems and it will simplify the way HPC simulations are managed.

1.1 Contributions

This thesis will prove that alternative storage solutions based on NoSQL technologies are a viable and convenient option when dealing with scientific simulations as they improve performance, speed up the research workflow and increase user productivity.

Additionally, we will describe how we extended the capabilities of NoSQL technologies proposing Qbeast; a novel distributed system for multidimensional indexes with arbitrary approximated precision that we will prove to be a viable solution to store, visualize, explore and analyze large scientific simulations. Qbeast includes two major contributions, the D8tree, and the AOTree, two novel indexing algorithms that target different application scenarios.

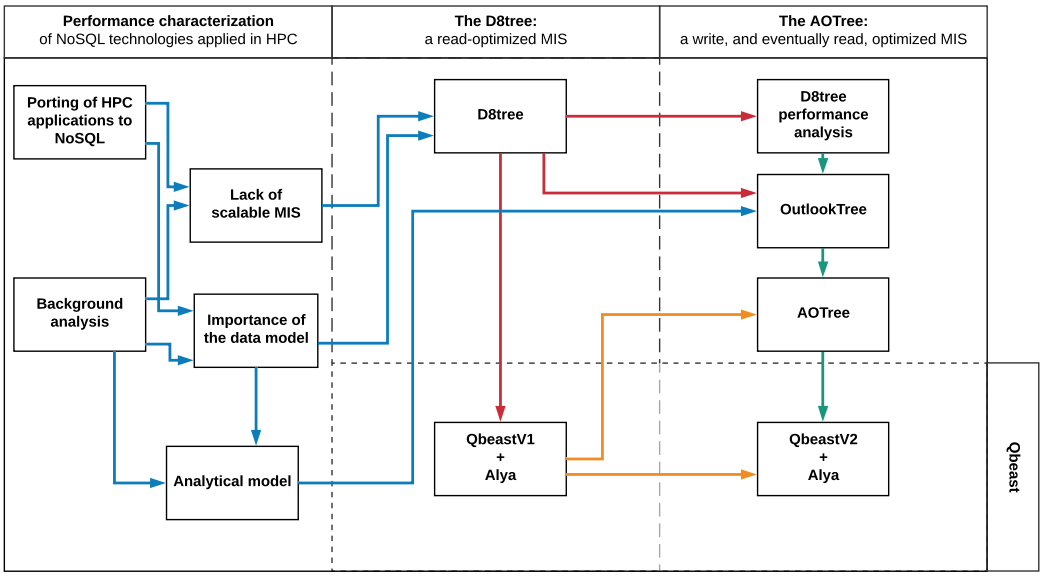


FIGURE 1.1: Integration between the four contributions of this thesis.

This document describes in detail the research behind our work, the development journey we followed and the results we achieved. Figure 1.1 shows the incremental steps we have taken and the path we followed working on this thesis. The Figure also shows how the different incremental steps can be organized into four main contributions. More precisely:

1.1.1 Performance characterization of NoSQL technologies applied in HPC

In the **first contribution**, we will present a broad background analysis of existing storage technologies with a review of their benefits, criticalities and missing functionalities for scientific applications. Then, we will motivate our preference for NoSQL key-value databases and we will analyze three HPC applications that we have improved using this technology. We learned from these use cases two vital

lessons. Firstly, how important the data model is in achieving performance and scalability in distributed applications. Secondly, that multidimensional indexing with efficient data sampling is a critical missing feature in NoSQL technology that can greatly benefit scientific research. These lessons will motivate the creation of our first MIS algorithm, the **D8tree**.

Later, we will propose an **analytical model** that estimates the performance of a distributed key-value database by taking into consideration the relationship between the data model - more precisely the cardinality of the key -, the size of a data element, and the number of servers. This model will help to overcome some of the limits of the D8tree, and it will drive the design of a new write-optimized MIS algorithm, the **AOTree**.

The work performed in this area produced the following publications:

[32] Cugnasco, C., Becerra, Y., Torres, J., & Ayguadé, E. (2017, August). Exploiting key-value data stores scalability for HPC. In Parallel Processing Workshops (ICPPW), 2017 46th International Conference on (pp. 85-94). IEEE.

[30] Cugnasco, C., Hernandez, R., Becerra Fontal, Y., Torres Viñals, J., & Ayguadé Parra, E. (2013). Aeneas: A tool to enable applications to effectively use non-relational databases. In Procedia computer science, Vol. 18, 2013 (pp. 2561-2564). Elsevier.

and it has been motivated by the following publications:

[53] Hernandez, R., Cugnasco, C., Becerra, Y., Torres, J., & Ayguadé, E. (2015, March). Experiences of using Cassandra for molecular dynamics simulations. In Parallel, Distributed and Network-Based Processing (PDP), 2015 23rd Euromicro International Conference on (pp. 288-295). IEEE. ^{oo}

[55] Hospital, A., Andrio, P., Cugnasco, C., Codo, L., Becerra, Y., Dans, P. D., ... & Gelpí, J. L. (2015). BIGNASim: a NoSQL database structure and analysis portal for nucleic acids simulation data. Nucleic acids research, 44(D1), D272-D278.

1.1.2 The D8tree: a read-optimized MIS

The **second contribution** is the development of the **D8tree**, a fully peer-to-peer read-optimized MIS that builds a de-normalized Octa tree index with stratified replication on top of key-value databases. The system takes advantage of the high write throughput of the underlying key-value database to build a read-optimized structure that can efficiently run MIS queries. Also, we will describe the design of the first version of Qbeast, the distributed system that produces the D8tree in real-time. Then, we will analyze the performance and shortcomings of the integration of the D8tree with a physics simulation run with Alya; a simulation code for high-performance computational mechanics developed at the BSC.

Finally, we will describe how our solution significantly speeds up and improves the daily workflow of scientists by enabling early access to the results and simulation steering.

The work performed in this area produced the following publications:

[31]Cugnasco, C., Becerra, Y., Torres, J., & Ayguadé, E. (2016, January). D8-tree: A de-normalized approach for multidimensional data analysis on key-value databases. In Proceedings of the 17th International Conference on Distributed Computing and Networking (p. 18). ACM.

[13] Artigues, A., Cugnasco, C., Becerra, Y., Cucchietti, F., Houzeaux, G., Vazquez, M., ... & Labarta, J. (2017). ParaView+ Alya+ D8tree: Integrating High Performance Computing and High Performance Data Analytics. Procedia Computer Science, 108, 465-474.

1.1.3 The AOTree: a write, and eventually read, optimized MIS

The **third contribution** of this thesis is the development of a new indexing algorithm that overcomes the shortcomings of the D8tree delivering higher writing throughput while eventually guaranteeing its same query performance. By studying the performance of the D8tree and using the analytical model to analyze the limitation of possible applications, we will propose the **OutlookTree**, an index that reduces the storage and transactional overhead of the D8tree by decreasing the data replication without compromising query performance. Then, we will present the Asymptotic Outlook Tree, **AOTree**, a patent-pending write and eventually read optimized indexing system. The AOTree uses opportunistic data distribution and optimization to build a write-optimized index that ultimately evolves into an OutlookTree, thus improving query performance even in write-intensive HPC environments.

As part of this contribution, we will compare Qbeast's AOTree implementation with PostgreSQL and with file storage on GPFS. Furthermore, we will describe how we achieved good performance integrating an MPI based application with our system which uses TCP/IP.

The work performed in this area resulted in the following publications:

In preparation: Cugnasco, C., Calmet, H., Santamaria, P., Gil, E., Sirvent, R., Becerra, Y., Torres, J., Ayguadé, E., Labarta, J. (2019). Qbeast, the HPC multidimensional database. To be submitted to ICPP2019

European patent request EP18382698.1, with title DISTRIBUTED INDEXES, applicant BARCELONA SUPERCOMPUTING CENTER - CENTRO NACIONAL DE SUPERCOMPUTACIÓN, and author Cesare Cugnasco and Yolanda Becerra.

1.1.4 Qbeast

The **fourth contribution** is Qbeast, the system that distributively implements the D8tree and AOTree. Indeed, as existing HPC problems motivate our research, we had to find a way to test our algorithms and architectures when applied in a realistic application scenario, thus verifying that any theoretical advances would result in real benefits. To test these assumptions, we can either build a prototype database from scratch, or integrate our software into an existing database. With the first approach, we implement a strip-down mock system with only a few functions available. A mock system has the advantage that it is simpler to understand what influences the overall performance, as there are fewer components to interfere, but the results are less significant. Indeed, a real-world application would require the missing features, thus potentially changing the final system behavior. For this reason, we preferred the second approach, thus we implemented a system that reuses part of the code and architecture of Apache Cassandra, a widely adopted and open source NoSQL database.

FIGURE 1.2: The Qbeast logo, by the designer Daniele Ramancin.

We named the new software Qbeast 1.2 (read as [ˈkjuːbiːst]), the "cubistic beast.", as an homage to the cubism art movement pioneered by Pablo Picasso that revolutionized the way we can represent a 3-dimensional world into a 2D canvas. Similarly, Qbeast aims to innovate the way multi-dimensional information is organized into linear storage.

1.2 Thesis structure

The thesis is structured as follows: Chapter 2 is our first contribution, and it contains an analysis of the data management requirements of three HPC applications and the description of the available solutions. This chapter also motivates our preference for key-value databases, and we propose an analytical model that explains the relationship between the data model and scalability. Chapter 3 describes our second contribution, the D8tree algorithm, and it examines its performance advantages and drawbacks when indexing static data sets or when managing the I/O of HPC applications. As our third contribution, Chapter 4 describes the path toward a write-optimized MIS index by illustrating the drawbacks of the D8tree, defining the OutookTree, and finally, proposing the AOtree that represents our final solution. The chapter also contains a performance comparison of the AOtree with GPFS and PostgreSQL. The implementation details and the architectural trade-off of Qbeast are described in Chapter 5 as the fourth and last contribution of this thesis.

Finally, in Chapter 6 we will draw our conclusions, and we will discuss possible future works.

Chapter 2

Performance characterization of NoSQL technologies applied in HPC

The first contribution of this thesis is a study of the data management requirements of HPC applications and it contains a description of the existing storage solutions. Also, this Chapter includes an analysis and modelization of the relationship between the data model and the performance of key-value databases when used for HPC applications. Firstly, we will analyze the data management requirements of three scientific use cases on which we have been working on in the last few years, focusing on their criticalities, missing functionalities and areas of improvement of the current solutions.

Secondly, we give a broad description of all the available distributed data storage solutions, and we will provide the reader with the required background information to distinguish between different architectural choices. Thirdly, we will motivate our choice to use key-value databases as a storage solution in HPC, and we will discuss the importance of the data model in achieving high performance and scalability. Lastly, we will describe an analytical model that we have developed to predict and study the influence of the data model in key-value databases and that we will use to design novel MIS algorithms.

2.1 Target scientific HPC applications

A large part of the work our research group has been doing in these years at the Barcelona Supercomputing Center has been adapting and studying the performance and the usability of NoSQL technology in HPC. One of the main problems we had to face was overcoming the natural resistance to change. No matter how inconvenient a legacy method or procedure is, users tend to oppose learning new ways to do things, even if it simplifies its daily workflow. Having this in mind, our main focus has been since the beginning to provide a familiar interface to users, requiring minimum changes in their way to work. However, unlike SQL databases, in NoSQL databases the way you organize data dictates which kind of query can or cannot be done, or

which ones will be fast or extremely slow. Therefore, our work always aimed to find the right trade-off between having a more performant data model or preserving interoperability with existing legacy code. Eventually, the experience we maturated drove our research toward a scalable multidimensional index.

In this section, we will provide a brief description of the work we have done over three main HPC applications coming from different scientific areas: life science, applied medicine, and neuroscience.

2.1.1 BigNASIM

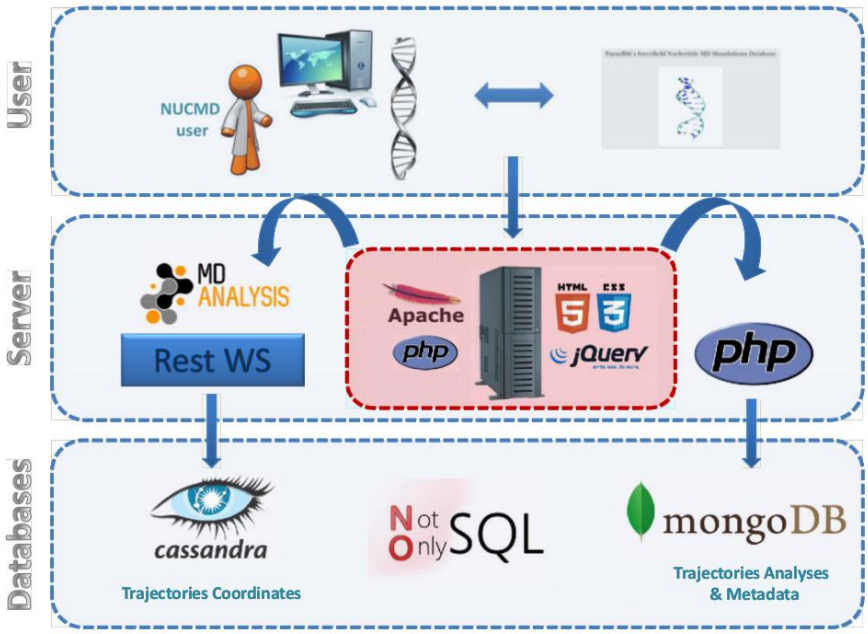


FIGURE 2.1: The BigNASIM architecture

Our first collaboration, BigNASIM [55] focused on building a NoSQL based service containing the structures that allow the analysis for nucleic acids simulation data. The service is online and accessible from `http://mmb.irbbarcelona.org/BIGNASim/`. Molecular dynamics - MD- is one of the bioinformatics tools that generate the largest amount of data and that consumes the highest volume of CPU resources in supercomputing centers. MD focuses on studying the physical movements of atoms and molecules. The simulations usually run for a pre-determined amount of time and the resulting data, composed by the molecules' positions, as well as their physical properties, are periodically stored in equally distanced timestamps. After almost 40 years since the first biomolecular simulation, MD has reached the maturity to tackle the simulation of macromolecules, on time spans that can go from the multi-nanoseconds to even microsecond scales. As a byproduct, huge trajectory files are stored for further analysis. However, it is common practice that such

simulations are often ignored and forgotten after a rather superficial analysis. The goal of BigNASIM is to create an open tool where researchers can upload their MD trajectory, so that their work can be reused in different studies, potentially saving thousands of computing hours, money and speeding up the research process.

The solution we adopted used a mixed approach, trying to take advantage of two different NoSQL databases: Cassandra and MongoDB. MongoDB is used to store all the metadata related with a simulation, while Cassandra stores the actual trajectory data. We used two separate systems as Cassandra performs better for large datasets with a structured form, while MongoDB is more flexible storing the differently structured metadata information that different providers use. Our work focused on Cassandra, with two main contributions:

1. Adapting a popularly used MD analysis python library, MDanalysis and MDplus to manage seamlessly trajectory stored in Cassandra.

2. Developing a custom trajectory bulk loader to speed up the loading of large trajectories on Cassandra.

Cassandra adopts a DHT architecture, and thus the popularity and the cardinality of the primary keys strongly influence performance.

LISTING 2.1: The BigNASIM trajectory data model

```
CREATE TABLE topology(
atom_num int PARTITION KEY,
atom_name text,
atom_type int,
chain_code int,
residue_code int,
residue_num int
);
CREATE TABLE trajectory(
frame int,
atom_id text,
x double,
y double,
z double,
PARTITION KEY(frame,atom_id)
);
```

Listing 2.1 describes the data model we used to store the trajectory data. Topology holds the description of the molecular system using atom numbers as the main indexing key, and storing the atom details, and the usual logical ways of grouping them (residue, chain). The Trajectory table stores the coordinates indexed using frame and atom numbers.

In this situation, the decision regarding the data model was mainly driven by the need to minimize the change in the existing legacy library. Indeed, by defining the frame number as partition key, we ensure that all the atomic coordinates at a given snapshot are stored contingently in the same node. Additionally, each frame has atomic identifiers as a second level index, allowing efficient access to any subset of atoms. We have chosen to prioritize frame based access, after analysing the accessing patterns of the MDAnalysis software used to handle trajectory data. Indeed, we were constrained by its interface, which always accesses to trajectories one frame at a time. Consequently, with our model, the existing algorithms manage a trajectory in Cassandra seamlessly as if it was a common file. At the same time, algorithms that require data of only a subset of atoms may be optimized to take advantage of the second level indexing. To move trajectory data in and out of the Cassandra subsystem, the use of the Python package MDPlus assures full compatibility with existing molecular dynamics software. Still, when dealing with massive bulk data loading into the database, the overhead introduced by the network communications and the data marshalling between different platforms can be a problem. For this reason, we developed a utility program that takes as input a trajectory file and converts it directly into SSTables files, the Cassandra internal binary data format.

2.1.2 Alya

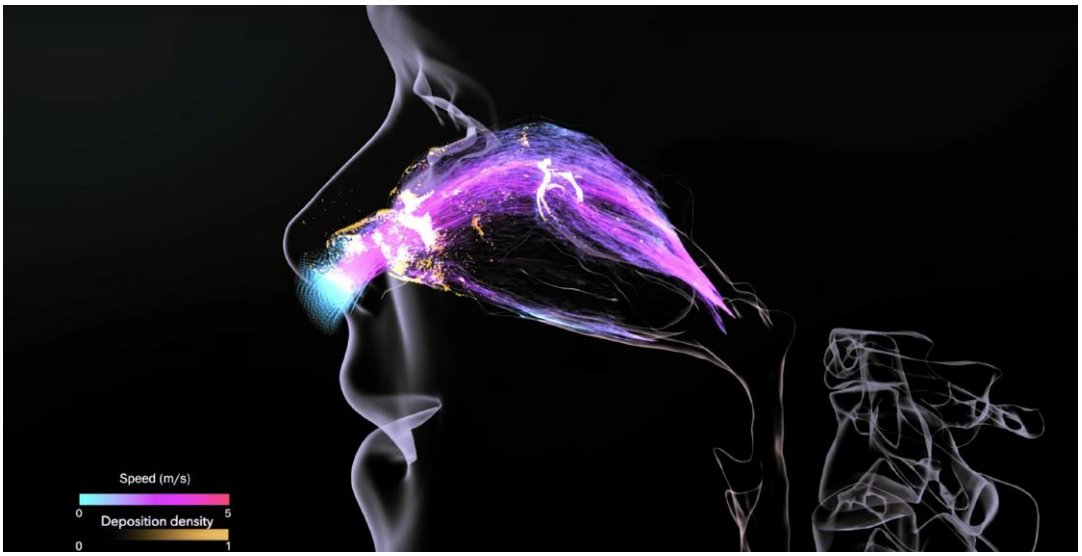


FIGURE 2.2: A render of the physical problem

Alya is a BSC in-house HPC-based multi-physics simulation code designed to simulate highly complex problems and efficiently run on high-end supercomputers. Alya is developed in the CASE department of BSC. In a previous work, Artigues et al. [14] made a first attempt to couple large simulations with a system that was able to visualize them and run custom queries. They used a brute force approach, using Hadoop to store the results and Impala to run custom queries on the data. hen, they generated a binary file in VTK file format, a popular library for manipulating

and displaying scientific data files. Then they used ParaView, an open-source data analysis and visualization application, to open and visualize these files. The downside of such an approach is that the simplest operation, such as the visualization of a sample of a simulation, requires first to load in memory all the data, then a random selection of the elements, writing the VTK file to disk and then opening the data in ParaView. On other words, if a scientist only needs to see a fraction of the simulation, the whole data needs to be loaded and filtered. A pivotal point in our research was realizing that this approach is neither scalable nor efficient and so we had to look for a smarter solution. That lead first to the D8tree and finally to the AOtree that are better described in Chapters 3 and 4. The previous workflow pipeline required firstly to run the whole simulation in a supercomputing facility. Secondly, once the simulation completed, copy the data to the Hadoop cluster, then run the query, copy the VTK in the local computer and finally visualizing it with ParaView. With our solution, Alya is directly integrated with Qbeast, our indexing system. Indeed, we provide the integration with Slurm, an open-source job scheduler used by many of the world's supercomputers and computer clusters. In such a way, the HPC code can be co-allocated with a Qbeast cluster of arbitrary size. In any moment during the simulation, the scientist can open in its desktop computer ParaView with our plugin (Qview, described in Section 5.5) and interactively visualize the data.

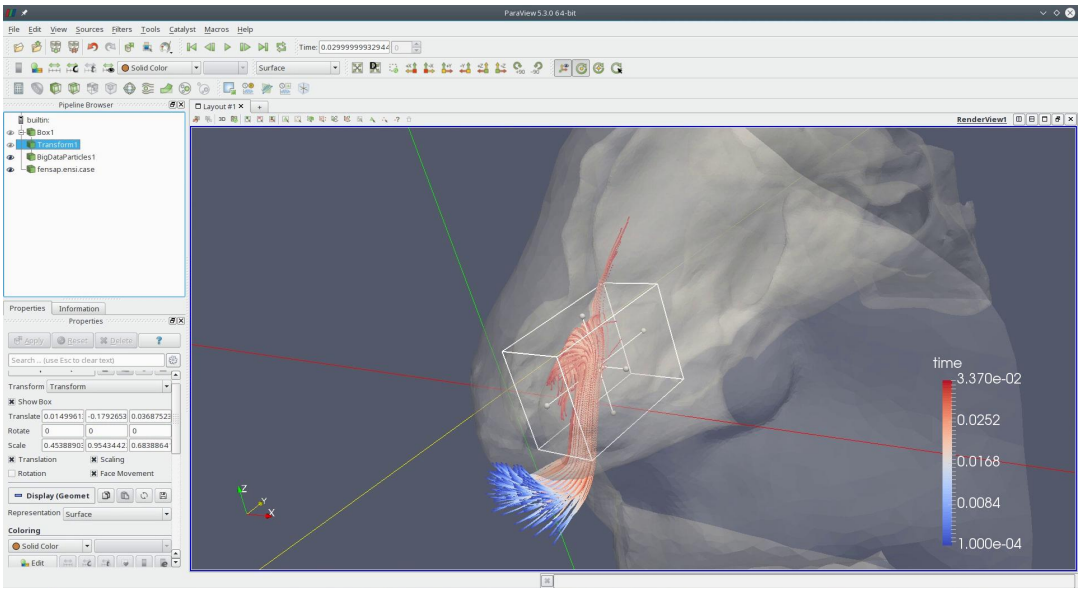


FIGURE 2.3: A snapshot of Qview, our ParaView plugin

Figure 2.2 shows a snapshot taken from a video render generated from a simulation of the physical problem. Render videos ensure stunning visualization, but they lack interactivity. The render is produced only once the simulation has completed, and the camera shows a pre-determined path and zoom. To change the perspective or the focus you need to re-run the render, which is an operation that can take hours even with GPU optimized machines. Figure 2.3 shows our approach with Qbeast.

In this case, we can access the data while the simulation is running and decide to visualize only a fraction of the data so that we can interactively explore the results. In this snapshot specifically, we are visualizing the trajectory of the particles that have flown by a specific area that we have selected from ParaView, while the code to filter and select the particle was written with Hecuba. The image comes from a demo that has been presented at the ISC and SC in 2017.

2.1.3 Cell Data

The third and last HPC application we will consider comes from the neuroscientist community. In the context of the Human Brain Project, one of the two European flagship projects, we have collaborated in studying the I/O bottleneck of different applications that are used in the HBP community. Above all, we will focus on the problem of cell data segmentation, that describes the tasks of recognizing brain cells in high-resolution microscopical images. Each image has a size in the order or the tens of GB, while a full brain scan can be several terabytes.

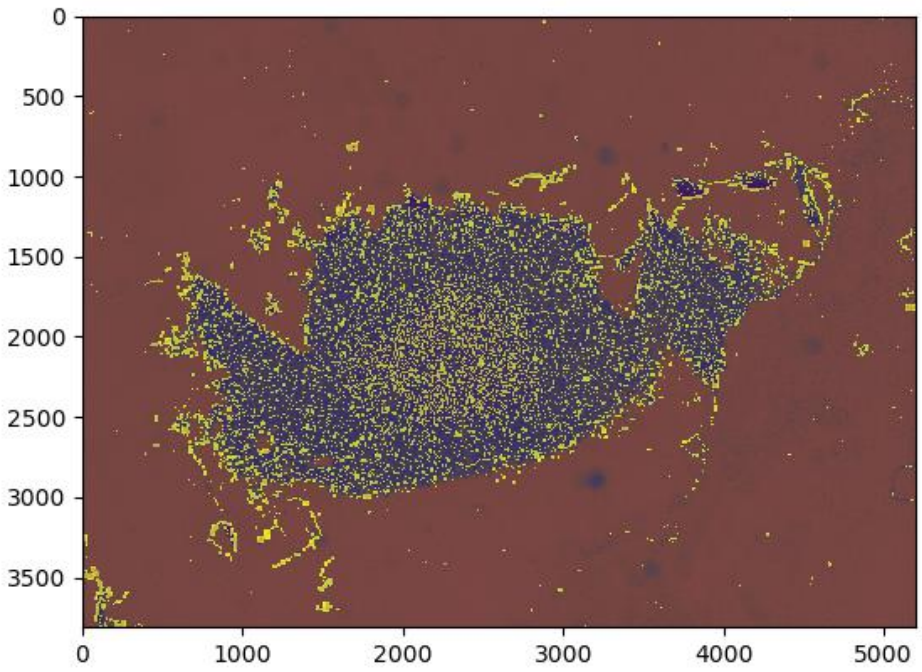


FIGURE 2.4: An example of cell segmentation

Figure 2.4 shows an example of cell segmentation. Each image is part of a horizontal section of the brain. The yellow lines represent the cell borders that have been detected. For each cell, we have to find the position in the image - and therefore in the space - and additional physical properties such as maximum width and height. In the first implementation, we analyzed, each image was divided into equally sized parts and the work distributed among MPI workers that analyze a disjoint subset of

them. In the end, each worker negotiates with the master node a unique ID for each found cell. Then, the workers generate a position matrix of the size of their input portion, mapping each image pixel to a number which either indicates the absence of a cell - 0 - or identifies the cell present in that position. Finally, the master collects all matrixes, merges them and stores the result in an HDF5 file, a commonly used binary format for multidimensional arrays.

This approach had two main drawbacks. Firstly, the nodes could only synchronize at the end of each part of the work, which led to suboptimal performance because areas with a higher concentration of cells tend to be slower to analyze. Secondly, the master limited the scalability as the amount of data we needed to store to record the cell position was approximately the same size of the input data, and moreover, it had to be transmitted to the master to be reorganized and persisted. We improved the application using PyCOMPSs, a framework for distributing computing workflows; Hecuba, a library for distributed data management; and Qbeast, our implementation of a distributed MIS system. These technologies are better described in Chapter 5. By using Hecuba to store data, the workers do not need to synchronize between them before persisting data as anyone can do I/O individually. Once the synchronization was removed, using PyCOMPSs instead of MPI, we allowed the scheduler to reorganize the execution of the tasks so that if a job took longer to compete, the others could proceed in parallel with no stall.

Finally, Qbeast allowed avoiding to create and store the position matrix, as it can index more efficiently each cell in the space. In some situations, using Qbeast instead of the position matrix in HDF5 meant shrinking the space requirements from approximately 40GB to 600MB for each image.

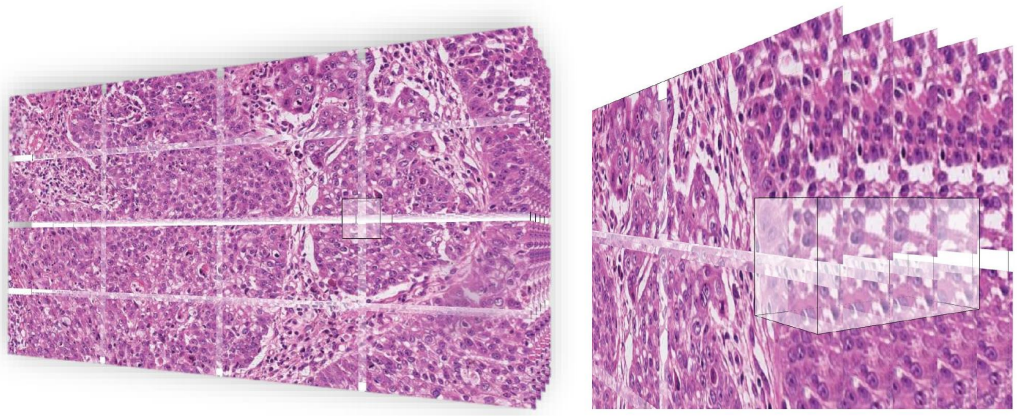


FIGURE 2.5: An example of how the brain images are partitioned

Figure 2.5 shows another application in which the indexing capability of Qbeast can be useful. On the left side, we can see how a plane of a brain scan is partitioned in segments so that the application can process each segment independently. However,

if a cell falls in between two partitions, it will be counted twice, and it will report the wrong physical dimensions. The highlighted box in Figure 2.5 shows an example of an area that we need to post-process to reconcile all intra-border cells. At the same time, a similar process of cell reconciliation is also required when merging a cell that spans on multiples plans, so that we can generate a more consistent atlas of the brain. By using Qbeast, it is possible to reduce the I/O requirements of such post-processing operations, as long as we have to load in memory only the cells that fall into the bordering areas.

2.1.4 Missing functionalities and common aspects

The three applications we have discussed have different requirements and are an excellent example of which kind of functionalities scientific use cases can require. For example, in the case of BIGNASim, the focus is on providing a storage backend that can seamlessly integrate with the existing legacy code without major changes. Furthermore, as BIGNASim is a service accessible by the whole scientific community, the data storage must be both highly available and able to horizontally scale to cope with the increasing number of users. Also, as the simulations' size can go up to hundreds of gigabytes, it is essential to reduce the distance between users and the datacenters, which is possible using multi-datacenter geographical replication. The second use case, Alya, underlines the importance of being able to visualize uniform random samples of the results while the experiment is still running without penalizing the overall simulation speed. In the cell data application, the focus is on the integration of the analytical code with the data storage systems, where it is paramount to partition and split large datasets by their space distribution thus improving the performance of space-correlated machine learning algorithms.

All three cases also have many similarities. For example, in all cases, the data is never deleted, nor updated and only new elements are added to the simulation. As we will discuss later, such usage pattern relaxes the requirements that a storage system must support, as long as many consistency issues cannot rise.

Furthermore, all three cases simulate real-world 3D physical problems thus justifying our advocacy for better multidimensional support in scientific storage systems. In particular, the Cell Data and the Alya use cases also show the need of efficient uniform data sampling on specific regions of the space. In the first case, to improve machine learning algorithms execution, while in the second to speed-up interactive visualization. In both cases, these functionalities can be supported by a multidimensional index with efficient data sampling support (MIS).

In the following sections, we will discuss the available data storage solutions, and we will provide the necessary background information required to understand the difference and characteristics of the different systems. Finally, we will motivate our preference for key-value databases, and we will discuss the importance of the data model in such systems. This study will then guide our design for two novel distributed MIS algorithms.

2.2 Background

In this Section, we will try to give a brief but broad view of the available solutions that deal with distributed management of data. We will highlight the most popular tendencies and the shared aspects that can be found in software alternatives from different sectors. We hope the reader will forgive us if we omit some relevant details for the sake of a more general abstraction.

In this effort, we will point out what accumulates heterogenous systems such as distributed parallel filesystems, whether POSIX compliant or not; SQL, NoSQL, NewSQL databases; and object stores.

At first, we shall start discussing the scope these systems aim to tackle. Databases traditionally fall in two broad categories; OTPL and OLAP. The first, Online Transaction Processing systems (OTPL) focus on relieving applications from dealing with concurrent reads and writes. Different clients can modify the same data without having to care about consistency. The typical workload comprises a high number of simultaneous operations that affect a limited number of elements. The canonical example is a bank with clients withdrawing money from their accounts at the same time. On the other hand, OnLine Analytical Processing systems are more suited for reporting, data mining, and forecasting. These types of workloads typically require reading and aggregating large sets of information. Therefore OLAP databases are mostly optimized for complex queries rather than concurrency. In other words, few users doing long running read-mostly queries.

In a typical scenario, the data is copied periodically from an OTPL database into the OLAP one with a process named Extraction Transformation Loading. The ETL process adds a delay between the time new business events occur and when their effect appears in reports and analytics. In numerous situations, as the value of information is said to decrease with time, a lag ranging from minutes to hours is unbearable. Hence, a relatively new tendency is Hybrid Transactional-analytical processing (HTAP)[45], which aims to unite real-time transaction and historical data in a unique system. As described by Andrew Pavlo et co.[83], there are generally three types of approaches to implement HTAP pipelines in a database application. The first and straightforward solution requires deploying two different database clusters, one serving the transactional workload while the second serving the analytical queries. In such a scenario, each update gets propagated to both clusters. Periodically, the application executes the required complex queries on the analytical cluster and then pushes the results into the front-end database. In such a way, the analytical workload does not interfere with the transactional one. As a drawback, we have two systems to maintain and a considerable latency between an event and the moment we can analyze it.

A somehow similar approach is the design known as lambda architecture[75] that combines three layers, a batch one for historical data, a speed/streaming one to provide views on incoming data, and a serving one that merges the two and delivers

the final information to the users. To provide a simple example, let's imagine we want to count the page visits of a website. With the streaming architecture we would probably use a framework such as Apache Hadoop[97] or Apache Spark[118] to analyze hourly the logs and to compute a view reporting the number of visits for each page at the time the batch runs. On the other hand, we would have a streaming framework, such as Apache Storm[108] or Apache Spark Streaming[116], to keep an approximation of the count of visits. The serving layer will then act as a glue of the two systems, merging the results and presenting it to the final users.

The problem of both approaches is that we must maintain two different systems and sustain the overhead of copying the data several times across the various components. The additional complexity also has a financial impact, as administrative cost and developing overhead increases the overall personnel cost. Indeed for large-scale database systems this cost is estimated to be 50% of the total cost of ownership[92].

A third and more desired approach is building a unique system capable of supporting analytical workload without degrading the speed of transactions. Many databases have started to move in such a direction. A typical approach is to use two different database engines and scheduling for the two types of workloads, this way they do not interfere. This solution is popular both in NewSQL databases, such as SAP HANA [99] and MemSQL [77], and NoSQL ones like Datastax Enterprise Cassandra and ScyllaDB [95]. Yet, it is still open for discussion if a "one-size-fits-all" approach is the best one [103].

The reader may wonder what these classifications, OLTP, OLAP, and HTAP have to do with parallel filesystems and object storages as they do not provide transactions or analytics functionalities. On the other hand, HPC applications that store data in files have to deal with concurrent writes and reads from different nodes and workers. Also, once the simulation completes, scientists often run additional analysis on the results, and that usually requires a change in structure and reorganizing the data in a process that is similar to the previously cited ETL. Then, scientists must write the code to run the analysis distributedly, building a system similar to the OLAP ones.

From our point of view, if not mere filesystems or object storages, the overall stack that HPC scientists use to support their pipeline can be classified, or at least share many characteristics, with OLTP or OLAP systems. For example, if many workers compete for the same file, a library such as MPI I/O is generally used to buffer writes and coordinate MPI workers. Alternatively, if each worker generates different files, a second phase is required to merge and transform the data. In section 2.5 we will elaborate further on this topic.

2.2.1 Common grounds

The goal of this section is to build a common ground of terminology, algorithms and common architectures that we can find in most of the distributed systems that

manage data.

Such a broad and detailed description will then help us to understand the architectural choices that we have taken in the design of Qbeast. More precisely, it will lay down a vocabolary that will help us to understand how our solution differ from the others and which kind of guarantees we can provide to the users.

More precisely, the algorithms, approaches, and architectures that I will list will touch in the following topics:

1. ACID
2. Concurrency control
3. Shared consensus
4. Durability
5. Data placement and Metadata management
6. Data model

2.2.2 Atomicy Consistency Isolation and Durability

Relational databases must ensure ACID guarantees to meet the standard. ACID is defined as:

Definition 2.2.1 ***Atomicy:***
An operation is either completely executed or completely aborted, by avoiding the existence of inconsistent information in case of a crash of the system.

A typical example is a transfer between two bank accounts: it is mandatory that the subtraction of the money from one account and the addition to the other are both successful or both aborted.

Definition 2.2.2 ***Consistency:***
The guarantee that operations in transactions are performed accurately, correctly, and with validity, concerning the application business logic.

Using the previous example, the system could avoid a transfer if the first bank account does not have enough money.

Definition 2.2.3 ***Isolation:***
It guarantees that two transactions, made at the same moment, are executed as if they were executed sequentially.

Definition 2.2.4 ***Durability:***
It means that after the committing of a transaction the data remains even in the event of a power loss or if a system crash occurs.

In the following sections I will describe how these properties are implemented in distributed databases.

2.2.3 Concurrency Control

With Concurrency Control we group all techniques that guarantee correct results - consistency - even when multiple operations run concurrently and while preserving speed. There are, however, different nuances regarding the meaning of consistency. Additional to the ACID definition 2.2.2, we can find:

Definition 2.2.5 *The guarantee that transactions see all the effects of transactions committed in the past.*

Definition 2.2.6 *The guarantee that database constraints, such as table relationship, uniqueness etc., are not violated.*

Definition 2.2.7 *The guarantee that a completed update is visible to all clients of a distributed database.*

In this discussion, we will focus only on the first and the last definitions of consistency.

The first definition 2.2.5 relates to the concept of Serializability, which is the property that the outcome of a concurrent transaction is the same outcome we would have if the transactions were processed serially, without overlap.

According to "A Critique of ANSI SQL Isolation Levels Hal"[17], the undesired phenomena that must be avoided to preserve serializability are:

Definition 2.2.8 *(P0) Dirty Write:*
Transaction T1 modifies a data item. Another transaction T2 then further modifies that data item before T1 performs a COMMIT or ROLLBACK. If T1 or T2 then performs a ROLLBACK, it is unclear what the correct data value should be.

Definition 2.2.9 *(P1) Dirty Reads:*
Transaction T1 modifies a data item. Another transaction T2 then reads that data item before T1 performs a COMMIT or ROLLBACK. If T1 then performs a ROLLBACK, T2 has read a data item that was never committed and so never really existed.

Definition 2.2.10 *(P2) Non-repeatable Reads:*
Transaction T1 reads a data item. Another transaction T2 then modifies or deletes that data item and commits. If T1 then attempts to reread the data item, it receives a modified value or discovers that the data item has been deleted.

Definition 2.2.11 *(P3) Phantoms:*
Transaction T1 reads a set of data items satisfying some <search condition>. Transaction T2 then creates data items that satisfy T1's <search condition> and commits. If T1 then repeats its read with the same <search condition>, it gets a set of data items different from the first read

Definition 2.2.12 *(P4) Lost Update:*
The lost update anomaly occurs when transaction T1 reads a data item, and then T2 updates the data item (possibly based on a previous read), then T1 (based on its earlier read value) updates the data item and commits.

Definition 2.2.13 *(A5A) Read Skew:*
Suppose transaction T1 reads x, and then a second transaction T2 updates x and y to new values and commits. If now T1 reads y, it may see an inconsistent state, and therefore produce an inconsistent state as output

Definition 2.2.14 *(A5B) Write Skew:*
Suppose T1 reads x and y, which are consistent with a particular constraint, and then a T2 reads x and y, writes x, and commits. Then T1 writes y. If there were a constraint between x and y, it might be violated.

The last two definitions were not originally included in the ANSI SQL standard, and they have been proposed by Berenson et al. [17] to address the shortcomings of MVCC.

The canonical way to ensure Serializability is to use the **two-phase locking** (2PL) protocol. The protocol states that a transaction should have two phases, an *Expanding/ Growing phase*, and the following *Shrinking/ Contracting* one. In the first, locks can be acquired but not released, while in the second, locks can be released but not acquired. The locks are either exclusive write-locks or shared read ones. The rule goes that if one transaction has a write-lock on a resource, the others cannot read or write it, while multiple operations can have simultaneous read locks on the same object. At the same time, if there is a read-lock on an item, it is impossible to acquire a write one.

The main problem of 2PL is the degradation in performance caused by lock contention. Also, as long as writes need to acquire an exclusive lock on a table, long-running reading queries block any updates. Indeed, in such a case an analytic query would likely acquire a read-lock on a whole table, stalling all concurrent writes for the duration of the first query.

An alternative solution is MultiVesion Concurrency Control (**MVCC**), which uses timestamps and incremental transaction ids to achieve transactional consistency with snapshot isolation (SI), which holds lower guarantees compared to Serializability. The main idea is that an update does not modify the old object but it creates a new version. In such a way a transaction T sees the database state as produced by all the transactions that committed before T started, but no effect from the overlapping ones. Thus, we avoid (P1) Dirty Reads 2.2.9, (P2) Non-repeatable Reads 2.2.11, (P3) Phantoms 2.2.11, without having to lock write or read operations.

To avoid (P4) Lost Update 2.2.12, the DBMS aborts a transaction T if a concurrent transaction committed an update of an item that T wants to change. This rule is often called "First-Committer-Wins". However, anomalies such as (A5B) Write Skew 2.2.14 [17] are possible.

An example of such an anomaly has been proposed by M.J.Cahill et al. [22]. Let's suppose we have a table *Duties* that keeps track of the working shifts of doctors in a hospital. A doctor can either be "*on duty*" or "*on reserve*" for a given shift, and there must always be a doctor on duty for each shift. A transaction that wants to put on reserve a doctor will first update their record, and then count the number of doctors on duty, and if there are none, abort the transaction. However, in such a scenario with the mere SI, if all doctors on duty try at the same time to switch on "reserve", they will be allowed to do so as they won't be able to see the conflicting updates.

While there are existing solutions for serializable isolation for MVCC, they either require to keep track of the entire read set of every transaction, with a massive overhead for read-heavy workloads [22], or work only for in-memory databases[79].

A final alternative approach is the mere timestamp ordering implemented by VoltDB[102], which splits the database in partitions and schedules transactions to execute one-at-a-time at each partition. Such an approach, however, has the downside that a transaction that touches multiple partitions slows down the whole system as nodes stay idle due to network latency [83].

The fourth definition of consistency 2.2.7 influences many aspects of the design of a distributed database, as it guarantees that a completed update is visible to all clients. In particular, it determines the overall throughput, latency, and stability of a distributed system. In general, there is not a "one-size-fits-all" solution, and the architecture of a distributed database must find the right trade-off between Availability and Consistency. Originally, such trade-off has been described by the CAP theorem introduced by Dr. Brewer in a keynote in 1999 and then formalized by Gilbert and Lynch [47] in 2002. The theorem states that in a web system there are three desirable properties, but it is possible to have at most two of them. These properties are:

Consistency
: Same as 2.2.7: all the clients read the same values at the same time from all the servers. When this is not possible, they receive an error.

Availability
: Every query receives a non-error response, but it might contain an obsolete value.

Partition tolerance
: Even though the network arbitrarily loses or delays messages, the system must continue to operate.

To ensure full ACID compliance, distributed RDBMS cannot sacrifice consistency and therefore, in case of a network partition, they lose availability, which means some operations will fail. For such reasons, they are usually cataloged as CA which stands for Consistent and Availability. Also, distributed RDBMS have a reduced availability in case a master server fails. In this case, part of the data may be accessible for read operations but not for the write ones.

The main focus with NoSQL databases started with the assumption that in modern web-based applications, the Partition tolerance is a must-have. Consequentially, in the last few years many different databases were developed omitting many of the common features of RDBMS, but having more focus on Partition tolerance. They can be divided into two main categories:

CP: Consistency and Partition
These databases focus on guaranteeing that each node always reads the same data; if a network partition occurs the requests will either fail or return the most updated value.

AP: Availability and Partition
These databases are designed to guarantee the availability of the system even in case of a network partition or the crash of a server. However, requests may not contain the latest information.

Such a model, however, covers only the behavior of the system in case of a network partition which rarely occurs. Many geographically distributed databases are indeed willing to lose consistency in favor of latency also when no network partition is present. In such a case, a better model is the one proposed by Daniel Adabi et al. [3]. They defined the PACELC model that extends the CAP in case there are no network losses.

PACELC model

As shown in Figure 2.6, in the PACELC model a system can be defined by how it behaves when the network works properly and when it does not. In case a network partition occurs, it has to sacrifice either the Availability for the Consistency or vice-versa. On one side, when there are not communication issues, it has to decide between Consistency and low latency replies. Databases can favor one or the other or can allow users to choose the best behavior for each scenario. In any case, this choice influences the performance and scalability of both the database and the application.

Two Phases commit RDBM and NewSQL databases tend to favor consistency over availability, and therefore they use a master-slave based approach, the Two-Phase Commit (2PC) protocol to achieve distributed consistency. As shown in Figure 2.7, the two phases are:

Prepare phase:

1. The master sends a Prepare request to each slave.
2. The slaves execute the operations and send the Prepared responses (either commit or abort) to the master. The resources are now locked.
3. The master collects all the responses.

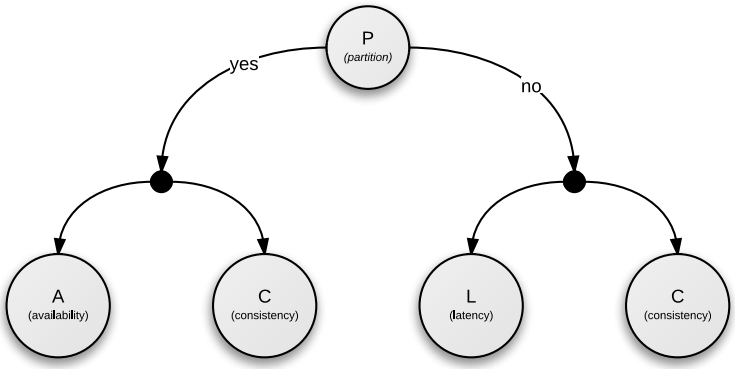


FIGURE 2.6: The figure shows represent the PACELC model

Committing phase:

1. The master decides either if the transaction is successful or not, and communicates its decision to all the slaves.
2. The slaves complete the commit (or rollback) of the operation, then unlock the resources and send an acknowledgment to the master.
3. The master receives all the responses and commit (or rollback) globally.

2PC drawbacks

The problem of the Two-Phase Commit protocol is that it is not network-partition tolerant. If the master fails after the committing request phase, the slaves continue to lock the resources, and they do not know if they have to commit or rollback. The problem in such a case is that the slaves cannot reach an agreement on whether the transaction is completed or not. As we will discuss in the next section, the issue relates to the more general problem of the **shared consensus**, and there are several alternative solutions adopted both in distributed databases and filesystems.

2.2.4 Shared consensus

The simplest way to coordinate many actors, both in computer science and society, is arguably to entrust one participant to decide and organize the others. Such a solution works well with few participants but tends to be problematic in large settings. The main problems are performance and resilience, since a single master node might not be able to manage an increasing number of slaves. Similarly, availability is compromised, as the system might stop working if the master is either temporarily

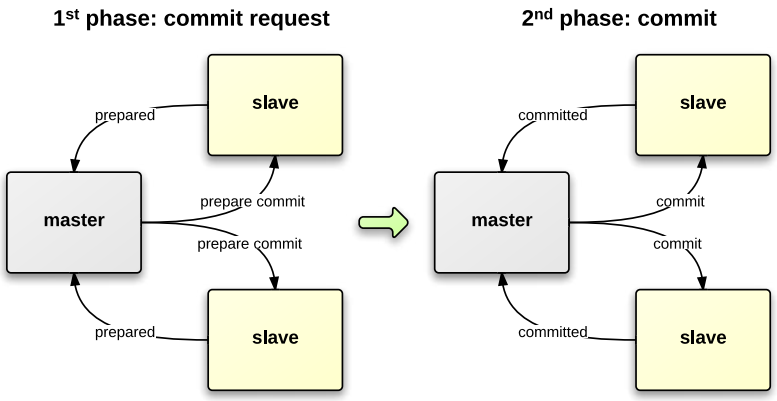


FIGURE 2.7: The figure shows the Two-Phases Commit protocol (2PC)

off-line or out of service. A straightforward solution for the first problem is to split the overall domain into partitions and assign a master to each one. In such a way, concurrent requests over disjoint partitions do not interfere with each other. However, in case of large transactions touching many partitions, the masters are required to coordinate between them, adding a layer of complexity and ultimately reducing performance. An additional problem arises when the data distribution changes over time. Thus a re-balance of the workload between masters is required. While some products promise to automate such a process, they still require additional maintenance work as they have to deal with situations of partition re-sizing or migrations. Such an approach is popular both in distributed databases and filesystems.

In any case, we still have to deal with the problem of a faulty or no responding master. A wide range of algorithms, often called *three-phase commit* (**3PC**) protocol, aim to avoid the case in which the failure of a participant prevents the overall process to proceed. The typical solution is to nominate a new master if the first one fails, but it requires to deal with the leader-selection process ensuring that all peers agree. Among the various ways to implement such a process, the most popular is the *Paxos Consensus Algorithm* proposed by Jim Gray and Leslie Lamport in 2004[49]. Their definition borrows from the more general problem of consensus which is a traditional topic in the distributed computing community. Indeed, many solutions with precise fault models and rigorous proofs of correctness have been proposed in literature[35][84]. In particular, Gray and Lamport use a specific variant called Paxos algorithm[72] [73], an asynchronous, non-Byzantine model, that takes care of all boundaries cases that can arise from lost or delayed messages when all parties

have to agree on a common truth. The algorithm guarantees that if a large enough subnetwork of participants is not faulty, a shared value can be reached.

The participants of the Paxos algorithm can be (not exclusively) proposers, acceptors, and learners. The first group proposes the values; the second are the ones that can promise to accept the value if the majority is reached, and the third ones are those who get to know the final agreement. Each proposal has a positive natural number, which must be incremental and unique among all parties, so that each participant has an individual sequence of possible numbers. We will call this number EID, election ID. For example, if we have ten nodes, the first one might use the EIDs 0,10,20, the second one 1,11,21, and so on. The messages that participants can send are:

Proposers:

1. **Prepare(EID)**: can you accept a proposal with such EID?
2. **Accept(EID, value)**: I accept this value with your EID

Acceptors, when replying to a **Prepare(EID)**:

1. **Promise(EID)**: yes, I can accept this EID
2. **Promise(EID, Accepted(old_EID,old_value))**: yes I can accept this EID, but I previously accepted this *old_value*. This message is sent in case of contentions when multiple Proposers want to reach a consensus with different values, thus it may happen that an Acceptor has already received both a Promise and an Accept with a lower EID.

At the end, the Acceptors send to all the Learners the message **Accepted(EID, value)**. The process starts when a **Proposer** initiates the Paxos, it chooses an EID from its range which must be higher than any EID previously observed. Then, the **Proposer** sends the message Prepare(EID) to all **Acceptors**. At this point, the **Acceptors** can behave in three different ways.

1. If they have not received previously any **Prepare(EID)**, they reply with a **Promise(EID)**
2. If they have previously received a **Prepare** message with higher EID, they ignore the message.
3. If they have previously received both a **Prepare(EID)** and an **Accept(EID, value)** with a lower EID, they reply with **Promise(EID, Accepted(old_EID,old_value))**, to let the Proposer know that another proposal has been previously accepted.

Then, after sending the **Prepare(EID)**, the Proposer will face two situations:

1. If it receives only **Promise(EID)** from at least half plus one of the Acceptors, it sends **Accept(EID, value)** to all of them
2. If it has received any **Promise(EID, Accepted(old_value,old_value))**, it will send A**ccept(EID,old_value)**

Finally, when an **Acceptor** receives an Accept(EID, value), it sends to all Learners Accepted(EID, value). When the Learners receive a majority of Accepted messages, they know the consensus for such a value has been reached.

With such a design the protocol works if at least 2F+1 participants are available despite the failure of F of them. Indeed, differently from the 2PC, after a timeout, a Proposer can retry the process without any risk of ambiguous state.

In the case of databases, we can use the Paxos for reaching an agreement over a commit. The Paxos Commit Algorithm [49] requires running a separate instance of Paxos agreement for each slave. The master initiates a commit; it sends to all slaves a message Prepare. The slaves then reply with either Prepared or Aborted, and instead of having the master collecting all replies, the system uses Paxos to ensure all participants reach an agreement regarding the decision to Prepare or Abort. Once a consensus has been achieved for each slave, all participants will know that the transaction succeeded *iif* all the slaves agreed value is Prepared, and rollback otherwise.

As it is easy to image, the cost of such a distributed algorithm is considerable. The slaves must wait for at least five rounds of message exchanges before completing a transaction, a number that can increase in case of faulty nodes or contention. More precisely, if there are N slaves of which we can tolerate F faulty ones, a total of

$$(N + 1)(F + 3) - 2$$

messages will be sent[49]. In other words, if we add one node, the system will have to exchange at least 3 more messages. Also, as we increase N, we should increase F, and thus the number of messages can rapidly increase, thus becoming a scalability issue in large settings.

Considering the PACELEC model, we can see how a distributed system, even when no partition occurs, may sacrifice trade-off latency in favor of consistency in such a scenario.

Another approach is the one used by Spanner [28], that uses the Paxos algorithm to coordinate leaders and slaves memberships, using long-lived time leases (e.g. 10 seconds) and it shows similar performance characteristics.

2.2.5 Durability

A system ensures Durability if none of the committed transactions are lost in case of power loss in the overall system. The typical approach is to use persistent memory to store a *write-ahead log* (**WAL**) of all completed transactions. The system writes in the log all changes before applying them, so that, after a crash, the system can replay all entries found in the log since the last checkpoint. The canonical method, known as ARIES[78], has been invented by an IBM researcher in the 1990s. A variant of WAL, called *journaling*, is also typically used in the management of metadata in

filesystems which avoids invalid intermediate states such as storage leaks, lost files or orphaned inodes that might occur in case of abrupt power loss. An alternative is a copy-on-write approach, which similarly to the previously discusseds MVCC techniques, creates a new copy of the metadata instead of modifying it, and then it updates the metadata information that pointed to the updated one. In case of a hierarchical structure such as a filesystem, this means that updates can propagate up the root node or superblock.

In the last years, the rise of persistent memory hardware technologies, such as *phase change memories* **PCM** [51], *Resistive Ram* (**ReRAM**), and *non volatile dual in-line memory module* (**NVDIMM**) products such as *Intel's 3D XPoint*[51] (now marketed as *Optane*), are very likely going to change the way durability is achieved. Theoretical solutions, such as short-circuit shadow paging[26] or Barrier-Enable I/O to reduce communication latency between the device and the memory [114], have already been proposed.

2.2.6 Data placement and Metadata management

In distributed data stores, data placement and metadata management often go hand in hand. Data placement strategy influences the distributed architecture entrusted to decide which machine should store each piece of information. Such a decision can be taken by a single actor or it can follow a particular rule shared by all peers. However, deciding who is entrusted to take such decisions or to ensure that all peers agree on the same rule, is part of metadata management.

In literature, we can find plenty of contributions regarding the problem of choosing in which node to store an object and how to ensure a balanced workload between nodes. These algorithms aim to select the best node that can store a particular object. In general, there are three main approaches:

Global mapping

Global mapping is a popular and straightforward solution that uses a global master which keeps track of the position of each item's replicas in the cluster. In such a scenario, the master can decide where to place a copy of the data in the replicas that have less load, and consequentially route requests.

One example is the Google File[46] System and its open source version - HDFS[20]. A master - the *NameNode*- balances the resource utilization by deciding where to send each replica. This allows a fast recovery from a slave failure at the cost of adding a single point of failure and a bottleneck in the system. Even if the master may have shadow replicas, as Konstantin[98] analyzed, the *NameNode* memory footprint grows with the cluster storage size and eventually it limits such architecture. Konstantin showed that 10'000 HDFS nodes with a single *NameNode* should scale up to 100 thousand readers and ten thousand writers. However, this estimation assumes a batch processing scenario with file chunks of 64MB, but to achieve

low latency response time on indexed data, we have to access few KBs at a time, and this dramatically narrows down the upper bound. As proof, McKusick and Quinlan[76] reported that GFS recently evolved to a more complex sharding design with multiple masters thus allowing lower response time and smaller file chunks.

Multiple masters

A way to avoid the limits of a single master is to use multiple ones, but to do so it is necessary to find a proper way to split the domain of responsibility of each master ensuring their workload is well balanced even when the data distribution changes. In RDBMS, such technique is named Sharding and it consists in dividing the data contained in the tables, in vertical or horizontal shards (subsets) and to distribute them on different nodes. In such a way, it is possible to speed up the write operations because, if executed on different shards, they run in parallel on different servers without lock synchronization. A typical example is to divide the table Users into two shards according to their geographic provenance. For example, they might store all the users from a continent in the nearest data center. However, even in such a scenario a global arbiter is required for multi-partition operations such as JOINs or GROUP BYs. A global arbiter is also required when we need to change or rebalance the masters' domain.

In file systems, a similar approach is used to partition the file system tree in subtrees, where a different server manages each one. Similarly, in this case the challenge is to ensure the size of partitions is uniform and there are no hot-spots. An example might be an application that starts to generate thousands of files in a directory subtree. Static subtree partitioning requires a system administrator to manually assign each subtree of the file hierarchy to specific servers called NameServer. For example, all files that go in directory /scratch might be handled by a server, while the ones in /archive from another. In Lustre[86] this is the only possible approach to scale the NameServers. A more flexible approach is the one adopted by distributed systems such as CephFS[112] and GFS [76], that use Dynamic Subtree Partitioning[113] to adaptively allocate the responsability of managing directory. However, it is not clear how the availability and consistency of the system are maintained in case of failure of one of the NameServers during the hand over of a subtree from one master to another. Also, moving a subtree is not a fast process, and it is employed only in case of long-running unbalances, while the short time ones are not addressed. For example, any application scanning the whole filesystem will likely overload one NameServer at a time.

A mixed approach is the one taken by IBM Spectrum Scale, formerly known as GPFS[94], which packs metadata and data together in the same block and thus there is no NameServer. However, concurrent clients still have to coordinate to acquire read and write locks on the files, to update the metadata (inode) and to allocate new blocks. The architecture of GPFS comprises a single centralized TokenManager, which grants authorizations - tokens - to the nodes. The first node to access a file also

acquires the metanode token, which means it is in charge of maintaining the inodes information and lock. In case of concurrent writes on the same file, all nodes send inode updates to the designed metanode. Therefore, GPFS has a mixed approach, as it uses a single unique node to grant and revoke permissions to manage a file and to coordinate concurrent writes on different parts of it; while it uses multiple metanodes to synchronise inodes updates.

Gossip

Gossip algorithms are widely adopted to ensure efficient, reliable and eventual distribution of information among peers. The main idea is that peers randomly communicate with each other, eventually propagating messages. The advantage of such an approach is that even if a node is down or communication is lost, the node will eventually get the information from a peer. The downside is that the information might require a considerable time to be spread. For such a reason, in DynamoDB[34] and the original version of Cassandra[71], the gossip algorithm is only used to maintain information regarding membership, dealing with joining, leaving or no responding servers. However, a later version of Cassandra started using the gossip protocol also to advertise schema changes, such as new tables of data formats. A different use of the Gossip algorithm is the NouDB one, which employs it to broadcast MVCC versions to all nodes[83].

Hashing

A final alternative is to use a deterministic algorithm to decide where to place a copy of the data. For instance, Distributed Hash Table systems - such as Cassandra[71]- use a pseudo-random hash function to place an object in one node of a cluster. Similarly, CephFS[112] uses a pseudo-random function (CRUSH) to allocate random blocks in distributed nodes. An issue of DHT systems is how to achieve balance in load and storage utilization. Its imbalance can be described with the single-choice balls-into-bins problem.

The balls-into-bins problem

The balls-into-bins problem supposes we are throwing at random **m** balls into **n** bins, and we wonder how many balls the most loaded bin will have. This problem is the basis of Hashmaps analysis, where a collision means storing multiple items in the same memory cell and it is an unwanted situation. The typical approach assumes that the number of cells is much larger than the number of elements. Only in 2006 Berenbrink et all[16] analyzed the "*Heavy loaded case*", which interests the DHT systems as we assign many items to each server aiming to achieve a uniform distribution. The authors showed the imbalance decreases with the number of records while it increases with the number of nodes. Indeed, when $m \geq n \log n$ - which is

the usual case for DHT databases - there is a node that receives

$$(m/n + \mathcal{O}(\sqrt{\frac{m * \log n}{n}}))$$

items with a high probability. We can also formulate the formula in terms of a ratio of imbalance between nodes :

$$p = \mathcal{O}\Big(\sqrt{\frac{\log n * n}{m}}\Big)$$

On the other hand, Mitzenmacher[89] demonstrated that one can achieve a better distribution with the "*power of two random choices*": instead of picking at random a single server, one chooses two of them, and selects the least loaded one. In this case the lower imbalance is just $\mathcal{O}(\log \log n)$. However, using the multiple-choice algorithm, we have to face an implementation trade-off. For instance, we can store items uniformly between servers but at the price of penalizing reads as the client cannot know which is the chosen server and thus has to question all replicas.

Microsoft's Kinesis[74] follows this approach achieving a better load and storage balance by allowing the client to choose **r** replicas over **k** possible servers. The drawback is that we have to question all k servers during a read operation and this might result in reducing k times the performance as database systems are often limited by the CPU.

Alternately, we can store multiple copies of the same item so that the client can pick the less loaded replica thus achieving a balanced distribution of read operations. However, it is costly to know the real-time load of each node, and the algorithm should maintain approximated load statistics which might not detect short living imbalances. Also, the second choice penalizes caching systems: if we ask an item twice from the same node, the second request will be faster as it is served out of memory. On the other hand, spreading calls to different servers results in a higher page fault number, and that might nullify the benefits of a more distributed workload. Indeed, Cassandra's driver selects a replica only if the original node is malfunctioning.

It is important to consider the ball into bin problem when designing a low latency system that ensures the proper exploitation of in-memory operations and a uniform workload across nodes. A typical pattern in HPC and Analytics is that the execution is limited by the fact that all nodes have to access to a relatively small partition of the data; a working set might rapidly change over time. As long as in this thesis we aim to exploit key-value data stores for HPC applications our interest is in retrieving any particular subset of data in the minimum time. In this situation, the $m >> n$ hypothesis is not always valid thus resulting in performance degradation.

2.2.7 Data model

If we want to distinguish all systems according to the way they organize data, we should distinguish between:

Key-value databases

In a key-value data model, the value of the key dictates in which server you can store and find a particular item. The advantage of this simple structure is that the queries are fast and it is possible to have a high level of concurrency.

Document databases

Similar to the key-value model, in the document one the value has a proper semantic, and it is usually stored in a JSON or XML format. Also, in many document databases, it is possible to create secondary indexes on one field of the XML (or JSON) structure.

Column-oriented

Column-oriented databases focus on analytical and reporting (OLAP) workloads. Indeed, storing data by columns instead of rows reduces the I/O required when computing aggregation over a large number of rows. Typically, this kind of database does not support row-level transactions, and data is periodically (usually daily)-moved to a transactional database (an OLTP system) through a process called Extraction Transformation Load (ETL).

Row-oriented

A row-oriented data-model is similar to the model used in Relational databases. Its structure is tabular such as the column-oriented one, with the difference that the information is orderly stored, reducing the I/O while accessing a single row instead of a single column such as in the column-oriented architecture.

Graph

A graph database uses graph structures for semantic queries with nodes, edges, and properties. The key concept is each entity can have a relationship -edge - with another one graph. Each graph can be associated with a weight or a value. The problem of this kind of databases is how to partition the data between servers so that a query can be executed efficiently. Indeed, a query may have to visit several nodes to complete, and if these nodes are in a remote location, most of the execution time goes for communication. This is usually the problem when using a random distribution of the data. An alternative is to use domain knowledge to partition the graph, for example storing together users from the same city.

Object stores

The object store data model is probably the simplest way to organize data. Each object is identified by its unique name; there are no folders nor other hierarchical organizations. The user defines a bucket or namespace in which each name must be unique. While there are no folders, some systems may visually aggregate objects with the same name prefix, thus providing a more user-friendly interface to users. However, as folders are not first citizen entities in object stores, there are not operations such as *move*, *delete*, or *list* all contained files in a folder.

Indeed, the list of object in a bucket is usually not kept up to date and it is updated hourly or daily. A common feature of object stores is to have rich support for user-provided metadata which might include information over the ownership, permissions or any other business related information to a specific object. Key-value and object store are usually correlated as both systems map a unique id to a value. While each available product has its characteristics, the main distinction between the two is the data granularity. Indeed, in a key-value database, each key is generally associate to a value with a size comprised from the few bytes to few megabytes, while in the second an object can size is easily be in the range of gigabytes or terabytes.

One important consideration is that the difference between these categories is often blurred, as long as it is possible to use the same system with different models. For instance, in some databases the data model is flexible enough to store data both in a column or a row oriented layout.

2.3 Distributed data: SQL, NoSQL and Parallel File systems

Now that we have defined some of the techniques that we usually find in distributed datastores, we can proceed to describe how generally SQL, NoSQL, NewSQL databases, POSIX or non compliant file systems and object stores can be distinguished.

2.3.1 SQL databases

Let's start with SQL databases. Data is arranged in tables with fixed columns and types, and we can access and update single rows or tuple without worrying of consistency between concurrent clients. Thanks to the relational model - introduced in 1969[36] - we can design the tables, and the links between them. This can be done by taking into account the relationship between the various business entities that we want to model, while we can (hopefully) not worry about how to organize the physical layout of data to achieve better performance. As we described before, SQL databases must comply with the ACID properties. To do so, they usually use 2PL or MVCC concurrency control schema to preserve Atomicity Consistency and Isolation, while WAL provides the Durability. Scaling the database to multiple nodes is

generally achieved by a shim layer that intercepts queries and routes them to the correct master partition while the 2PC protocol is used to coordinate intra-shard transactions. We will consider PostgreSQL as the main representative of these databases as long as it is open-source, advanced and supports multidimensional indexing. In particular, PostgreSQL uses **MVCC**, **2PC**, **B+trees**, and **WAL**.

Then, with the explosion of the internet and user-generated content, Internet services providers started to put under new stress both existing databases and filesystems.

2.3.2 NoSQL

On the database side, NoSQL databases have many combinations of architectures and data models, as the main idea is that a "one-solution-fits-all" was not enough anymore and after 40 years of development a strip down of functionality was necessary to achieve the desired performance [103]. After all, even if OLTP systems are capable of long-running analytic queries, the most popular approach is to relax consistency and periodically copy all information into an OLAP system.

Among the many NoSQL databases, we will consider Apache Cassandra[70], MongoDB and Redis.

Although, initially, Cassandra did not implement transactions, now it supports a few subsets of lightweight partition levels, compare-and-set operations, such as INSERT IF NOT EXIST using a Paxos based consensus protocol. They are, however, restricted to single partition key updates. The data is organized according to a Log-structured merge-tree, **LSM-tree**, a data structure optimized for high insert volume that uses multiple separate structures to maximize the hardware resource usage. Then, data distributes according to the key hash - **DHT**. The information about membership and schema are shared through the **Gossip** algorithm. Differently, MongoDB is a Document database, usually in JSON format, with document-level transaction support using **MVCC** - but it does not avoid skew writes 2.2.14, and **2PC**. In case the master fails, a new master is elected, but in the meantime some partitions are unavailable, and there might be an inconsistency between the new and the faulty master [3]. Both Cassandra and MongoDB use WAL for durability (in MongoDB it is often called journaling).

Redis is an in-memory key-value database with an advanced set of programming functionalities as maps, lists, sorted sets or hyperloglogs. All operations are in memory, and the secondary storage is used only for periodic checkpointing using a compact but not indexed file format. Alternatively, it can use append-only files for higher durability (similar to a WAL). In any case, the size of the database is limited to the size of the primary memory.

2.3.3 NewSQL

As a counterbalancing force of so many low-level and highly specialized databases, NewSQL databases started to target applications that need, or do not want to deal with consistency issues, but yet require the scalability and the high-availability that typical RDBMS system could not provide. In this category, we will consider Google's Spanner[28], VoltDB[102] and SAP HANA[99].

Spanner[28] was developed by Google in 2012 as they claimed NoSQL DBMSs caused their developers to spend too much time writing code to handle inconsistent data and using transactions made them more productive. In their design, they use MVCC with 2PL to coordinate writes, while they use Paxos to manage distributed transactions and atomic clocks and GPS to ensure external global consistency. With such a design, Google claims to have created a system that has (almost) all three CAP properties: Consistency, Availability and Partition tolerance. Spanner has achieved an availability close to 99.999% thanks to the fact that Google owns its own wide area dedicated network. Thus, it can guarantee network partitions are extremely rare events[21], making the system Consistent and Available de facto, as they claim.

Contrarily from Spanner, many other NewSQL databases have taken the "in-memory" approach. In their paper "The End of an Architectural Era: (It's Time for a Complete Rewrite)" published in 2007, Stonebraker and Co. [103] analyzed how much performance was lost in RDBMS due to a legacy architecture. They also realized that specialized engines could achieve nearly two orders of magnitude increase. The aftermath of this study is VoltDB, an in-memory database that tries to leverage a multi-core architecture with a share-nothing design. In VoltDB the database is divided in shards, one-per-core, and the transactions can run one-at-a-time, with no distributed locking, but it has to use a centralized scheduling manager for transactions that span over many shards. This approach works very well with "small" transactions that stay in the same shards, but it does not work with long lasting or large transactions as the whole system would effectively run one operation at a time. SAP HANA[99] is another NewSQL in-memory database that aims at excelling in both transactional and analytical workloads. To this end, it uses a hybrid memory storage format that combines three different ways to store data in the record lifecycle - a write optimized way, a columnar way and a compressed way. Initially, the whole database had to fit in memory, with a severe impact on cost, but now it supports off-loading to disk to rarely used data. At the same time, SAP HANA ensures durability with a WAL and periodic snapshotting, which makes the database span from in-memory to persistent storage in a configurable way.

On the other end of the spectrum, we have file systems and object stores, but we shall distinguish between POSIX parallel filesystems; such as GPFS[94], LUSTRE[86], BeeGFS[91], and CephFS[112]; and the not-POSIX filesystems which the notable champion is HDFS[97]. Then we have distributed object stores, which can

be consistent or eventually consistent. Ceph Object Storage and Google Cloud Storage fall into the first category, while in the second we have Swift and Amazon S3.

2.3.4 Distributed File Systems

GPFS[94], now in the IBM spectrum scale, is a very popular choice in HPC, and it is also the system used by the Barcelona Supercomputing Center. GPFS uses a mixed approach as it stores metadata and files together on the disks, stripping the files in parts that are distributed along the nodes so that it can improve throughput. However, while the filesystem tree structure is distributed between disks, the nodes are required to acquire permission to read/write and create files. The authorizations (tokens) are granted by a unique TokenMaster which can become a performance bottleneck in some particular cases. Differently, GPFS uses a multi slaves approach to handle the inodes; the first node to acquire the token for a file becomes its metanode, and so all operations concerning the update of the inode, from reading/writing time updates, new directory etc, are handled by a single entity while other concurrent clients have to coordinate with it. In case of a failure of both TokenManager and meta note, the system eventually chooses another candidate but in the meantime, requests can fail, and the system is partially unavailable. On the other hand, LUSTRE divides metadata and data blocks using a single NameServer or a multiple manually partitioned NameServer, each of them is responsible for a partition of the filesystem tree. In both GPFS and LUSTRE, redundancy is provided using hardware RAID systems, on top of which optional software replication can be added on selected files. A diametrical approach is the one taken by CephFS that instead assumes the failure of a disk on large installation is frequent, thus it directly manages the replicas of the data according to a pseudo-random hash function, CRUSH weighted to take care of the disk placement (rack, data center, etc) and available resources. Regarding the filesystem tree, CephFS separates metadata and data by using partitioned NameServers. Differently from Luster, CephFS employs Dynamic Subtree Partitioning, that allows to automatically move the responsibility of a NameServer to another with a process similar to the 2PC.

The last of the filesystems we are going to take into consideration is the HDFS, the Hadoop Filesystem which employs a single NameServer with stand-by replicas ready to take over in case of crashes of the NameServer. The NameServer also decides where to place a copy - usually three - in different nodes depending on their utilization. The data nodes are usually co-allocated with the Hadoop/Spark ones so that computation can take place in the node that contains the data. Indeed, differently form typical HPC installations, Hadoop is designed for a cluster of commodity servers with the locally attached disks with no hardware redundancy, where the Hadoop daemon co-lives with the Hadoop computing application. Having a single NameServer has been often pointed out as a major bottleneck of HDFS, and thus many solutions have been proposed. Among many, Spotify's proposal, HopFS[80],

which uses a NewSQL database - MySQL cluster - for the NameServer management. They observed 16 times improvement of throughput.

2.3.5 Object stores

Object stores have the advantage that they do not have to manage the filesystem tree. Thus they put less stress on the centralized NameServer which is usually only used for creating unique buckets/namespaces. On the other hand, they usually have to handle the massive quantity of data, often over vast distances. Above the many Object stores, it is worthy to consider Amazon S3 and Swift for the eventual consistent data stores, while Google Cloud storage and CephFS are examples of consistent ones.

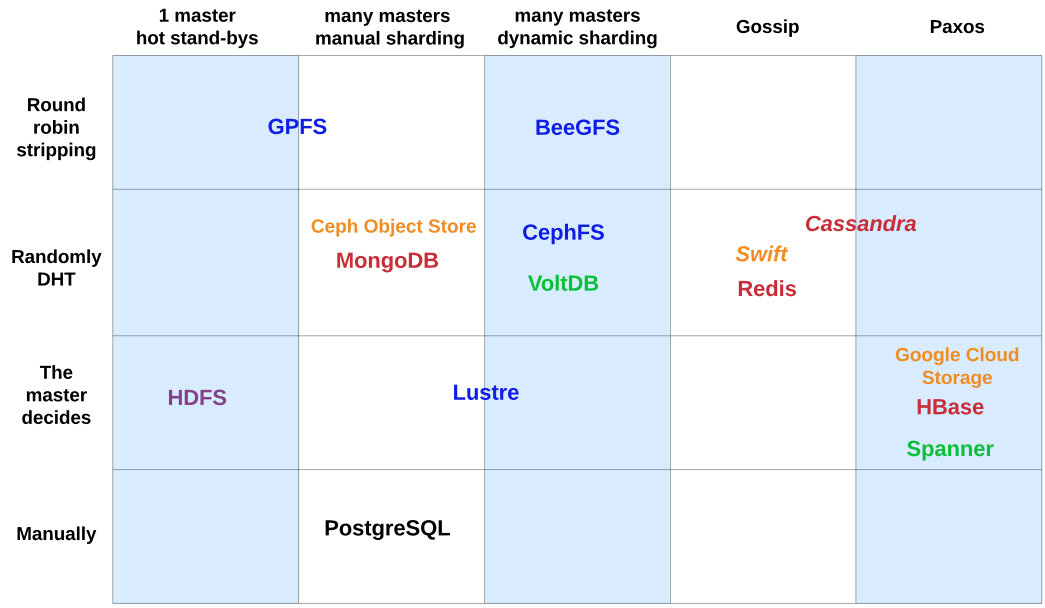


FIGURE 2.8: Different approaches used by data store systems to handle metadata and coordination vs data placement.

In Figure 2.8, we can see how the different available products are designed and what they have in common. In particular, it is interesting to note that there is not a unique or globally accepted architectural solution for none of the categories. For example, Google seems to favor Paxos master slaves architecture, and its competitors prefer more peer-to-peer solutions. Similarly, each of the products that target the HPC market have different architectures. In general, we see similar solutions reused in several applications with varying levels of nuance.

2.4 Multidimensional indexing

In literature, there are dozens of contributions about multidimensional indexing algorithms and data-thinning techniques. The first are divided in two main categories:

Space partitioning indexes and Binary Tree evolutions.
With the term *space partitioning* we refer to a class of hierarchical data structures that recursively decompose a certain space into disjoint partitions so that any element can be stored in one, and only one, partition. This constraint leads to a simple search procedure, but it causes the tree to be skinny, tall and potentially unbalanced.
Indeed, while a one-dimension binary tree can use node rotation techniques to re-balance itself, the lack of a universal order on vector spaces, makes these operations impossible.
Differently, other index algorithms, such as the R-tree and its variants, do not have the constraint of disjoint spaces. Each node is in charge of a specific space, but nodes can share common areas. This relaxation allows the node to have a better fan-out and balance, however it comes with a higher search cost as long as results can be found on multiple tree paths.
Although many algorithms are available in literature, we will describe only three of them, as they are the most common, and most of the available algorithms are their evolutions.

2.4.1 Quad-tree

The Quad-tree is a space partitioning algorithm where the splitting strategy is "space driven". This means that, no matter how the data is distributed inside a block, when it splits, the resulting children nodes will have a size and a position which depend only on the height of the tree and space dimensions.
For example, in a two-dimensional space, a cube splits into four sons, each of them covering one fourth of the space. Similarly, in a three dimensions space it breaks into 8 children, and so on, increasing the number of dimensions.
The benefits of this algorithm are that, given the constraint dimensions of the cube, they can be coded in few bytes. Moreover, it is possible to use space filling curves in order to enumerate each cube. In this way, when querying for a specific region, it is possible to calculate which will be the prefix code of the interested nodes, thus allowing extremely fast look-ups.
On the other hand, if the data is not uniformly distributed, this algorithm produces unbalanced trees, as long as the denser areas result in deeper tree paths. Moreover, the algorithm is forced to half the domain of each dimension at each split, and this is an undesired behavior, particularly when data clusters on one dimension only.

An Oracle study [68] suggests Quad-trees could be recommended for update-intensive applications using simple polygon geometries, high concurrency update databases, or when specialized requests such as "touch" are frequently used.

2.4.2 KD-tree

The KD-tree is a space partitioning algorithm where, unlike Quad-trees, the splitting strategy is "data driven". The splits are executed one dimension at a time: the elements of the nodes are ordered according to the chosen dimension. The node is then split by partitioning across the median element. The algorithm continues cycling on all dimensions.

The benefits of the KD-tree is that it produces balanced trees when constructed on static data, but inserting new elements on an already built index can lead to an unbalanced tree. Another drawback compared to quad-trees is that a query must descend through all the node hierarchy in order to know the dimension of a node, so fast look-up methods are not possible.

2.4.3 R-tree

R-trees have been proposed for the first time in 1984 by Antonin Guttman [50]. Since then, many variations of the original algorithm have been developed over the years: R+Tree [96] in 1987, R*tree [15] in 1990, Hilbert R-tree [65] in 1994, to more recent researches such as Priority R-tree (PR-tree) [11] in 2008 and FLAT [105] in 2012. This non exhaustive chronological list of algorithms, shows how the research on R-trees has a long history and is still active today.
Without entering in detail in each version, the original R-trees' approach is firstly to relax the constraint of the shape and size of the boxes and secondly to rely on heuristic policies to decide how to subdivide an area.
The idea is to split the space in rectangles (the R of R-trees) using data driven strategies. When a node is full, it divides into smaller ones, and a heuristic approach is used to divide the contained elements into subgroups. For example, one technique consists in dividing the rectangle at the median point, while other ones use more complex data clustering methods.
Once the points are grouped, the algorithm calculates the Minimum bounding rectangle (MBR) which contains the elements. When inserting new elements in the index, the R-tree starts searching from the root which node has a MBR that can contain the element. In case there are any, it picks the node whose MBR requires the least enlargement necessary to accommodate the element. In this second case, it may be necessary to propagate the enlargement of the MBR up to the root node.
As Oracle's paper [68] states, in most cases R-trees outperform Quad-trees up to a 2-3 factor.
The main disadvantage of R-trees, which is also the reason of the existence of so many of its variants, is the overlapping of the rectangles. This adds an overhead in the query as long as the algorithm must visit more blocks and descend multiple paths to satisfy a request.
This has historically been considered as a drawback as long as it produces additional I/O on the rotational disk. In modern distributed databases, it would be interesting

to see whether or not these additional requests would lead to a performance drawback, or instead a gain thanks to the increasing level of parallelism.

2.4.4 Distributed Multidimensional indexes

Recent papers have proposed different alternatives on how to implement both Quadtrees and R-trees, as well as hybrid approaches, on distributed databases such as Cassandra or HBase. For instance, in 2013 Ling-Yin Wei et al. [111] proposed a hybrid approach named KR+-index: on a higher level they use a Quad-grid with Hilbert keys so that sequential reads can perform better. At a lower level, they store data into rectangles produced by R+tree indexes. Unfortunately, their indexing method cannot support efficiently approximated queries, which is a fundamental requirement for our application.
Other works, such as HGRID[52] and MD-HBASE[81] present the same limit. The work of Zhang et al.[119] in 2009 takes a different approach adopting a master-slave architecture when considering the problem of consistency in DHTs. The master creates a gross grain R-tree index where each leaf points to one or more slave nodes. Thus, each slave node creates a KD-tree on the space area assigned. The drawbacks of this approach are that firstly the system needs a master, which is a possible point of failure of the whole system which damages the overall availability. Secondly, distributing data across by assigning to each node a space area, can lead to an unbalance workload: indeed, a logical hot spot in the data will result in hot spots in the cluster.

2.4.5 Multidimensional sampling

In literature, we can find little work on the generic problem of multidimensional sampling while we can find a considerable corpus of contributions for the special case of Geographical sampling. Indeed, regarding the problem of sampling geographic data sets, Sharma et al. [93] propose an extensive theoretical analysis of the data-thinning problem for a various set of situations. They propose an integer programming formulation of the data-thinning problem that allows analyzing the influence of all possible constraints. As described by Sharma et al. [93] in their work, these constraints are:

Visibility: there is a maximum number of elements you can visualize at the same time.

Zoom Consistency: if an element is visible in a zone at a given level of zoom, it must also be visible when narrowing the visualization (zooming in).

Adjacency: at any zoom level, if an element is visible in an area, it must also be visible in all contingent areas. This constraint ensures consistency while "panning" in a visualization.

Following the previous constraints, we can aim at different objectives; for example:

Maximality: show as many records as possible while respecting the consistency constraint.

Spatial Density: the visualized elements have the same spatial distribution of the whole data set.

Fairness: each element has the same probability of being visualized.

Importance: the higher the importance of an element, the higher should be its probability of being visualized.

Some of these objectives are mutually exclusive. It is easy to see that one must either choose to pursue the goal of Fairness or Importance. Similarly, one must choose between Maximality or Spatial Density. Let us imagine we are visualizing an area at an arbitrary zoom level. In this area, **N** elements are visualized, which represent the **P**% of the elements present in that zone. Now, if we want to move to a contingent area, we should decide whether to visualize **N** elements from this area or the **P**% of them. In the first case, we aim to the Maximality goal while, in the second case, we preserve the Spatial Density.
Both approaches have their benefits and drawbacks, and thus we will study both cases, leaving the user the opportunity to choose whether to use one instead of the other.
We have chosen to focus on Fairness as long as it fits better our application case. However, aiming to Importance would only require trivial modifications.

For our application, the most interesting scenario is the one considering a data set composed of only points. As a matter of fact, our work was inspired by their proposal: a randomized thinning algorithm for geoset composed only by points. The authors proved that we can respect the constraints of Visibility, Zoom Consistency and Adjacency by simply assigning to each point a number - a priority - independently and uniformly at random. Their algorithm aims at building a Spatial Tree ST, which is a balanced 4-ary rooted tree with Z levels representing the possible levels of zoom. To build it, they propose to use random numbers defining a global ordering between points. In such a way, it is possible to select which points will be shown in each zoom level by simply picking the first n elements sorted by priority.
However, Sharma et al. [93] proposes to build the ST with the support of an external Spatial index that can be, in their opinion, any "standard fashion spatial index" such as Hilbert or Guttman. The external spatial index is used to retrieve all the elements present in a given cell ordered by their priority. Unfortunately, in the paper, it is not clear how this spatial indexes should be created, but we can consider two possible solutions. In the first, they build a bi-dimensional R-tree on the dimensions x and y (or lat and long). In this scenario, filling the higher levels of the ST - those with a coarser zoom level - would require to scan the whole index. At the same time, the lower levels of the ST - those with higher zoom detail - would likely require reading the content of a single R-tree rectangle. The second solution is to build a 3 dimensional R-tree on the dimension x, y and priority. In this case, the higher levels will

most likely match with R-tree's rectangle, as long as the elements will also be partitioned in rectangles according to their priority, thus allowing to efficiently select the elements with high priority value. Again, this is a drawback of the ST's lower levels, the ones which contain elements from a narrow x,y spatial range but a wide - if not the whole - priority range. This kind of query, implemented on a R-tree, will likely require to navigate through several rectangles, thus requiring to filter in memory a considerable amount of data.

Assigning a random priority to each point to then select the first k elements with a higher value, resembles the problem of NN-neighbour and top-k queries which are a classic subject of both spatial and common databases.
A top-k query aims at finding the first k elements that match the given constraints, sorted by priority. For example, in a full-text search the priority is how a document relates to required keywords. The typical solutions for the sole top-k queries are Fagin's threshold algorithms[39] and its evolutions such as the one proposed by Theobald et al.[107]. These solutions use multiple weighted inverted indexes, one for each attribute, concerning the relevance of each item to a specific characteristic. A query is then resolved by visiting the indexes related to the characteristic interested by the query. In the case of spatial databases, these techniques have been extended by combining the usage of R-tree and inverted indexes. For example, Cong et al. [27] propose the IR-tree, which maintains an inverted index for each node of the R-tree. Similarly, Zhou et al. [120] propose to create an R-tree for each distinct keyword and, therefore, merge the results. However, all these solutions are built with the idea that the priority of each point depends on the query. Indeed, whether a query includes or not a characteristic, the relative relevance between points changes.

Differently, in the Alya use case (Section 2.1.2), as well as others, the points' priority does not dependent on the query and, on the contrary, it needs to be constant to ensure the property of zooming consistency and adjacency. This constraint allows us to benefit from this global priority order to optimize how the elements are stored. These optimizations are not possible with top-k queries, as long as the priority is computed on-line for each query, with considerable additional performance cost.

2.5 I/O in HPC

Nowadays, high performance computer simulations run on thousands of parallel cores and generate outputs of the order of TeraBytes. How to parallelly store, organize, and analyze these results has become a key factor for both performance and usability. Traditionally, the output with the results is stored in files. There are plenty of different formats, with some of the most popular being HDF5[41] and netCDF[64]. Although their data models differ, since version 4, netCDF uses HDF5 as the backend, and thus they have similar performance characteristics. The problem with file storage is how to achieve high performance while thousands of concurrent processes

are accessing in parallel. A straightforward approach is to generate an output file for each process, but this then requires having thousands of files which are complex to manage and analyze. Alternately, HDF5 allows creating data set partitions – hyperslabs – that can be written and read independently by different processes. HDF5 offers two MPI-based accessing modes: independent and collective. In the first case, all processes can access the file without any synchronization, but this can cause several random I/O calls and thus penalises performance. Collective access reduces the I/O by assigning a subset of MPI tasks to act as "aggregators" so that they can gather smaller and independent requests in larger and contiguous I/O accesses. On the other hand, the collective tasks add communication and synchronization overhead, which may reduce performance and scalability.

Whether to use an approach or another is strongly influenced by the data layout and the distributed filesystem[59][88] used. For example, if each process writes into a hyperslab that maps to adjacent disk partitions, there is no need for collective I/O. However, this is not the case for particle simulations. The typical dimensions of particle simulations are time step, position, and particle identifier. However, each process simulates all the particles that fall into its sub-domain at a given time step. Therefore, each process writes into randomly distributed positions. Indeed, libraries specifically tailored for particles simulations, such as H5hut[58], heavily employ collective I/O.

An alternative approach is the one that Alya implements when writing Computational Fluid Dynamic in HDF5[29]: each process writes into a different dataset inside the same file. In this manner we can achieve good I/0 performance, but the parallelization of the simulation fully dictates the file layout.

2.6 HPC visualization

Visualization in HPC is a vast topic that spreads from studying new ways to represent the information graphically: how to correctly encode data into images that can be correctly interpreted by humans; to new ways to interact with data, like virtual or augmented reality; and to the design of hardware architecture that allows to render in real-time huge quantities of data. While the topic is fascinating and vast, we will only focus on the definition of Post-hoc, In-situ visualization and data thinning, as long as a more extensive analysis lies outside the scope of this thesis.

However, bandwidth and storage limit the amount of information we can effectively analyze. Indeed, in HPC systems we have to deal with three main funnels in terms of data bandwidth. The first is the speed data can be moved from the CPU and the RAM, the second is the speed at which we can move data between nodes and lastly the speed we can persist data to disk. With a broad estimation, we can take as an example the Marenostrum IV supercomputer. Each node has 12x8 GB DDR4-2667 DIMMS, which translates to a theoretical maximum throughput of 21GB/s each RAM bank, for a total of 126GB/s for node. Each node is interconnected with a

100Gb Intel Omni-Path, with a throughput we can approximate to 10GB/s. If we want to persist the information in the local SSD, we will touch the limit of approximately 320MB/s, while with a more performant NVMe disk or GPFS we can reach 2GB/s at most. In other words and with a broad estimation, all of the data we can simulate we can move in the network one-tenth and store only one-hundredth. As a result, simulation codes usually reduce the quantity of data sent to persistent storage using different strategies, such as random sampling of periodic snapshotting. However, in some applications, such a reduction of precision can be problematic and thus an in-situ approach is adopted. In-situ comes from Latin, it means "on-site" and it refers to the architectural choice of moving the analytical computation or the rendering required for a visualization closer to the simulation. According to Rivi et al[90], there are three main approaches for *in-situ* visualization:

Tightly coupled: the visualization/analysis code has direct access to the simulation memory as they run on the same code and they are integrated. This approach virtually allows to analyze or visualize all the data that is simulated, but it has the drawback that the two systems compete for the resources, thus potentially limiting the scalability of the simulation itself.

Loosely coupled: the simulation and the in-situ code run on different nodes, and they share data with either a pull or a push driven approach. As the data needs to be copied between the two systems this approach is limited by the network capability.

Hybrid: in this setting, a tightly coupled part of the in-situ code reduces the quantity of data and then sends it through the network for further processing.

Existing frameworks use different approaches. For example, ICARUS[18] implements a virtual file driver for HDF5, so that the existing application can use the HDF5[41] API to send data to the ICARUS as it was writing to a file. Other libraries, such as the ParaView Coprocessing Library[38] require to modify the simulation code to be integrated with the co-processor and to communicate with the outside visualization code. The approach they encourage is to use the co-processor to extract the salient features from the data and then send them to persistent storage. In a second phase, the visualization code can interactively visualize and render the stored data.

An alternative is Paraview Cinema [6, 8] that stores a smaller resolution version of the data, tailored for visualization. The philosophy behind Cinema is that beyond certain scales, it is more efficient to store images with all possible combinations for a visualization (distance, angle, variable, etc.) than the actual data needed to produce those images.

These approaches show how in-situ analysis is often combined with *post hoc* analysis. The first allows overcoming the bandwidth limitation, while the second ensures more flexibility.

Indeed, the problem of in-situ visualization is that *panta rhei*, "everything flows," and therefore we cannot visualize twice the same data. In the case of visualization, this means that the in-situ application generates videos or images with a single point of view and focus, limiting the possibility to explore the results. Ahrens et al. [7] proposed as a solution to take multiple images of the simulation with different points of view and store them in a database for a second-time analysis. This approach guarantees more flexibility, but still, the user cannot generate new analysis or renders in a second moment.

Additionally to more interactivity, a *post hoc* approach also has the benefit that can use more extensive sets of data to generate visualization, as the size of the RAM does not limit it.

All previously cited solutions integrate with ParaView, which is one of the most popular tools used for HPC visualization. ParaView is a modern visualization and analysis tool that has post-processing capabilities for modeling and simulation workflows. Its plugin in architecture and the fact that it is open source made it one of the most used visualization programs in HPC. It also allows distributed processing using MPI to distribute the worker between slaves, while an external master is used to present the result to the user.

2.7 On the state of the art

As we have discussed, there are plenty of different technologies and methodologies that all aim to the same goal, managing information, but with very different approaches.

We can see that we have two central tendencies that go in opposite directions. From one side, we have the strong advocacy of highly tailored and specialized piece of software[103] which often comes with a significant tear down of functionality. For every job, we can use a specific tool, which then brings us to adopt tens of different solutions that have to be integrated and maintained with no little human effort. Furthermore, even understanding which is the precise scope of each tool can be challenging.

On the other side, we are assisting a push back to the opposite direction, with new technologies that aim to be the solution for highly available and distributed SQL[28],[102], while others even target themselves as HTAP, thus ready to tackle both OLTP and OLAP workloads[99]. While the concept of a unique solution for all applications sounds appealing, it is, in our opinion, achievable only for a few cases, while any divergence from the "template application" would most likely require more *ad-hoc* solutions. On the other hand, the effort of providing high level, easy-to-use interfaces is not only appealing but necessary as the level of complexity of software is likely going to rise in the future. Thus, we cannot expect developers, industrial or scientists, to be expert in all the areas required.

At the same time in HPC, we discussed how even if in-situ analysis and visualization focus on overcoming the memory barrier limitations; hybrid and *post hoc* approaches are still convenient and often necessary. Indeed, going toward more data-driven research, scientists need tools that support a more interactive and exploratory approach in the analysis of large simulations, and in our opinion, this is only possible if we store in persistent memory snapshots or random samples of the simulated experiment.

To deal with the throughput and analytical requirements of HPC applications, our tool-of-choice above all the distributed data storage alternatives are key-value databases. In our opinion, key-value databases have the right trade-off between performance and a high-level interface. The ACID guarantees provided by SQL and NewSQL databases represent the highest level of abstraction a user can require for managing data, but they come with a considerable performance cost. As we discussed before, supporting distributed transaction in a highly available matter is possible, but it comes with a considerable overhead regarding latency and network traffic. A cost we can avoid in the many scientific applications where the data is never updated nor canceled, thus reducing the need for transactions.

On the other hand of the spectrum, distributed filesystem or object stores ensure the highest writing throughput, but only if the I/O pattern is regular and sequential, which is more an exception than the norm in our experience. Furthermore, they do not support indexing, forcing the users to scan the whole file to find the required information.

For such reasons, we decided to use key-value databases, as they can achieve high-throughput with low latency even with unregular I/O patterns and they also support secondary indexing. Above many available databases, we decided to study Apache Cassandra due to its popularity, its large corpus of both academic and industrial research and the vivid open source community that develops it. Furthermore, Apache Cassandra can be easily integrated with existing powerful distributed computing frameworks such as Apache Spark and PyCOMPSs with Hecuba, which allow to query databases and perform large aggregation or complex machine learning analysis at scale.

2.8 Apache Cassandra's architecture

Apache Cassandra inherits many characteristics from *Google BigTable*[19]'s data model and *Amazon's Dynamo*[34]'s architecture. We can simplistically describe Cassandra's data model as a large distributed HashMap, where each entry contains a SortedMap. We need the key to access the outer map, as it is used to calculate which node contains the data. Differently, the second map is orderly stored on a single server. Thus it is also possible to retrieve slices of data with any desired range of key values.

In CQL, the first level key is named **partition key**, while the second one **clustering key**.

LISTING 2.2: Java-like signature of the Cassandra data model

```
HashMap<K1, SortedMap<K2, Object>> cassandraData
cassandraData.get(k1).slice(fromK2, toK2)
```

2.8.1 The cluster structure

The data distributes among the nodes according to the **partition key** value. Thus a Table row is the smallest unit that we can split and scatter across multiple nodes.

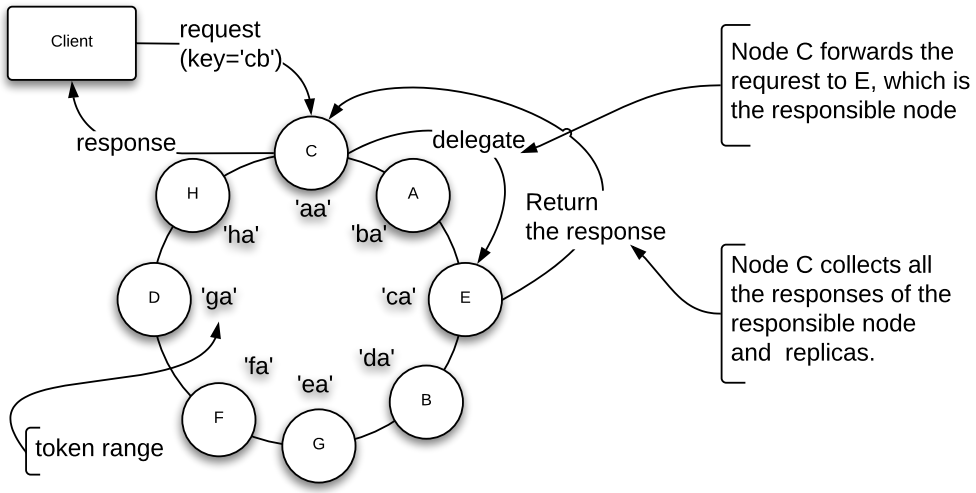


FIGURE 2.9: How data is distributed among nodes in Apache Cassandra

Figure 2.9 is a schema of how the data is distributed across nodes in a Cassandra cluster. Each node is assigned a token range, which indicates the interval of value for which it is responsible. The nodes has a random position in the logical ring, where every node is responsible for the values included between the value of its token and the token of the next node.

When a user wants to write or read the value of a row, it must provide its **partition key**. The database computes the key hash and uses it to forward the request to the node with the assigned token range. The hashing step is required to avoid that logical hot spots in the data, result in unbalanced work distribution across the nodes.

2.8.2 Cassandra's write and read paths

To implement the D8tree and the AOTree we modified some internals of Apache Cassandra. Thus, before explaining our work, it is vital to describe all the steps that the database performs for each write or read operation.

Figure 2.10 shows the two different paths of action that Cassandra follows when it receives a write or a read request.

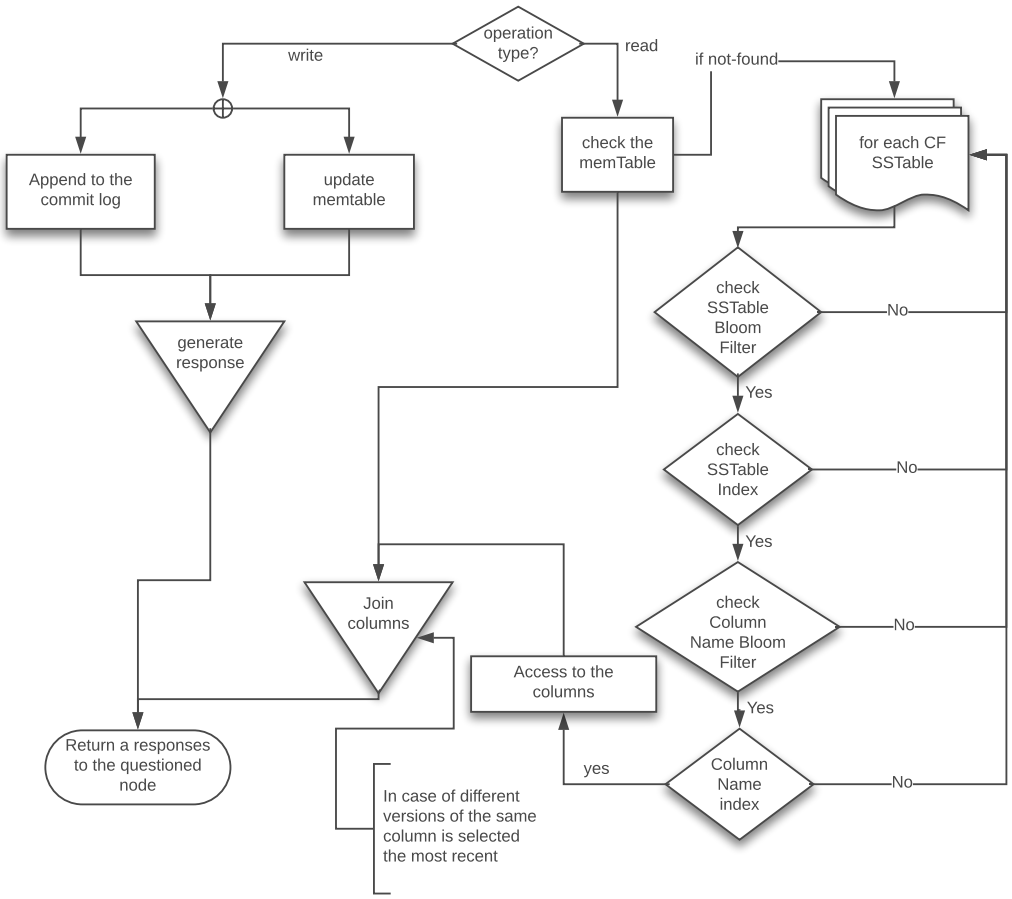


FIGURE 2.10: A schema describing the steps involved during a read or write operation in Apache Cassandra

Write path In the case of an update, Cassandra performs two main tasks. At first, it records in the commit log (WAL), the operation to ensure durability. Secondly, it updates an in-memory structure named Memtable. A Memtable is a write-back cache that contains the most recently updated values of a specific table. Once updated the commit log and the corresponding Memtable, Cassandra can immediately acknowledge the success of the operation to the client. In a second moment, when a Memtable reaches a configurable size or temporal threshold, Cassandra writes in persistent storage its content in a single file called SSTable.

The advantage of such a design is that Cassandra can optimize the I/O reducing the number of random accesses to disk, favoring fewer but larger sequential operations. Indeed, Cassandra records multiple operations every time it updates the commit log. Similarly, delaying the flush of Memtables to SStables speeds up I/0.

Compaction Periodically, the Memtables are serialized and flushed to the hard disk in SSTables. An SSTable is immutable and contains data relative to a single

Table, but there are usually multiple SSTables for each table. Since SSTables are immutable, when we want to modify a value that is already stored in an SSTable, Cassandra does not change the old file but it writes the updated value in a new SSTable file. Similarly, Cassandra handles deletes using a "tombstone" indicating that the element no longer exists. Periodically, a process named Compaction merges multiple SSTables in a unique one removing the duplicated and outdated values.

Read path As described in Figure 2.10 when a Cassandra node receives a read request, it has to look into each MemTable and each SSTable. In the presence of multiple versions, it returns to the most updated one. Since the SStables stay in secondary storage, Cassandra tries to reduce the I/O keeping in memory an approximated probabilistic structure that it uses to determine when skip reading a specific SSTable as it is impossible that it contains the required value. Cassandra manages multiple bloom filters [25] for each table, both at the row and the column name level so that it can guess if a file has any value matching with both the partition and the clustering key. Given the probabilistic nature of the Bloom filters, they have relatively rare false positive results, which means the database might access an SSTable even when it does not contain the required information.

2.9 The importance of the data model

An important milestone in developing any distributed application, and even more designing novel MIS algorithms, is to understand how the data model influences performance. Indeed, NoSQL databases do not employ the relational data model, but they expose to the user directly how data will be stored in the distributed system. Therefore, different layouts - data models - dictate what information can be retrieved in a matter of milliseconds and what in hours.

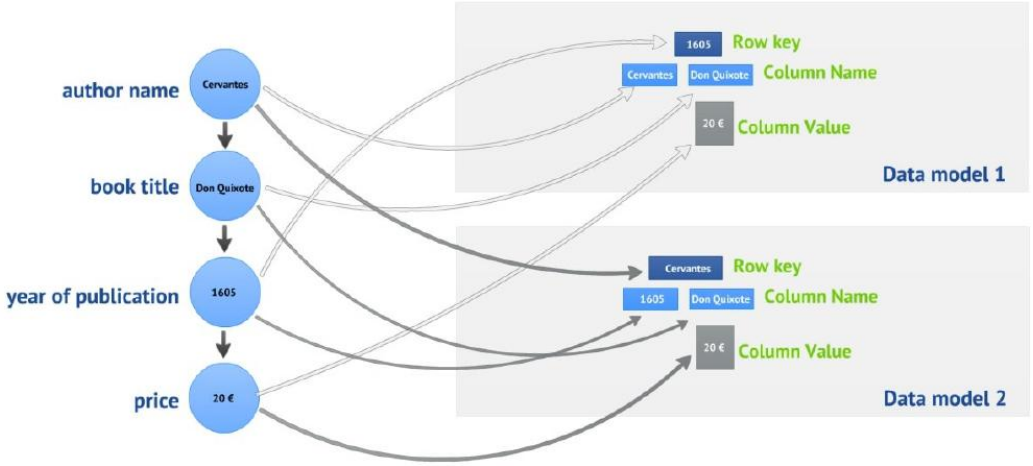


FIGURE 2.11: An example of two different datamodels

Figure 2.11 shows a simple example of how the data model can influence performance. Let us imagine we have a bookshop and we want to store in Cassandra how to catalog all of our inventory. Each book has an author, a title, a year of publication and a price. We can assume that the combination of author and book title is unique. Even in such a simple setting, we have to decide between two different ways to store data. If in our application it is vital to look for all the books published in a particular year, we would likely use the Data model 1, as all publications will be organized chronologically. However, if we only use this data model and we want to search all books from a given author, we will have to scan the database and check in which year he or she published a book. Vice versa, Model 2 allows a fast lookup by author name, but it does not allows other types of query. In the development of an actual application, the developer should, therefore, decide which kind of queries are essential and use a data model that favors them, while determining which interrogation can take more time.

2.10 Aeneas

An essential part of the effort in all the HPC/NoSQL applications we developed has been finding the right data model configuration. This work can be tedious, as it requires to set up the database, modify the application to store the data with a different data model, store the data in the database and then set benchmarks with different queries with a proper query distribution to understand which model performs best. Even understanding how and why a data model performs better can be challenging, as it entails gathering metrics information from several software stacks and machines. To simplify such a process, we developed Aeneas, A tool to Enable Applications to Effectively Use Non-relational Databases[30] which is freely available from the URL `https://github.com/cugni/aeneas`.

Figure 2.12 describes the overall Aeneas architecture that is composed of three main parts: a loader, a workload generator and a graphical web interface that allows to analyze and compare the results. The first component, the loader, takes as input XML files that follow a precise XSD structure and that allows to describe at high level the data model of the application - what we call the reference model - and then it allows to map the information into a databases layout, for example changing the value order or storing them in multiple tables. Once the data is loaded, or while it is loading, the workload generates a configurable workload using different types of statistical distribution, such as a uniform, a Gaussian or Zipf distribution. In the meantime, a daemon co-hosted application gathers metrics from both Cassandra and the OS and it stores them in the databases for further analysis. In a third moment, the user can access the data and visualize it from the graphical interface, using different plots and visualization techniques.

We used Aeneas to support the development of all our HPC applications. We also employed it to analyze the link between data models and their performance.

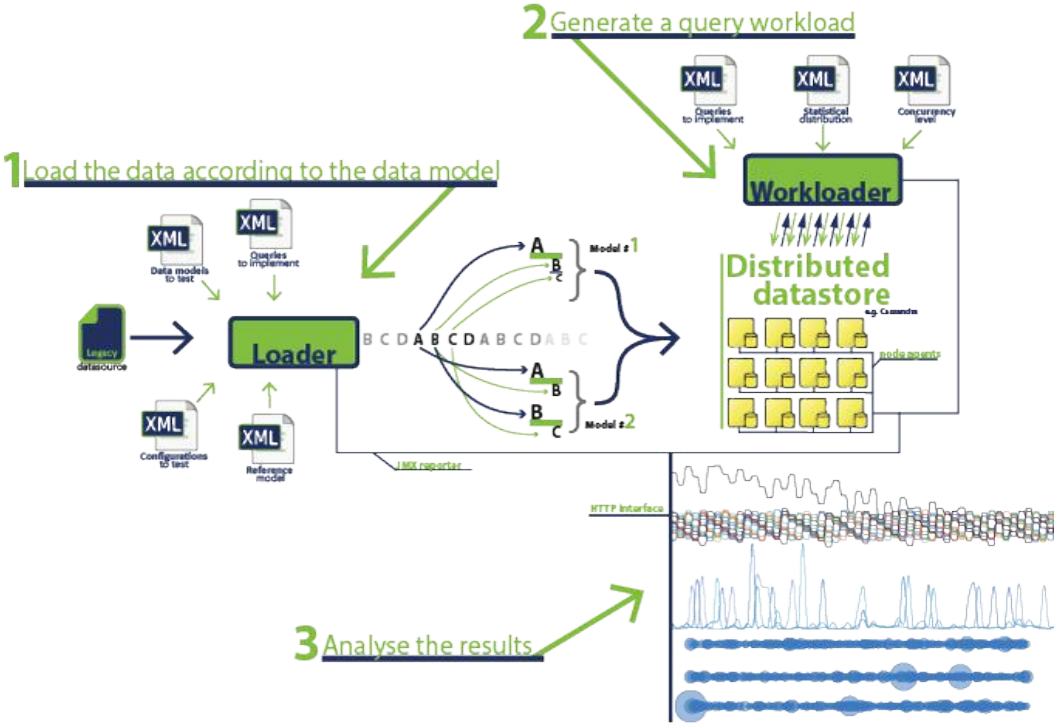


FIGURE 2.12: The Aeneas platform

Indeed, we published a study reporting the performance of 5 different data models for a scientific application [53]. Hernandez et al. [54] also proposed an Automatic Query Driven data modeling in Cassandra based on these experimental studies. Tai et al [104] recently proposed a similar approach. Replex is a multi-index data store that uses different layouts to save multi-keyed data. They also proposed a novel way to recover the information lost from a node crash by reading the data from another data model.

2.11 The analytical model

In our experiments, we often observed that the number of keys, nodes, and the hardware characteristics strongly influence the actual scalability of the system. Therefore, we developed an analytical model that allows finding the right system configuration to meet the desired performance for each kind of query type. In this section, we describe the math behind the random distribution of data and the experimental tests we conducted to estimate the performance behavior of a key-value database. The two parts put together assemble a model that can be used to find the right architecture for each distributed application thus greatly simplifying the developing effort. For example, each of the applications described in Section 2.1 has different requirements and characteristics, so we had to find the best architecture and configuration for each case. Therefore, we had first, to speculate on each one of the possible bottlenecks in our distributed system. Secondly, we had to create a prototype, run an

intensive test while profiling its performance. Finally, we checked if our assumptions were correct. If they were not, we had to iterate the whole processes. This method is time-consuming, and it had to be run multiple times, as long as each application tends to suffer from different problems. Also, in many cases, we have a trade-off between the various factors, thus it is difficult to find the right balance. A typical case is deciding when to use a master-slave or a peer-to-peer approach: a master with a centralized logic is easier to implement, but the capability of a single node might constrain the performance.

Likewise, we found a similar trade-off when assigning jobs to nodes: we can use pseudo-random policies, which are faster and do not need a master, but then we might have workload imbalance. Additionally, we can alleviate the imbalance by partitioning the work in more and smaller tasks or by replicating the information in more servers so that clients can pick the least loaded replica. Both cases have an overhead regarding CPU, storage or cache affinity.

How can we find a good balance between these aspects? Can a one-size-fits-all solution exist? Should a system that aims to few milliseconds response time have the same infrastructure of a batch-oriented one?

Therefore, we propose a benchmarking methodology that allows summarizing in a few metrics the behavior of distributed systems so that we can understand which are the limits of such a design. Also, we provide an analytical model that - once fed with the results of our benchmarking - can guide the improvement of the system architecture by giving precious insights about which components of the system are or will be the bottleneck at any given level of parallelism. Thanks to this model, a developer can, in front of a set of technologies and SLAs, choose the right architecture for its system.

A well designed distributed system should clearly distinguish between business logic design and how the system distributes mainly for two reasons: reusability and separation of concerns. Indeed, we want to write code that works well for different applications; without requiring any algorithm modification. We prefer to entrust the underlying distributed platform to configure and optimise the algorithm's execution.

Following this principle, modern distributed BigData platforms, such as Apache Hadoop, Spark or PyCOMPSs, allow writing an algorithm as a chain of operations to perform on data, leaving the platform to decide where and how to execute each step. Similarly, NoSQL databases, abstract the distribution of the data among servers, relieving the client from choosing where to search for an item or how to handle failures. The developer is only required to describe the application data model, while the database takes care of uniformly distributing the load and preserving consistency.

However, the data modelling and the system behaviour are not entirely orthogonal [30][53]. For instance, in Distributed Hash Table (DHTs) databases, the cardinality of the key has a significant influence on how the workload splits among servers.

Indeed, each distinct key randomly maps to a server with the optimistic assumption that - with a "high enough" cardinality - requests will spread uniformly. In other words, we have the undesired situation where a business logic related decision prejudices the system performance.

We will study Apache Cassandra [71] which architecture is described in Section 2.8. Cassandra has two levels of indexing, one distributed built on the *Partition Key*, while the seconds, which index the *Clustering Keys*, are orderly stored in a single node. This double layer of indexing gives the user the possibility to decide which items sort and group together and which ones spread randomly among the cluster. For example, let's suppose we want to index each phone number in the world: we can choose to group each record by country - e.g. using the national prefix as the *Partition Key* - by city or, at the end of the spectrum, store individually each record. Each choice has its benefits regarding which kind of query can serve efficiently: while the last configuration allows accessing only single users, the others also permit reading and aggregating by country or city respectively.

In the first case, we will have around 200 keys: one for each country. In the second one, we can estimate about one million keys while something of the order of the billions for the single user indexing. Now the question is: if we want to store all this data in ten servers, will it spread uniformly? We can use Formula 2.1 to estimate with high-probability how many more keys - in proportion - will go in the most loaded node. The formula, here briefly introduced, has a wider description in Section 2.2.6.

$$p \approx \sqrt{\frac{\log n * n}{m}} \tag{2.1}$$

Where m is the number of keys and n the number of nodes. With Formula 2.1 we can estimate for the first case that one of the ten nodes will have 27 countries assigned- which is about $\sqrt{\frac{\log 10*10}{200}} = 0.339 \approx 34\%$ more of what would have been a perfect distribution. In the two other cases, as the imbalance decreases with the number of keys, we will expect an unbalance of 0.5% and 0.015%. In the first situation the imbalance problem is evident, but in the second one, when grouping by city, we can encounter a similar problem. Even though the cardinality is high enough to distribute the cities uniformly among servers, some cities are much bigger than others. As a matter of fact, about half of the population lives in the 500 most populated cities, so with such a layout half of the queries would have an unbalanced distribution: we can expect to have one node with 21% more load than average. Even worst, doubling the server increases the imbalance to 35%.

Obviously, in this simple example, we could just redistribute the data since we know which grouped elements will have more load. But what if we do not have such information? Or if the popularity of items rapidly changes over time? With this work we aim to study this particular problem, providing a methodology and a mathematical framework that allows evaluating beforehand the performance consequence of any particular data model design.

2.11.1 Methodology

The methodology we propose has two different phases: the first one is composed of several steps and it is based on performing a broad range of tests to analyze how different configurations influence the system; the second one consists on defining a statistical model to guide future application designs. In the following subsections we describe the steps that we have performed.

Scalability analysis and data model influence. The first step in our methodology is to analyze how the data model affects the trade-off between a more balanced workload and a higher job fragmentation and thus influences the performance and scalability of the application. As Formula 2.1 states the expected workload imbalance in a DHT system is inversely proportional to the root of the number of keys. In simpler words, the more keys, the more uniform workload the node will have. On the other hand, more partitions result in more operations, messages, index entries and thus disk accesses.

To study this trade-off, we created a testing prototype to evaluate the behaviour of three data models. All three data models partition the dataset into blocks, which is the data unit to store in the database, and all of them differ in the number of elements per partition. Differences in the amount of blocks in each model are enough to affect the uniformity of the workload distribution and the number of operations necessary to perform a given query. In section 2.11.2 we describe the results of our experiment and we analyze the influence the data model has on the scalability of our case study.

Definition of stages As we will see in section 2.11.2, the work imbalance is not always enough to explain the lack of scalability, so further analysis is required. To understand which part of a distributed system is responsible for the lack of scalability it is necessary to study and to analyze in detail the application performance. However, we had to face two problems: first, the system is running asynchronously on multiple servers which means that we have to record, gather and correlate metrics from multiple sources. Second, as requests last less than a second, the common performance tools like Ganglia [43] are useless since they are designed to detect long-running phenomena. We found out that the best approach is to identify the primary data flow phases and to record the time that requests spend in each of them. This approach simplifies the identification of common patterns such as bottlenecks or workload imbalances.

Bottleneck identification and analysis Once we identified the critical phases in our application, we can analyze the timings by searching for possible architecture weaknesses. For example, if we observe that requests spend a considerable amount of time during phase **master-to-slave**, we can hypothesize we have a network problem. Similarly, if we see that requests spend significantly more time **in-queue** in one node, we might investigate on the workload imbalance.

Section 2.11.2 presents the results of the battery of experiments that we have performed to analyze each bottleneck and how the conclusions of our analysis allow us to improve the implementation and the deployment of our application.

The analytical model While the performance analysis allows improving the efficiency of the system, our final goal is to generalize our results into an analytical model which allows exploring the feasibility of new architectures. Indeed, with the analytical model, not only we can find the optimal trade-off between requests granularity and workload balance, but we can also simulate the performance characteristics of new and arbitrary complex architectures, giving us a valuable tool to research which is the best solution for each particular use case and platform. In section 2.11.5 we describe how we have defined the model for our case study.

2.11.2 Performance analysis

We have implemented a prototype to run on real data generated by the Alya simulator and indexed by our D8tree indexing system. In this prototype multiple nodes have to compute a simple aggregation — count by type — over one million elements grouped in blocks of various size. The test starts when a master asks the slaves to compute the aggregation over data stored locally. Each slave accesses only the local blocks using the underlying key-value data store. In this simplified prototype, the master knows from the beginning which are all the requests it has to issue. The dataset is composed of one million elements and we used three data models with different workload distribution characteristics. We named the three data workloads as follow:

coarse-grained: 100 partitions, of 10,000 elements each.

medium-grained: 1,000 partitions of 1,000 elements each.

fine-grained: 10,000 partitions of 100 hundred elements each.

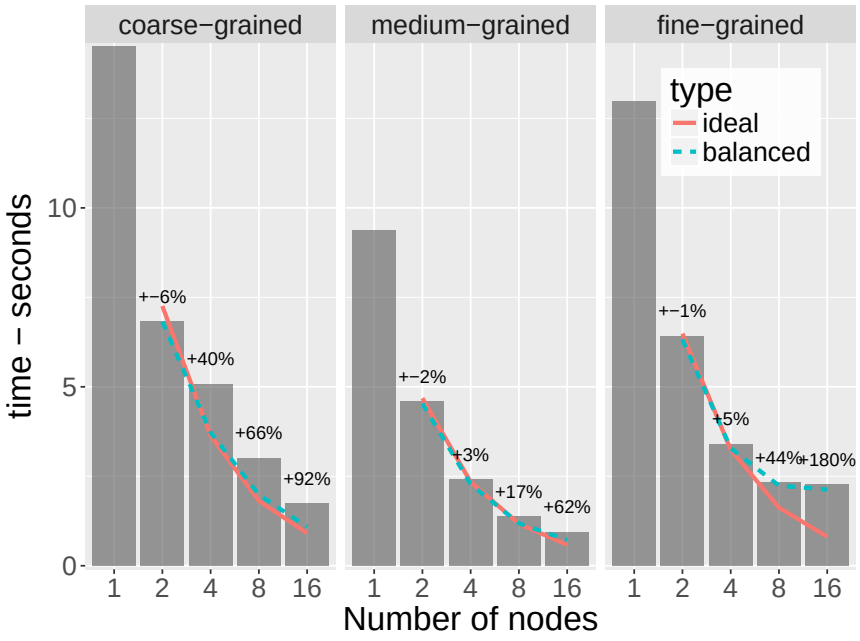
As the data is indexed with the D8tree system, the elements are stored in space partitions named **cubes**. We selected - in a pre-query phase - all the cubes with sizes that matched the three workloads. We picked at random cubes with one hundred, one thousand and ten thousand elements and we pre-computed the list of keys each workload has to read.

Our tests ran on on-premises servers equipped with two Intel Xeon Quad-Core L5630 with a maximum clock speed of 2.13GHz for a total of 8 cores and 16 threads and 24 GB of NUMA RAM. Each server had a SATA2 SSD and a rotational Hard disk and Intel Gigabit Ethernet as network interface. The three tests ran on the same database table. We run the tests on clusters of increasing sizes: 1, 2, 4, 8 and 16 nodes.

2.11.3 Influence of the workload distribution

In this section, we describe the first set of tests we ran on our prototype system, focusing on how the data model influences the scalability of the application. Fig-

FIGURE 2.13: Data model influence on scalability.



ure 2.13 shows how the three cases reveal different scaling profiles when doubling the number of servers in the system. The bar shows the time we observed; the solid line (labelled *ideal*) shows the query time we should have experienced if the query was scaling linearly while the dotted line (labelled *balanced*) shows the time we would have if the workload was distributed perfectly. The labels on the bars state the relative difference between the ideal and real times. The *balanced* line is calculated by counting how many queries each node served and then measuring the relative difference between the most loaded node and the average. Finally, we estimated how much time the query would have run if the load was distributed uniformly.

The first thing to notice is that none of the models scale perfectly, with a degradation that increases with the number of nodes. With 16 nodes, we observed times between 62% (**medium-grained**) and 180% (**fine-grained**) worst than an ideal scalability. In both **fine-grained** and **medium-grained** workloads the *balanced* and *ideal* lines overlap almost perfectly. This suggests that request imbalance between nodes is the primary cause of lack of scalability in these two workloads.

A different case is **fine-grained**, as the *balanced* line diverges from the *ideal* one. With 16 nodes, compensating the imbalance does not significantly counterbalance the lack of performance, so that the *ideal* line is 180% lower than the *balanced* one.

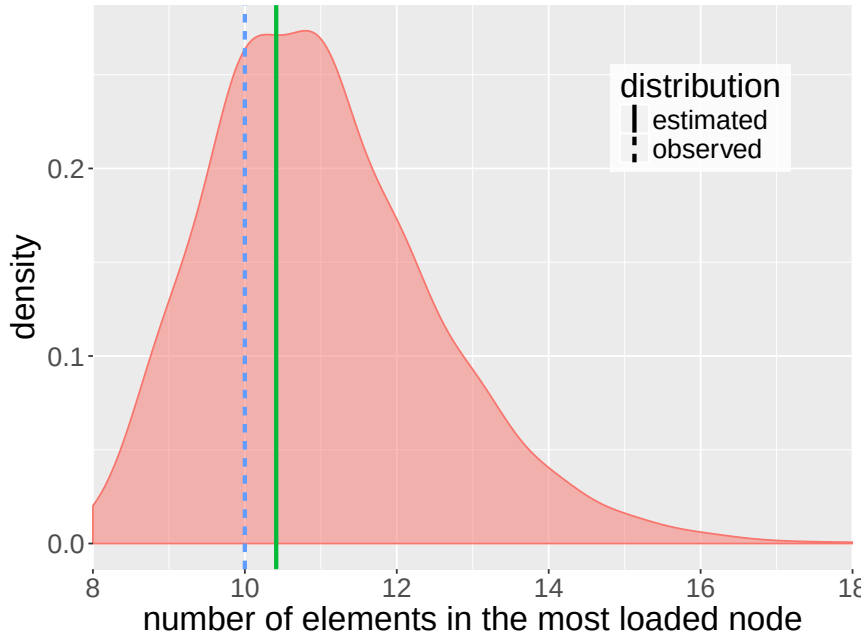
Since we ruled out the workload distribution as the cause of the performance degradation, we will later present a more detailed study. As confirmation of our hypothesis about the effect of work imbalance on the application scalability we analysed **coarse-grained**, which is the workload that splits the query into only one hundred keys, each of them containing ten thousand elements. Given the relatively small number of partitions, this policy shows the higher imbalance. As long as each key is stored at random in a server, the smaller is the ratio keys/nodes, the higher is the probability that some nodes will have more work than others.

FIGURE 2.14: Operations per node vs. sub-query time.

Figure 2.14 illustrates how the number of keys assigned to each node and thus, the number of operations that each node has to perform, affects the performance of the query. Figure 2.14 shows the results of the execution of **coarse-grained** workload on 16 nodes and it is composed of two charts: on the top a bar plot that shows the absolute number of requests each node served while the bottom graph shows the time required to complete each request on each node. All requests begin at the same time, so the query ends when the slower request completes.

At a first look, we can see how the two metrics are strongly correlated, but they are not identical. Indeed, we can see that the peaks in the number of operations match roughly the peaks in query time, but we cannot say the same for the lowest points. For example, node **G** completes 5 operations in almost the same time as node **F** finishes 2 of them when queries run in-memory on multiple cores CPU. Even though results show a considerable variance in all our tests, we observed that the node that served more requests is also the last to complete. Therefore, as long as the distributed operation completes when all nodes end their jobs, it is the slowest node

FIGURE 2.15: fine-grained: probability density with 16 nodes



to dictate the overall time. Summarising: **1)** the slowest node is the one that dominates the total time **2)** the slowest node is usually the one with the most queries **3)** the workload imbalance is proportional to the number of servers, and thus it explains why **coarse-grained** does not scale linearly. It also suggests that we can estimate this performance drawback using Formula 2.1.

Intuitively, it might seem wrong how the requests are distributed in figure 2.14. We were demanding 100 keys on 16 nodes so — in a perfectly balanced case — the most loaded node would have to serve $\lceil \frac{100}{16} \rceil = 7$ operations while in our case it served 10, which is 43% more. Therefore, we might wonder if this test is just a highly unfortunate case.

Figure 2.15 shows the probability density relative to the number of requests the most loaded node has to serve. We generated the graph with brute-force by distributing at random 100 keys between 16 nodes and recording how many keys fell in the most loaded node. Figure 2.15 shows the recurrences: the two vertical bars represent the imbalance we observed in the experiment — the blue line — while the green one is the value predicted by Formula 2.1. Figure 2.15 shows that our observation was not particularly unfortunate. On the contrary, in 60% of the cases we would have a more unbalanced scenario.

The central part of Figure 2.13 shows how **medium-grained** has lower imbalance as it uses ten times more rows. For example, with 16 nodes we reduced the overhead from 108% to only 44% compared to policy **coarse-grained**.

Model **fine-grained** shows a distinct behaviour. As we issue ten thousand queries at the same time we would expect an almost homogeneous workload distribution, but the system stops scaling with more than 8 nodes.

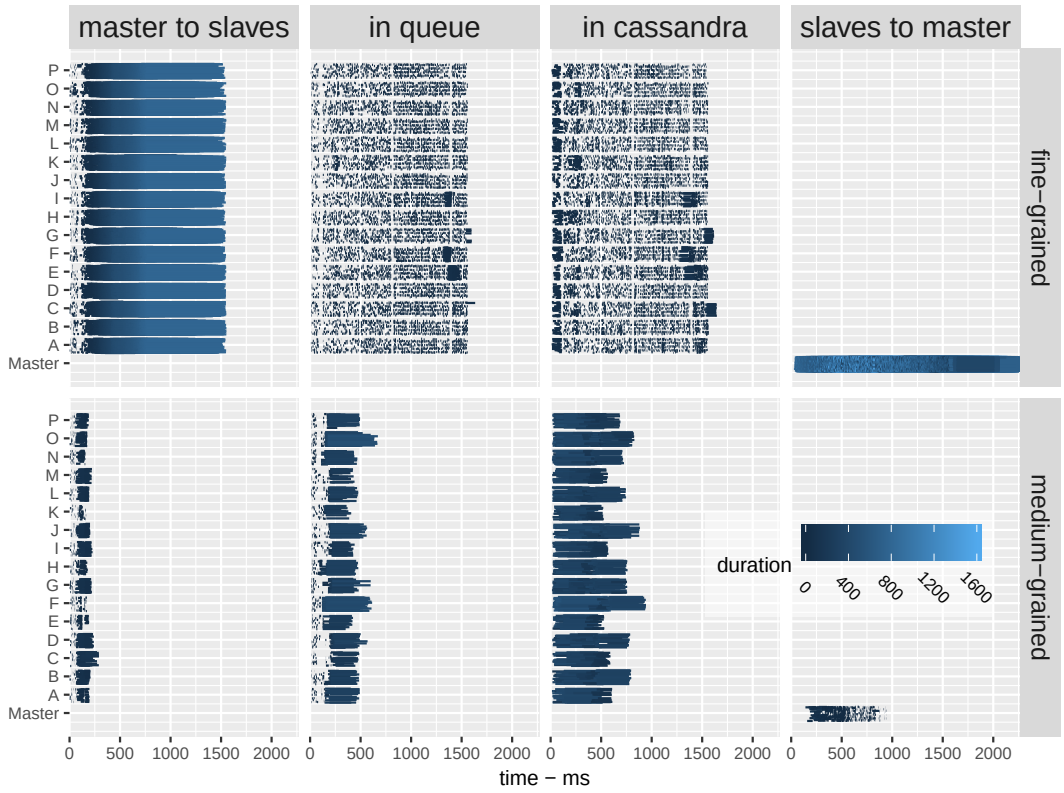


FIGURE 2.16: Profile patterns: **medium-grained** and **fine-grained**

2.11.4 Definition of stages and identification of the bottlenecks

To investigate the origins of the unexpected lack of scalability on **fine-grained**, we have analysed the code and the prototype architecture to define the main stages in a request execution and then we recorded the times spent by requests in each phase. We identified the critical points of a request execution such as the ones where the system interacts with different software or hardware layer or with remote machines. We identified the following stages:

1) master-to-slaves: the time between the master issues a request and when one slave receives it. **2) in-queue**: the time a query waits in a slave before it is sent to the database. **3) in-cassandra**: the phase where the request is sent, processed and returned by the database. **4) slaves-to-master**: the time for the master to receive a partial result sent by a slave. This four-phase analysis turned out to be crucial for understanding the overall system performance: it allows pointing out which components of a distributed system limits the performance.

Figure 2.16 shows the duration of each request in each phase in the two executions: one with **medium-grained** — the bottom part — and **fine-grained** on the upper side. Each cluster of segments is a single phase in a specific node, while the length of the bars and their colour describe the time spent by a request in each phase. In such a way, short-lasting events which should be the norm in a well operating system are almost invisible thus highlighting eventual system congestions.

The bottom part shows **medium-grained**: we can see that the **master-to-slaves**

phase lasts at most ≈ 300 ms, and also that Cassandra is not fast enough to satisfy all of the requests as quickly as they arrive. This is evident by looking at the **in-queue** phase where a lot of requests spend a considerable time waiting. Finally, the plot clearly shows how the workload distribution between Cassandra's nodes - as we saw before - is not uniform, and also that it directly influences performance. Indeed, this workload executes in less than 1 second, which is approximately the time required by the slower slave node — F — to complete all its requests in the **in-cassandra** phase. Hence, we can deduce that Cassandra is the weak link in the chain in this scenario, but also that, we can achieve a significant performance improvement with a more uniform work distribution. On the other hand, **fine-grained** presents an entirely different pattern. In this case, the master requires up to 1.5 seconds to finish sending all requests: an extensive time that leaves Cassandra idle most of the time. We can see it in both the **in-queue** and the **in-cassandra** phase. The first one is empty, meaning that all requests spent practically no time in queue. In the second one, we can see several empty - white - spots. These spots are the proof that Cassandra was processing requests faster than our system was able to issue them. In other words, a consistent portion of the whole execution time was spent idle while waiting for the requests to reach Cassandra. Here the major system bottleneck is the master node: it simply cannot send messages fast enough to keep Cassandra's nodes working at their full capabilities.

It is intrinsic of the master-slave architecture to be limited by the master's capabilities at some point. However, we wanted to understand which was the limiting factor. At first, we investigated the network: our cluster network has a star architecture, so each node is directly connected to a switch that dispatches the packages between nodes. With such an architecture, we suspected that the query saturated the outgoing connection of the master: yet we measured that the outbound traffic was only 7.5 MB split in 15 thousand packets. We measured that such a transmission takes 7ms in our cluster, way less than the 1.5s we observed.
Once excluded the network, we carried out a detailed profiling of the master application. We found out that we were hitting the CPU bottlenecks: we were consuming too many CPU cycles for each message. The CPU net cost of sending a message is the combination of several aspects influencing the actual implementation, such as the programming language and model, as well as the platform and libraries used. When building our prototype, we aimed for a good balance between speed of development and performance, so we used an Actor based library which runs on the Java Virtual Machine platform. Using profiling techniques, we found out that the there are two major contributors to the high message CPU cost. The first, and the most influential one, was the messages serialization. In Java, messages are objects and thus we have to transcode them into their binary representation — the serialization — to send them over the network. The default Java serialization implementation focuses more on flexibility than performance: it allows serializing at runtime any object, at the cost of adding extra meta-data into each object's byte representation.

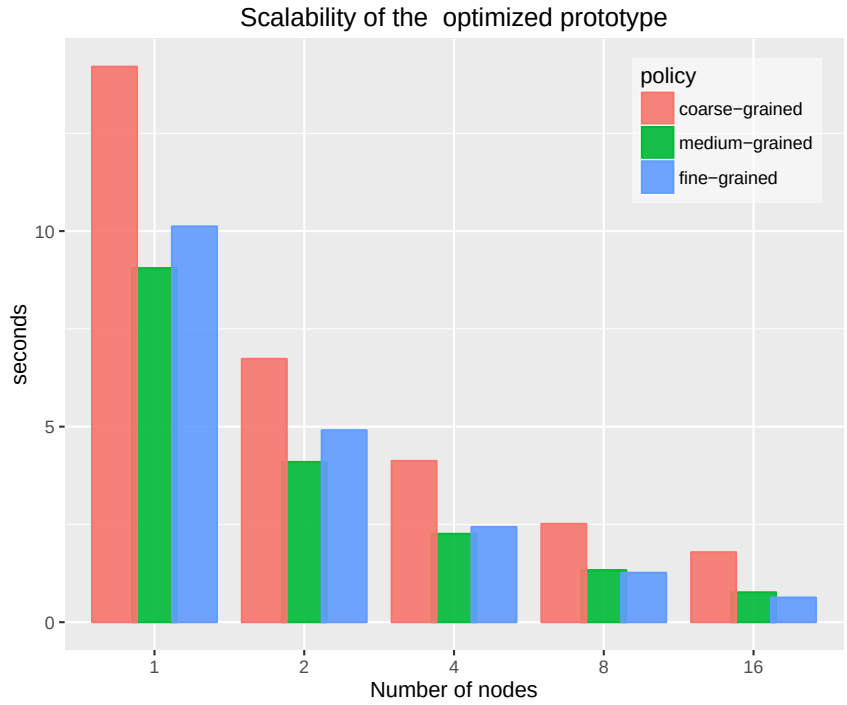


FIGURE 2.17: Performance reducing bottlenecks

In our profiling, we measured that the serialization phase took about 400ms of the whole execution. Serialization is a well-known issue of many distributed platforms running in the JVM, and thus, there are several alternatives which aim to reduce both the amount of bytes size and the CPU cycles required. Among many, we choose Kryo [69], a library that allows to reduce consistently both the bytes and the CPU required, by explicitly declaring which classes need serialization. The second optimization took place in our prototype code, where we observed that usually inexpensive operations, such as logging and integrity checks, were too costly at such a frequency so we had to work to reduce their effect.
With such optimizations, the master node of our distributed application had a great improvement, and the master node changed from sending ten thousand messages from 1.5s to just 192ms. Breaking it down to the single message, it is moving from 150 to 19 microseconds each, almost one order of magnitude of difference. Also, with a more efficient serialization, the amount of data transferred in the master to the slaves lowered to 900KB, which travels over the network in approx 700 microseconds. As a natural consequence of such an improvement, the results of our tests changed, especially for the **fine-grained** workload, which with the highest number of keys, was the more penalized by the message overhead. Figure 2.17 shows how performance changed.

The comparison of Figure 2.17 and Figure 2.13 shows how improving the master changed the performance profile of the various models: for instance now **fine-grained** shows almost linear scalability and it became the fastest workload when

running on 4 nodes and more. Indeed, **fine-grained** is 12% slower than **medium-grained** on a single node for the higher overhead of issuing ten times more queries, but this handicap is soon compensated when the number of nodes increases. For example, with 8 nodes **medium-grained** has an imbalance of 16% while it is only 4% for **fine-grained**: a delta that nullifies the initial 12% handicap. It is interesting to see how, even in this simple case, a one-size-fit-all solution does not exists and depending on the number of nodes we might prefer one configuration rather than another.

2.11.5 Performance Modelling

The last step in our methodology is to synthesize the results of the performance analysis into an analytical model that can guide designers to select the most suitable data organization for their applications. Thus, we created an approximated model for each of the different components that play a major role in a distributed application. So far, we observed three principal aspects: 1) the time the master needs to send all requests 2) the time that the slowest slave takes to finish 3) the time the master needs to fetch all results.

We saw that the request granularity - the number and the size of partitions in which the whole job is split - influences the performance of most of those aspects. Therefore, for each aspect, we created a regression model function of the number of partitions. We built these models upon observation recorded during tests run on our hardware and software stack. While the specific regression models may be realistic only for some hardware/software settings, the overall model and methodology can be applied to any system: it would simply require to run the same tests on the different hardware/software stack and create a new regression.

We saw in the previous experiments that the behaviour of the distributed system was influenced by the slowest of the high-level factor: the speed of the master sending requests, and the time required by the slowest slave to fulfill them. Therefore, at the highest level we can synthesize the model as:

$$max\{master_{speed}, slave_{slowest}, result_{fetching}\} \tag{2.2}$$

The $master_{speed}$ is the time the master node needs to send a single request to one slave. As we saw in Figure 2.16, the master can be the major bottleneck in performance. We have been able to speed-up the master by optimizing its code implementation, but this is not possible in all situations. For example, if the master has to compute expensive operations to issue each request, or if there is a dependency between them. For instance, navigating through an index, the master needs to examine the content of each call before deciding which are the next elements to read. In both cases, it is beneficial to understand in a designing phase how much time the master can spend in such operations so that a developer can determine which are the lower and upper bounds when adopting a master-slave or peer-to-peer approach.

In this thesis, we focused on the simpler case in which the master knows all the keys to visit from the beginning. Therefore its behaviour can be easily modelled as:

$$master_{speed} = keys * time_{msg} \tag{2.3}$$

In Formula 2.3, the keys are the number of partitions in the system, while the $time_{msg}$ represents the time spent, end to end, between the moment a request is sent and received.
We saw in Figure 2.14 that the slowest node is - unsurprisingly - the one that has to complete more requests. For this reason, to simulate the slowest node, we have to consider how many operations the most loaded node will have to perform, and then estimate the time required by the database to fulfill them. Putting all together, the $slave_{slowest}$ model must take into account: 1) the workload distribution between nodes 2) the time required by the database to compute a request. These aspects result in the following formula:

$$slowest_{slave} = key_{max} * DB_{model} \tag{2.4}$$

where key_{max} the maximum number of requests a node is expected to receive, and DB_{model} is the time the database requires to serve them. We can deduct key_{max} from Formula 2.1, and therefore:

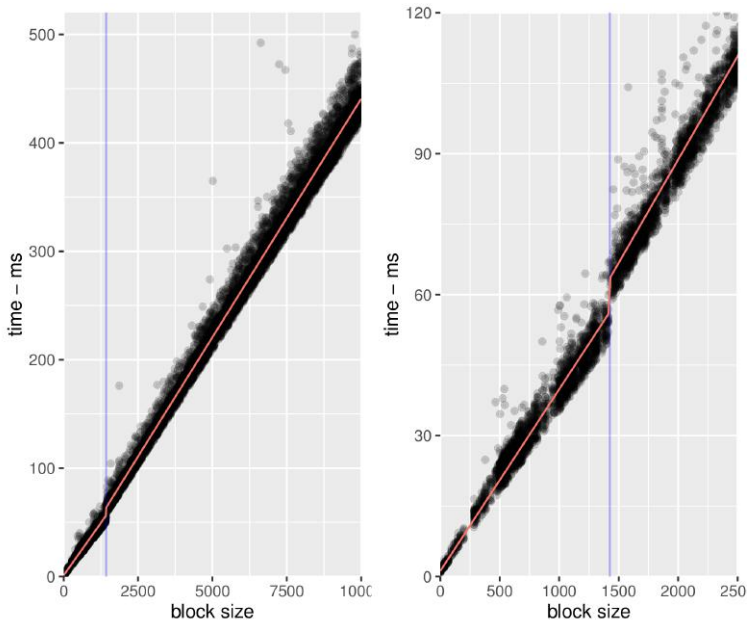
$$key_{max} = \frac{keys}{n_{slaves}} + \sqrt{\frac{keys * log(n_{slaves})}{n_{slaves}}} \tag{2.5}$$

As we discussed before, this formula is influenced by the number of nodes and the number of keys - partitions - in a way that promotes the increasing number of keys when we have more nodes. However, splitting a job into too many smaller partitions has a performance drawback causing overhead in the database.

Database model

The design of databases aims to reduce the latency of the average query by adopting greedy strategies; caches, indexes and bloom filters; that minimize the duration of most of the requests at the cost of introducing variance. For example, a miss in a cache of a false positive in a bloom filter can arbitrarily make a request orders of magnitude slower than average. Also, databases optimize the hardware resources by executing multiple operations at the same time, but this introduces a performance degradation caused by the interference accessing shared resources. For such reasons, we found out that the best way to model the database was first to study how the time required to serve a single request varies in relation to its size. In a second step, we estimated the performance degradation caused by the interface of concurrent requests.

FIGURE 2.18: Response time versus row size.



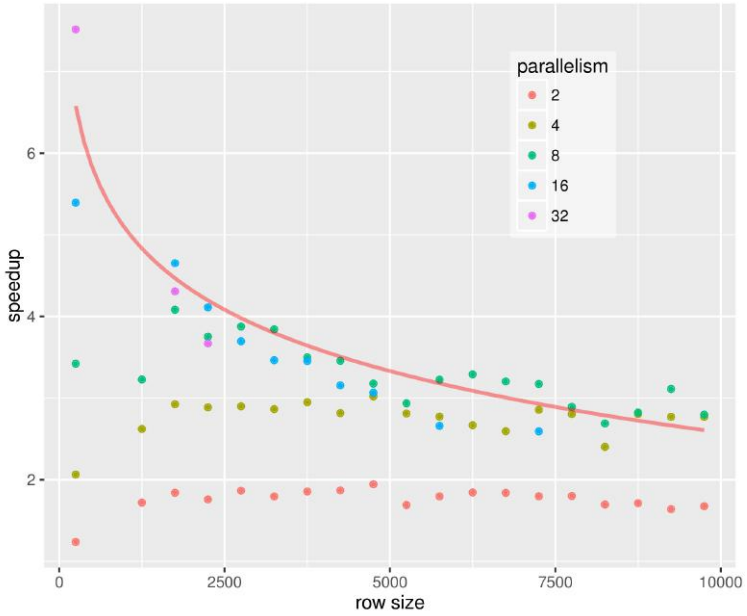
To build the DB_{model}, we made a stratified sampling of the rows in our dataset so that we could get the same number of random samples for each range of row size. Then we execute several repetitions of our test reading in random order the rows we selected previously. Figure 2.18 shows how the query response time changes related to the query size.

Figure 2.18 shows two plots: the first describes the whole test execution, while the right plot is a close up that shows only the requests with up to 2500 elements. The close up shows an unusual pattern in the Cassandra response time: at around 1425 items per row there is a discontinuity point. We found out that a Cassandra internal parameter *-column_index_size_in_kb* - was the cause of such behaviour. As Cassandra uses two-level indexing, it maintains for each row a column index but, as it is not efficient to index each entry, it records only the first, and the last column each *column_index_size_in_kb*, so rows smaller than 64KB are not indexed. As it turned out, 1425 rows are approximately 64KB and thus the index overhead caused such inconsistency. For such a reason, in Formula 2.6, we opted for a piecewise function:

$$query_{time} = \begin{cases} 0.773 + 0.0439 * key_{size} & \text{if } key_{size} > 1425 \\ 1.163 + 0.0387 * key_{size} & \text{otherwise} \end{cases} \tag{2.6}$$

We repeated the tests allowing different numbers of concurrent requests. Increasing the parallelism has an adverse influence on the system variance and the performance of the single queries, but it increases the overall throughput. We observed that the increase of the throughput is not constant, and it degrades in correlation with the row size. Cassandra seems to be able to perform at a higher parallelism with smaller queries and thus, to estimate the parallelism speed-up, we also have

FIGURE 2.19: Speed-up of parallel queries.

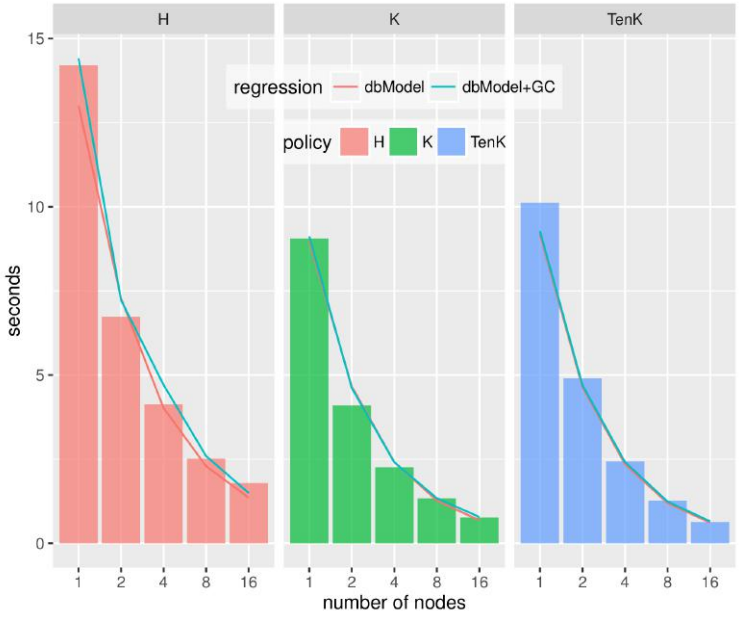


to consider which is the optimal parallelism for such a row size. To figure this out, we formulated another test: we created another stratified sampling of 20 groups, each of them with a row size range of 500 elements. For example, the first group has keys with sizes one to five hundred, the second from five hundred to one thousand, and so on up to ten thousand items per row. We queried all keys, for each group at a time, testing different levels of parallelism. Finally, we computed the maximum speed-up we achieved compared to the time required to get an element at a time. Figure 2.19 shows the speed-up we achieve raising the parallelism correlated with the query size. The colour of the dot represents the level of parallelism. The graph shows two general trends. Firstly, the larger the queries are, the lower is the degree of parallelism that performs better: The small queries perform best with 32 requests at a time, the medium with 16 while the large ones with 8. Secondly, the red line in Figure 2.19 shows that we can get a good approximation of the parallelism speed up as long as it shows a logarithmic proportionality with the row size. Formula 2.7 expresses the relationship between query size and the speed-up obtainable by running queries in parallel. Figure 2.19 shows the maximum speed-up we could achieve by raising the parallelism correlated with the query size. As you can see, the two dimensions have a logarithmic relationship that we expressed into Formula 2.7.

$$parallelism_{model} = 12.562 - 1.084 * log(key_{size}) \tag{2.7}$$

Finally, putting together Formula 2.6 and Formula 2.7 we can define the DB_{model} as shown in Formula 2.8. The formulas allow modeling the database throughput in relation to the size of the key.

FIGURE 2.20: Observed versus predicted time.



$$DB_{model} = \frac{query_{time}}{parallelism_{model}} \tag{2.8}$$

Validation

We validated our model by comparing the estimated times with the one we recorded in our previous tests. In Figure 2.20 the bars show the times we measured while the two lines show the values we estimated with our model.

The precision of the estimation is high, especially considering the high variance we observed in the tests. The only correction we had to carry out was for policy **coarse-grain** to compensate the overhead caused by the Java Garbage Collector, which our model does not contemplate as long as its influence is negligible in a properly configured system. Figure 2.20 also shows the line **dbModel+GC**, which adds the GC time into the model, notably increasing the model accuracy.

2.12 Model analysis

The flexibility of Formula 2.2 allows us to get useful insight on many different questions. For example, we can use an optimizer to find which would be the best number of rows for the query we run.

Figure 2.21 shows which would be the optimal time we could get on our system with the correct number of data partitions. It is interesting to see how the optimizer increases the number of rows when there are more nodes. Cassandra seems to perform at best if we split the one million elements into $\approx$ 3300 rows. However the

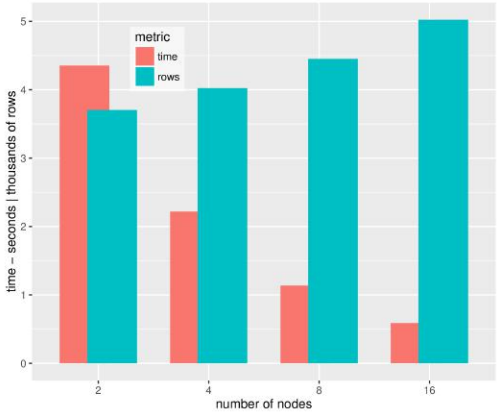


FIGURE 2.21: Optimal number of rows and the predicted time.

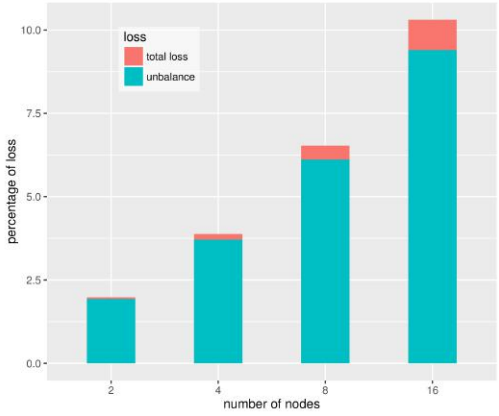


FIGURE 2.22: Optimal settings versus ideal scalability.

optimizer is willing to sacrifice some of the database efficiency in exchange for a better work distribution when adding more nodes. It means that we have to mediate between two conflicting aspects: the database efficiency and the workload distribution.

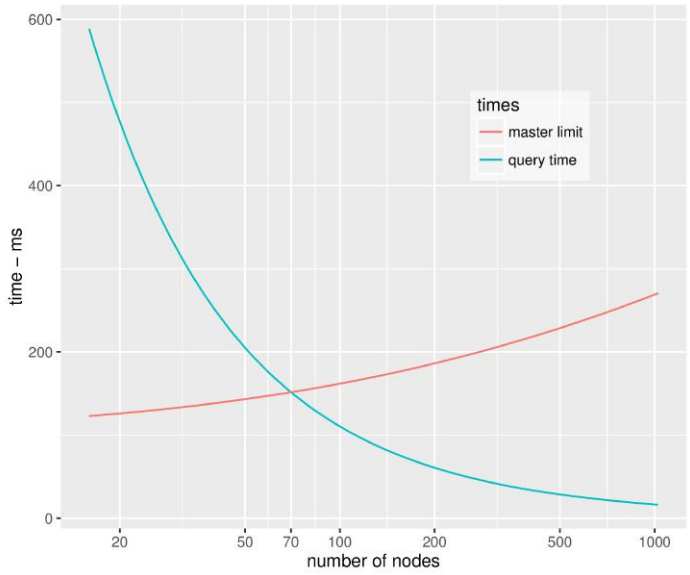
Figure 2.22 shows in percentage how much more time it takes the query to run on multiple nodes compared to an ideal linear scalability. We can see that even finding the optimal configuration parameters; we still have a consistent loss. For example, with 16 nodes the query requires $\approx 10\%$ more of what would have been necessary with a distributed workload.

Also, Figure 2.22 shows the difference between the total amount of loss and the fraction caused by the imbalance increase. This difference quantifies how much database efficiency the optimizer sacrificed to improve the performance. We can look at these results from another perspective. Let's suppose we are replicating the data in multiple nodes and that the master employs a replica selection algorithm so that it can ensure a balanced workload. For simplicity let's round up the numbers: the database performs optimally when issuing 4 thousand rows; the whole query takes 8 seconds on a single node, while the single request takes 11 milliseconds if we are issuing 16 queries in parallel per node.

On a cluster of 32 nodes, the query should run in $\frac{8}{32} = 0.25$ seconds if the system scales perfectly. To do so, the algorithm should be able to issue at the very first moment $16 * 32 = 512$ requests, and then continue issuing the same number of requests every 11 milliseconds. However, as we saw before in our prototype, sending a message takes about 19 microseconds, and thus sending 512 of them takes $19 * 10^{-9} * 512 = 9.7ms$, leaving almost no time for the algorithm to run. As a consequence, the time left for the replica selection algorithm reduces so much that it is likely that with more than 32 nodes the master will start to be the major performance bottleneck, and the system stops to scale.

We touch similar limits when distributing the requests at random. Figure 2.23 shows how the query time decreases by adding nodes. It also demonstrates that

FIGURE 2.23: Load distribution limits for a single master.



with more than 70 servers, the master requires more time to send the requests than the time the database would need to serve them. This limit is higher if compared to the previous case with the replica selection algorithm, and it is so for two main reasons. Firstly, the master has a simpler logic, so we can hypothesize it issues all requests at the beginning of the query. Secondly, with a random distribution, the system does not scale perfectly, consequently leaving more time for the operations of the master.

2.13 Summary

In this Chapter, we first analyzed the requirement of scientific HPC application, and we provided a broad view of the existing distributed data storage solution available in the market. Above the many alternatives, we motivated our decision to use key-value databases in HPC and to extend their capability by supporting MIS. Finally, we presented our work on understanding which are the major aspects to consider to ensure the scalability of a distributed application running on a key-value database. We created a benchmarking framework to compare different configurations, and we presented our research on the obtained results. Also, we proposed an analytical model that enables developers to estimate the influence of each part of a distributed system on the overall performance, so that it is easier to find the right balance for each application requirements. The model allows finding the perfect number of partitions in which to split a distributed query so that we get the right trade-off between database efficiency and workload distribution.

Finally, thanks to our module, we speculated, and estimate at which point either the master-slave approach or the replica selection algorithm can limit the performance.

The knowledge gathered in this Chapter is used in the design of both our novel indexing algorithms, the D8tree described in Chapter 3, and the AOTree in Chapter 4. In particular, in Chapter 4 we will use the analysis we have presented to study how we can improve the performance of the D8tree. Our model will be used to understand up to which level it makes sense to replicate data in order to gain performance on a distributed database system.

2.14 List of publications

Cugnasco, C., Becerra, Y., Torres, J., & Ayguadé, E. (2017, August). Exploiting key-value data stores scalability for HPC. In Parallel Processing Workshops (ICPPW), 2017 46th International Conference on (pp. 85-94). IEEE.

Cugnasco, C., Hernandez, R., Becerra Fontal, Y., Torres Viñals, J., & Ayguadé Parra, E. (2013). Aeneas: A tool to enable applications to effectively use non-relational databases. In Procedia computer science, Vol. 18, 2013 (pp. 2561-2564). Elsevier.

Collaborations:

Hernandez, R., Cugnasco, C., Becerra, Y., Torres, J., & Ayguadé, E. (2015, March). Experiences of using Cassandra for molecular dynamics simulations. In Parallel, Distributed and Network-Based Processing (PDP), 2015 23rd Euromicro International Conference on (pp. 288-295). IEEE.

Hospital, A., Andrio, P., Cugnasco, C., Codo, L., Becerra, Y., Dans, P. D., ... & Gelpí, J. L. (2015). BIGNASim: a NoSQL database structure and analysis portal for nucleic acids simulation data. Nucleic acids research, 44(D1), D272-D278.

Chapter 3

The D8tree: a read-optimized MIS

This Chapter contains the second contribution of this thesis and it introduces the D8tree, our first novel read-optimized distributed MIS algorithm.

As discussed in Chapter 2, the shift to more parallel and distributed computer architectures coincides with the rise of NoSQL databases that offer a scalable and reliable solution for "Big Data". However, none of them solves the problem of analyzing and visualizing multidimensional data. There are solutions for scaling analytic workloads, for creating distributed databases and for indexing multidimensional data, but there is no single solution that points to all three goals together.
In this chapter, we describe our first solution to the problem; The **D8-tree**, a Denormalized Octa-tree index that supports all three goals. It works with both analytical and data-thinning queries on multidimensional data ensuring, at the same time, low latency and a linear scalability.
We have implemented a D8-tree prototype, and we compared it with PostgreSQL, using the multidimensional plugin PostGIS, on a set of queries required by an in-house HPC application. We found the D8-tree outperforming PostgreSQL in all tested queries, with a performance gain up to 47 times.

The classic algorithms available in literature for multidimensional indexing (MI), such as R-trees or Quad-Trees (Sections 2.4.3, 2.4.1), were designed for optimizing the access to disk. They were designed to fit an environment where the major performance cost was the seek time when accessing data sectors. Thirty years ago, when A. Guttman firstly described the R-tree, reducing the number of accesses to disk was surely one of the most important factors to take into account to guarantee good performance. Since then, the technology background has inevitably changed. A modern database system runs on multiple servers, each of them having multiple CPUs. Moreover, rotational disks are being rapidly replaced by Solid State Disks (SSDs) which do not have any latency times when accessing non contiguous sectors. As a consequence, random access to disk sector does not imply any more performance drawbacks.

As we described in Section 2.3.2, NoSQL databases ensure high availabilty and scalability at the cost of reducing the database functionalities to a limited set of simple, but fast and scalable, operations. Consequentially, a user is forced either to

continue using legacy systems or to implement an ad-hoc middleware software.

We found ourselves in the described case when working on the Alya HPC application described in Section 2.1.2. The goal of our collaboration is to enable users to navigate interactively in the simulation. The interaction must be done so that a user can have the opportunity of increasing or decreasing at will the level of details visualized, and also the precision of the query results.
These requirements need a backend service able to handle 3-dimensional queries but, at the same time, keeping the response time in the order of a second; a limit that is necessary for ensuring a natural interaction with a human being.

We took into consideration several existing solutions, as well as novel algorithms proposed in literature, but we found none of them satisfactory for our requirements in terms of performance and scalability. Indeed, while there are many scalable indexing platforms supporting 2-dimensional Geo indexing - like Solr or ElasticSearch -, we found none of them fully supporting the more complex 3-dimensional case.

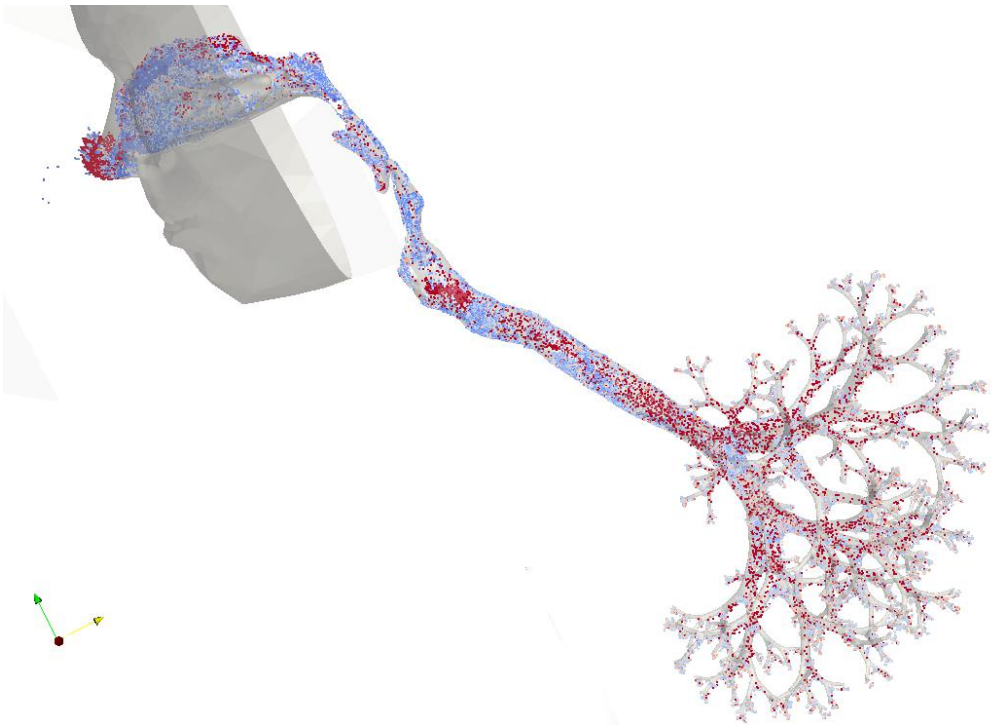
Hence, we decided to create a novel indexing algorithm, designed to be implemented on top of modern distributed key-value databases. Its design employs denormalization techniques to increase the scalability and the parallelism of multidimensional queries. It also allows to run efficient data-thinning queries, enabling the user to tackle massive data sets by analyzing smaller, but statistically relevant, subsets. We have implemented a prototype and benchmarked it to a state-of-the-art relational database such as PostgreSQL. Our solution has proven to execute faster on all the tested queries, being up to 47 times faster than the relational counterpart with a modest higher cost in terms of space.

3.1 Motivation

Our research started as a collaboration in a project that employs computational mechanics (see Chapter 2) to study how the particles flow through the human respiratory system. As shown in Figure 3.1, using Alya 2.1.2 and a partial model of the human body it has been possible to simulate how the particles are dragged into the bronchi during an inhalation. Of particular medical interest is the simulation of Lagrangian particles transported by fluids, that can be either medicine or pollutants in the air. In its original set up, Alya wrote the particles' properties in persistent storage. A typical simulation generates 300 Gb of plain text (CSV) data, but the amount of data stored is artificially limited due to performance considerations. We used two physical modules of Alya to carry out this work. The incompressible Navier-Stokes equations are solved in a sniff condition, using a mesh of 350M elements, while Lagrangian particles are transported by fluids, using drag law and Brownian motion [56, 57, 109].

The goal of the study is understanding how drugs are assimilated by inhalation, thus allowing to improve the existing medicine assimilation techniques. However, the simulation does not provide the result of a problem directly; it generates a huge

FIGURE 3.1: A screen-shot of the application



data set which has to be made available to a scientist for producing useful contributions.

Our collaboration aims to make the simulation easy to explore and visualize. Their size exceed the quantity of memory available in a single computer, thus we have to limit the number of elements visualized at the same time. Also, the user must be able to increase or reduce at will the number of particles visualized, to focus only on limited regions or to execute analytical queries. For example, narrowing the visualization in a particular part of the trachea, asking where the particles that were in an area have flown to, or knowing the percentage of the different types of particles in a given space, etc. Last but not least, all these requests must run in a time of the order of a second - if not less -, allowing a painless and responsive navigation of a simulation.

As we analyzed in Section 2.4.4, these requirements are not available with the existing solutions. There are solutions for scaling Big Data analysis, for creating distributed indexes and for indexing multidimensional data, but there is no single solution that points to all three goals together. Indeed, the first two are ineffective in the case of multidimensional data due to the lack of a universal order of elements in a vector space. At the same time, the existing MI algorithms cannot linearly scale since it is not possible to navigate through their structure with arbitrary level parallelism.

Our solution supports all these goals together: a framework allowing to execute both analytical and data-thinning queries on multidimensional data ensuring, at the same time, short response times and a linear scalability on multiple servers.

3.2 NoSQL characterization

The classical indexing algorithms, such as R-trees[50], were designed to take advantage of the hardware's characteristic at the time. A time where seek latency of the hard disks was the biggest performance bottleneck, RAM was extremely expensive, and databases ran on monolithic single core machines. The architecture of the computers has changed dramatically since then. The CPU's clock frequency growth has stopped, giving the way to new architectures built on multiple cores. Modern CPUs have also adopted a Non-Uniform Memory Access (NUMA) design, which distinguishes between the local and the remote RAM, opening new possibilities for performance improvement when applications embrace share-nothing parallelism.
As a consequence, the first step for designing a new indexing algorithm is to analyze what influences the performance of a modern key-value database. Even if different products may have different performance profiles, our opinion is that all the available products will show a similar behavior in this context.
In Chapter 2.9, we analyzed how the data model influenced the performances of an application while retrieving elements with a known identifier. In this scenario, the client was directly accessing the database.
Differently, in this case, as long as we are searching for data matching a multidimensional constraint, we cannot operate in the same way. We need to add a layer between Cassandra and the client which takes care of translating the client's requests into simple key-value Cassandra operations.
The main function of this new middle-layer is to implement the indexing algorithm optimizing the navigation through a distributed key-value system, instead of a sequential block device.
In this scenario, the partition and clustering keys are no longer the element's identifiers, instead they are the index pointers for organizing data into groups. For example, implementing an R-tree, the partition key can be the rectangle identifier, or in a quad-tree the partition's binary code. In any case, the indexing algorithm will navigate through these groups searching for the elements that satisfy a query.
There are two main aspects influencing performance: the size of the group, and the number of groups read in parallel. While the first can be configured simply as a parameter, it is not possible to do the same with the level of parallelism. Indeed, the chosen indexing algorithm strongly influences the maximum number of groups accessible at the same time. As long as multidimensional indexes resemble a tree, the fan out of each single node limits the number of requests executable in parallel. Intuitively, if each node has only two children, we will have less parallelism than if each node has eight children.

3.2.1 Influence of parallelism

We wanted to design an index that could maximize the benefits of running in parallel in a distributed environment. Thus, the very first step was to set up a simple

experiment to find out how parallelism influences performance. We inserted in Cassandra a sequence of numbers from 0 to 64 million splitting them into 65536 groups containing 1024 elements each. In this case, the groups identifier is a number between 0 and 65535. We have chosen to make groups of a size of 1024 to make them the same size of disk sectors. In our implementation, the group is a Cassandra row, where the group's id is the partition key and the numbers are the row values.
The testing client goes through all the groups asynchronously by issuing up to **p** requests at the same time, where p is a configurable parameter. For example, with p set to 8, the client issues 8 requests immediately and then it waits for the termination of at least one of them before issuing the following ones. The scope of the test is to see how **p** influences the time required to read the whole dataset.
We executed the test in four combinations of two variables: when the client resided either in the same node of Cassandra or in a remote node, and when the data required was already in the system cache or not.

The test ran on a premise server equipped with two Intel Xeon Quad-Core L5630 with a maximum clock speed of 2.13GHz. In total it counted 8 cores and 16 threads. Both CPUs mount 6 blocks of 4Gb DDR3 RAM memory for a total of 24 GB of RAM with an NUMA architecture. The server had a SATA2 SSD and Hard disk. The network interface was an Intel Gigabit.

FIGURE 3.2: Influence of parallelism

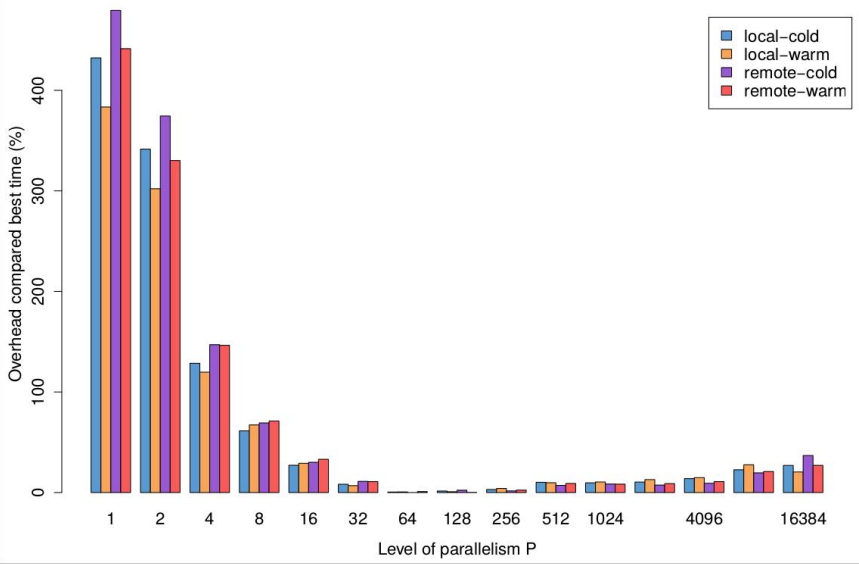


Figure 3.2 shows the performance reading the whole data set using different values of **p** and repeating the test ten times. It displays the median overhead of each value of p in comparison to the best one. We can see how the number of parallel requests dramatically influences performance. Indeed, between a simple configuration with **p** equal to 1 to the best one with p equal to 64, we have a speedup of more than 400%. We experienced the best results with **p** between 64 and 128. This result is coherent with Cassandra's documentation which states that when data fits in memory, as in our case, the optimal number of threads can be estimated multiplying by

eight the number of cores. Of course, this is a rule of thumb, but it seems pretty realistic in our tests.
It is interesting that performance, in all four configurations, increased almost linearly until reaching the optimal point between **p** of 64-128 but after performance decreases with a logarithmic trend. This suggests that, if an algorithm has to deviate from the optimal **p**, it is better to aim to a higher value of parallelism instead of lower ones. These considerations are fundamental for our solution: indeed, we can achieve high performances only designing an algorithm that can run at the same level of parallelism at which these databases work at best.

3.3 Our proposal: the D8-tree

In the previous tests, we have seen that we can achieve the best results when issuing between 64 and 128 queries per node at the same time. We also saw that this level of parallelism relates to the number of processors for each node and seeing the trend of computer architectures nowadays this is likely to increase in the future.
Furthermore, we want to design an index that can scale horizontally on multiple servers. Understanding how the load distributes among the nodes when randomly assigning tasks requires a more sophisticated analysis that will be part of our future work. We can, however, make a rough estimation: if a node performs optimally with **p** parallel requests, when we have **n** nodes, the optimal result will be somewhere close to $\mathbf{n} * \mathbf{p}$. For example, if we run a cluster with 8 nodes, we can expect to have the best performance when issuing overall between 512 and 1024 requests at the same time.
Hence, when executing any arbitrary multidimensional query we would like to have an index that allows us to execute about $\mathbf{n} * \mathbf{p}$ independent queries at the same time. However, choosing at will the level of parallelism is not something possible with the classical approaches where we are limited by the node fan-out and the query selectivity.
Let us take as an example the quad-tree, an index that divides the domain space into squares. If a square is full, it splits into four disjoint offspring partitions and the elements are distributed among them. Anytime a cube splits, it adds a level to the index. When executing a query we have to start from the root node, then descend to the children, then the grandchildren and so on. From each node, we can descend in parallel into the four children so that a query runs with a parallelism of 4 on the lower level, 16 in the following lower level, and so on descending.
However, the fan-out limit is only theoretical: a query might interest only few children of each node thus reducing, or eliminating completely, the parallelism. For example, if a query only needs the elements residing into a small domain area, the execution is likely to descend the whole tree considering only one child per each node thus making any parallelization impossible.
The previous example points out also another drawback. When descending the tree,

the algorithm has to navigate through several nodes and this, on a distributed system, is expensive and also a serious threat to the scalability of the whole system.

Finally, we considered the problem of thinned queries. Our research started from an application requiring to execute efficiently queries sampling a small subset of elements from a wide spatial area, as well as queries aiming to all the elements available in a narrow space. As we discussed in detail in Section 2.4.4, it is not possible to execute efficiently using a classical approach such as the R-tree.

We came up with a solution to all the previous problems that embrace the data de-normalization, a standard technique in databases to gain in performance at the cost of duplicating parts of the data. Our solution is straightforward: instead of having a Spatial tree for zooming and a Spatial indexing for retrieving the element, we merged these two structures mapping them directly into a key-value store such as Cassandra.
We will refer to our index as **D8-tree** (*Denormalized Octa-tree*).

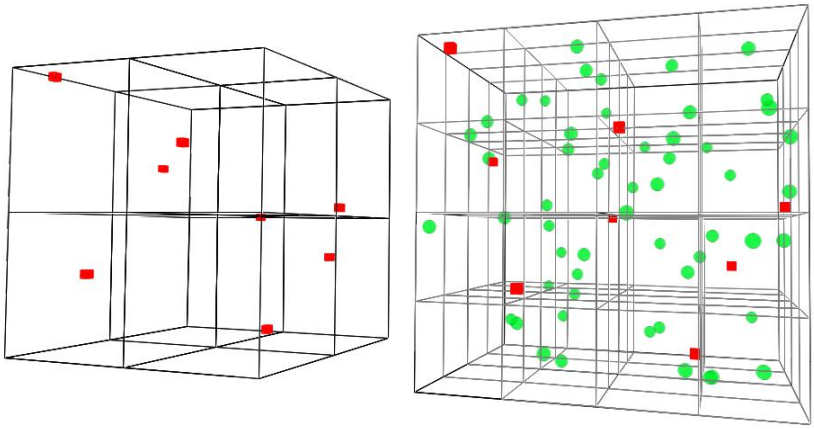
Definition 3.3.1 *A **D8-tree** is a balanced 8-ary rooted tree T(Z,K) composed of **Z** levels and $\frac{8^{1+Z}-1}{7}$ nodes where each node contains the TOP-K elements of the relative sub-domain space.*

Let N_j^i be the **i***th* node at the level $j \in [1, Z]$ where $i \in [1, 8^j]$ is the node identifier at the level *j*. Let e_n be an element stored in the D8-Tree and $priority(e_n)$ its (randomly) assigned priority. Finally, let $L_j = \{e_k : e_k \in N_j^i, i \in [1, 8^j]\}$ be the union of the elements present at the *j* level.
Then the following proprieties are valid:

1. The space domain of N_j^i is one partition over 8^j of the overall $\mathbb{R}^3$ space. Thus, $\forall j \in [0, Z] : \bigcup_{i=1}^{8^j} N_i^j = \mathbb{R}^3$
2. Each level L_j has 8^j nodes, and thus it contains up to $K * 8^j$ elements.
3. $L_i \subseteq L_j$, if $i \leq j$ and therefore if $e_n \in L_i$ then $e_n \in \{L_{i+1}, , , L_Z\}$
4. $\min\{priority(e_n) : e_n \in L_i\} \geq \min\{priority(e_n) : e_n \in L_j\}$, if $i \leq j$

In other words, the characteristic of the D8-tree is to replicate the elements with higher priority on the top levels of the tree so that the data contained within each level is also stored in the lower ones. Figure 3.3 shows a representation of the levels 1 and 2 of a D8-tree where each node can accommodate maximum one element. It is possible to see how the elements available at the first level, the red cubes, are also accessible at the second level of the tree. Figure 3.4 shows the structure of an Octa-tree where denser zones are more partitioned thus making data thinning more complex. Indeed, if we want to sample one point from the top-left quadrant, the one with the green pyramids and blue spheres, we should access each of the 15 partitions reading all the elements and then randomly select the one we need in memory. With the D8-tree, instead, we would need to read just one point from the first level of the tree as long as it is already a random sample of the underlying space area.

FIGURE 3.3: The first(left) and second level (right) of a D8-tree.



As a result, in the **D8-tree** an element can be replicated up to Z times. However, the Formula 3.1 demonstrates that, in case of elements uniformly distributed among the space, the amount of data duplicated is about one eighth of the overall data stored.

$$\lim_{Z\to\infty} \frac{\sum_{i=1}^{Z-1} K * 8^i}{\sum_{j=1}^{Z} K * 8^j} = \lim_{Z\to\infty} \frac{8^Z - 8}{8 * (8^Z - 1)} = \frac{1}{8} \tag{3.1}$$

Formula 4.1 converges very quickly to $\frac{1}{8}$, so if Z=5, the space overhead is already about 12.497...% .

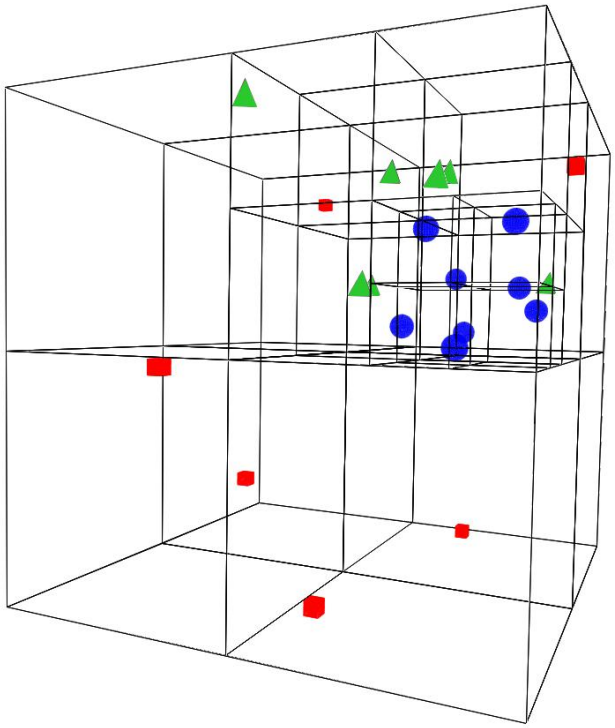
Obviously, the case of a data set uniformly distributed in space is unrealistic and in a real case the quantity of data replicated might exceed several times the size of the original data. However, we think the benefits exceed the drawbacks of storing duplicated data. We will also discuss further in Section 3.5, how we can modify the index to reduce repeated information.

The benefits of adopting de-normalization are several. First of all, if we want to visualize a spatial data set, and K equals the maximum number of elements visible at the same time, we can execute the data thinning queries extremely efficiently. Indeed, the query requires reading only the data from a single node and, most importantly, this is true at any zoom level. This was not possible with a common MI, as long as we would read all elements in the zone and then select K of them in memory (see Section 2.4.4 for more details).

Secondly, the interrogation of the index no longer requires passing through all the node's ancestors. Indeed, as long as all the data stored into the ancestor is also available in the offspring, a query can go directly to the smaller node that describes the desired space. For example, reading elements from a small portion of the domain, such as a space smaller than $\frac{1}{8}$ of R^3, we can jump directly to one of the nodes at the second level. Similarly, if smaller than $\frac{1}{64}$ to the ones of the third level and so on.

We can benefit from this property in two different situations. The first when

FIGURE 3.4: A Octa-tree structure



we read a small set of data, as long as we can access directly to it avoiding useless navigation through the index.
The second situation is when we need to read a considerable amount of data. In this case indeed, we can speed up the execution spreading the work on many computer nodes. For example, with a query about the data residing in the domain of node N_i^j, a client could decide to read directly to its 8 children or 64 grandchildren. This "**generational jump**", can lead to consistent performance improvements. Indeed, as we have seen in Section 3.2, executing in parallel 64 read requests results in up to 400% performance gain. In the same way, splitting the query in 512 nodes would result in an optimal hardware utilization using an 8-node cluster.

3.3.1 Index implementation

We have successfully implemented the D8-tree on Apache Cassandra using Apache Spark to create the index from a static data of 19GB of 3-dimensional particles trajectory. The indexing algorithm has been implemented using Apache Spark's programming model which is described is Section 5.4.2.

Listing 3.1 shows the algorithm implementation using Scala and Spark. We used a bottom-up recursive approach. The algorithm has 3 inputs: the data set, the total number of levels in the tree Z, and the maximum number of elements in node K. It starts with a set-up phase assigning to each element an arbitrary priority. Then, it designates each item to one of the nodes of the current level. If a node contains

more than K entries, it keeps only the first K and discards the others. The algorithm iterates up to the root level considering only the data that have not been discarded yet.
The algorithm uses a particular function to assign each element to a node. It returns node identifiers, which are a string of K symbol $\in [1, 8]$ composed in such a way that every node shares almost the same id of the father, and a part of the suffix of an additional symbol. For example, if a node identifier is the string "7342", its parent id will be "734". More precisely, we use a Z-curve to generate the node identifier using the binary algorithm proposed by F. Frisken and N. Perry [42].
Thanks to the property of the node id, we can handle nodes as bare strings. Indeed, at each iteration, the algorithm removes the last symbol from each id to assign the elements to the node's parent. To complete the level, the algorithm aggregates all the elements with the same node id, and lastly, it selects the first K elements falling into each node. The algorithm proceeds in such a way up to the root node.

We used a bottom-up approach because it fits better when spreading the work among different nodes. Indeed, an operation such as *groupBy* requires sending all elements with the same key to the same worker and, therefore, same node. With a top-down approach, our algorithm should start by aggregating all the data in the first node, thus resulting in poor performance and frequently in the crash of the whole system due to the overload. On the contrary, the bottom-up approach starts by spreading uniformly the load across the workers and then, when climbing the tree, it allows to reduce the working data to the only elements fitting in the previous level, thus alleviating the complexity of the algorithm.

We took a different approach implementing the query algorithm. Indeed, we decided to implement the code for each query in a separate application instead of directly executing them with Spark. This choice was driven by the need of more flexibility, as long as we could try different implementation details. The decision was also driven by the fact that many of the queries do not require to perform any aggregation on the data read, thus making unnecessary the usage of a system such as Spark.

3.4 Experiments

We carried out a benchmarking test between our solution and PostGIS, an extension PostgreSQL - the popular relational database - optimized for handling spatial and multidimensional data. We conducted the experiment on three queries considered vital for our application. These queries differ in the size of the space they delimit and, therefore, the number of results. They reproduce the usage pattern of a user that first analyses a simulation from a distance and then incrementally focuses on smaller areas.

LISTING 3.1: Simple D8-tree Scala indexing algorithm

```
val k: Int = 10000 // Max node size.
val z: Int = 10 // Max number of levels.
def D8tree(data:RDD[(String,(Float,Point))],
           level: Int) = {
  val survivors = data.map{
      case (key,(rand,point))=>
        (key.substring(0,level),(rand,point))
    }.groupBy{ case(key,(rand,point))=>key }
    .flatMap { case (cube, elements) =>
      elements.toList.sortBy{
        case (key, (rand, point)) => rand
      }//Select  the first K points.
      .take(k)
  }
  //If at the root level, it returns the
  // result, otherwise keeps  iterating.
  if(level>1){
    D8tree(survivors, level - 1)
    // Merging the results.
    .union(survivors)
  } else survivors
}
// Let's create the first keys: The function
// createKey returns the id of point's cube at
// level Z.
val lowerLevel = input.map(point =>
  ( createKey(point,z),
    (Random.nextFloat(), point) )
)
// Call the recursive function.
recurQuad(lowerLevel,level=z)
```

More precisely, **Query A**, which is the wider one, returns almost one million elements, **Query B** almost one-hundred thousands and the last one, **Query C**, returns about a thousand elements.

Query A and **B** return an amount of results which cannot be easily visualized, and thus we need to select only part of them. To do so, we will use two different data-thinning techniques that reproduce different objectives: "*LIMIT 1000*" to maximize the number of elements visualized, and "*TEN PERCENT*" to preserve the items' density. The first one simply sets a constraint about the maximum number of results, while the second limits the percentage of data to read. Either way, we must guarantee Zoom and Pan consistency, and we must be able to return the items in a random, but fixed, priority. For example, using the first policy on Query A ($\sim$ 1 million results), we will pick only the first thousand of elements sorted by priority. In the second way, we will select the elements with a priority between 0 and 0.1 (assuming $priority \in [0, 1)$) which should be about one hundred thousands elements. We also tested "*LIMIT 1000+*" which is an alternative implementation of the query "*LIMIT 1000*" in PostGIS.

We repeated all tests ten times and selected the median value to discard outliers in the result. At each repetition, the databases were restarted and the system cache was dropped. We used the same on-premise machine described in Section 3.2. Finally, we ran the test with two different computer configurations. In one we stored the data on a rotational disk while in the second in a solid state disk.

TABLE 3.1: Performance Speedup D8Tree vs PostGIS

	Query A		**Query B**		**Query C**	
	hdd	**ssd**	**hdd**	**ssd**	**hdd**	**ssd**
all	97	90	254	123	662	41
LIMIT 1000	4789	3388	1675	738		
LIMIT 1000 +	767	63	649	30		
TEN PERCENT	356	285	666	115		

Table 3.1 shows the relative speedup of the D8-tree for each query, each different storage device, and each query setting. In all our tests, the prototype implementation of the D8-tree built on Cassandra outperformed PostGIS by being between 30% and 47 times (4700%) faster. We tested both types of queries, approximated and not, and this lead to large speedup variances. Indeed, the D8-tree shows consistent performance, unlike PostGIS that is unable to execute efficiently the approximated queries thus causing the huge speedup difference.

The data thinning queries on an R-tree index can be implemented in two distinct ways. In the first one, we build a multidimensional index on only the data attributes (in our case x,y and z) and then we filter data in memory. The second option is to include the random priority in the MI. In our case, it means building a 4-dimensional R-tree. We have tested both implementations - "*LIMIT 1000*" and "*LIMIT 1000+*"- , as long as it is problematic to execute efficiently this kind of sampling in SQL. More

precisely, "*LIMIT 1000*" refers to the simplest, and often the only possible, approach: sorting all elements by priority and then selecting the first thousand ones. Not surprisingly, the relative performance is extremely poor: the database has to sort a considerable amount of data to then return only a small subset of elements. In such a way, reading all the elements or just the first ones requires almost the same time. For the sake of a more fair comparison, we implemented the "*LIMIT 1000+*" query by hypothesizing that the client knows previously which percentage of the data falls into the first one-thousands elements. In such a way, the client can execute a range query on the 4D-index so that it returns the first elements. Such an approach leads to an evident performance improvement, but in a real application, it is not trivial how the client could know in advance how the data distributes in space.
The D8-tree does not have this problem; the top levels store the items with a higher priority, and thus the algorithm can simply descend the tree as far as it has read enough data to comply with the request. The data retrieved is already sorted by priority and therefore the D8Tree algorithm does not need to know "*a priori*" how the data distributes, nor does it have to read and sort the whole data set. For this reason, the D8Tree can be up to 47 times faster than "*LIMIT 1000*", and still be between 30% and 700% faster then the "omniscient client" approach, "*LIMIT 1000+*".

FIGURE 3.5: HDD vs. SSD.

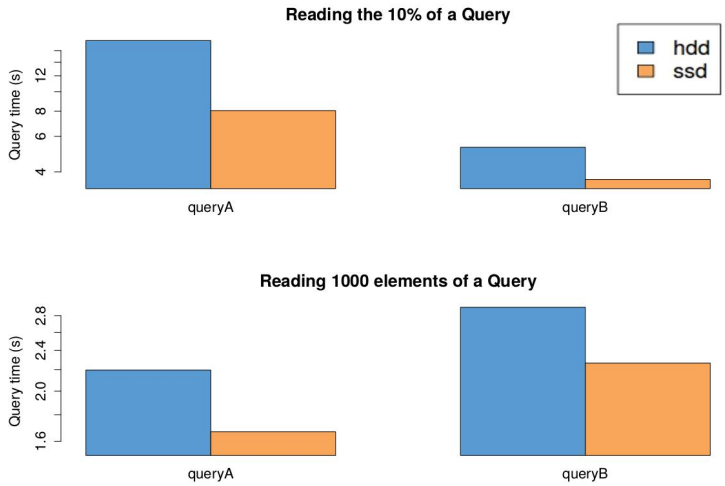


Figure 3.5 shows the time required to perform the two data thinning queries on Cassandra. You can see how, in both configurations, with HDD or SSD, the queries execute in few seconds, thus making possible an interactive exploration of the simulation by a human user. Furthermore, the Figure shows expected time in the worst case scenario: on a real application we experienced a much lower response time. Indeed, in our test the database runs on a single node and all requests need to access directly to disk, as long as we drop the system cache before execution. Table 3.1 also shows a considerable difference between the speedup of HDD and SSD. As Figure 3.5 shows, this difference is not caused by a drawback of the D8-tree on SSD;

on the contrary, it is proof that PostGIS executes a high number of random reads. Indeed, the two tested devices have similar absolute speed, and they differ in the latency required accessing different sectors.
It may not surprise that the D8-tree outperforms PostGIS in data thinning queries- as long as we created it for this scope- but it is interesting to see that it also works better for the "read all" kind of queries.

3.5 Data replication

The characteristic of our approach is to embrace data de-normalization which means replicating more times the elements to gain in performance. We have proved that this method is valid for the data thinning of multidimensional data, but now we have to evaluate the cost in terms of disk space. In Section 3.3, we have shown that the space overhead is about 12% when data distributes uniformly. However, in real applications data has a skewed distribution and it leads to higher space occupation. Indeed, for the previous test 3.4 we created an index of 10 levels with nodes that can contain up to ten thousand elements and with such a configuration, we experienced a duplication of the data of about four times. Even though it may seem a high cost, it is worthy to notice that in distributed databases it is a common practice to replicate data at least three times to ensure the availability of the information. Indeed, an interesting approach to the problem can be merging the duplication required by the index to the one needed to guarantee the availability of the data. In such a way, the cost of the index's replication would be virtually zero.
It is also important to notice that to improve PostGIS' performance we had to create two external multidimensional indexes - one of three dimensions and another of four dimensions - which did not necessarily result in less space occupation of the D8-tree. For example, for the previous tests, the D8-tree required 67GB of disk space, while PostGIS used about 50GB.

FIGURE 3.6: Percentage of elements at any level

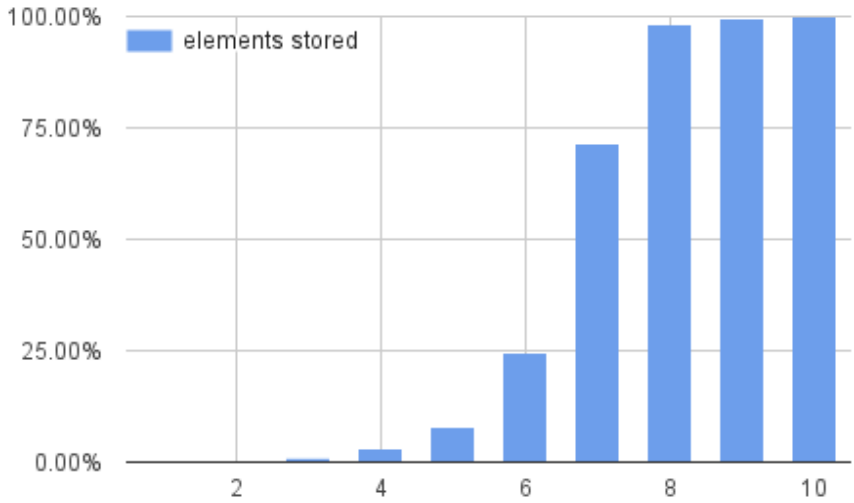


Figure 3.6 shows the percentage of elements available at any level. According to the D8-tree proprieties, the last level - the tenth - stores the whole dataset. It is interesting to see that the 8th and 9th levels already store respectively 98.4% and 99.7% of the data. It means that from the 8th level on, only very small parts of the tree nodes are full, thus adding further from this level causes a considerable high price in terms of replication. As a matter of fact, if we consider the only elements stored up to the 8th level, the D8Tree would have a replication of only factor two.
As part of our future work we plan to make a deeper study about data replication in indexes. However, the distribution shown in Figure 3.6 lets us believe that with minor modifications of the algorithm it is possible to reduce the burden of data replication without compromising performance.

3.6 Real-time D8tree indexing for HPC

In the previous section, we have described an alternative approach to index and analyze multidimensional data with a non-relational database. The novel algorithm, the D8tree, has proved to speed up query performance and to outperform the state-of-the-art available solutions when dealing with multidimensional data visualization of static data. Indeed, the D8Tree has shown to be up to 47 times faster than PostGIS serving data-thinning queries, and it has also proved to be substantially more rapid for the other kinds of requests examined. While these results show the feasibility and the quality of our approach for multidimensional approximate queries on static data. In this Section we will focus on the next step, which is extending the D8tree for real-time indexing. With a real-time scalable indexing system, we can improve the scientific workflow by allo wing scientist early access to the results, simplify the overall data management, and achieve better performance in many cases.

In this section, we continue our previous work by presenting the first version of Qbeast, a distributed, peer-to-peer system, able to build a D8tree under the intense I/O workload generated by an HPC physics simulation.

3.6.1 I/O for HPC applications

Large scale time-dependent particle simulations can generate massive amounts of data, making it so that storing the results is often the slowest phase and the primary time bottleneck of the simulation. Furthermore, analysing this amount of data with traditional tools has become increasingly challenging, and it is often virtually impossible to have a visual representation of the full set. In this Section, we propose a novel architecture that integrates an HPC-based multi-physics simulation code, a NoSQL database, and a data analysis and visualization application. The goals are two: On the one hand, we aim to speed up the simulations taking advantage of the scalability of key-value data stores, while at the same time enabling real-time approximated data visualization and interactive exploration. On the other hand, we want to make

it efficient to explore and analyze the large database of results produced. Therefore, this work represents a clear example of integrating High Performance Computing with High Performance Data Analytics. Our prototype proves the validity of our approach and shows great performance improvements. Indeed, we reduced by 67.5% the time to store the simulation while we made real-time queries run 52 times faster than alternative solutions.

High performance simulations can run on thousands of computers for several hours and generate massive quantities of data, such as the position and properties of particles at each time step. Large scale simulations track hundreds of millions of particles, and the size of the output files containing all this information can easily be in the order of terabytes.

Traditionally, simulation results are stored in one or more files in formats such as CSV, HDF5 or netCDF, but as described in Section 2.5, writing parallelly into a single file can be expensive as it often requires additional collective synchronization and communication between processes, which can limit performance.

As we approach the exaflop scale and simulation data grows in size, the future of scientific visualization hinges not only on more powerful hardware, but also on efficient algorithms that can reduce the precision of visualization while maintaining the analytical proprieties of the data. To address this issue we implemented Qbeast, a distributed peer-to-peer system that will be described in Chapter 5. Qbeast builds the previously described D8tree (Chapter 3), thus allowing efficient sampling while building multidimensional indexes in real-time. Our aim is to give the user the possibility of choosing the right trade-off arbitrarily between level of precision and response time. Our system allows researchers to visualize simulations in real time and after the simulation has finished, it allows visualizations of complex queries as well. For example, what is the path followed by a specific particle, how do particles mix in a region, where do particles in this region come from, how many particles go across a given section, and all options that can be programmed into a DB query.

3.7 Real-time D8-tree index creation

We previously presented the D8-tree for static datasets[31]: it used a batch processing approach to build the index, which is efficient but it requires the simulation to have completed. To analyse a simulation at run time, we have to build the index at the same speed the data is generated. The D8-tree structure contains copies of the same item at multiple levels of the tree. For each level k, the whole domain is decomposed in 8^k partitions that we call cubes. Each cube has a maximum capacity of C. When a cube reaches C, it sorts its content by a random priority and then it discards the items with lower values. Therefore, a query that starts with the root leaf reads a uniform random sample of data. While descending the tree it can increase the number of samples. Maintaining updated such a structure on disk requires sorting all the elements in each cube, which has a prohibitive performance cost under

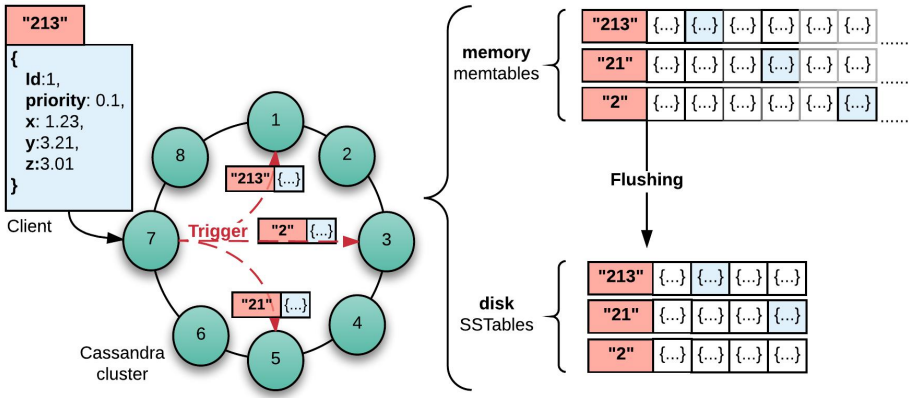


FIGURE 3.7: Dynamic D8-tree indexing: (left) A push query of a new data item triggers a replication to the higher cubes into in-memory MemTables, with records ordered by priority. When a Memtable reaches its threshold size it is flushed to disk, but only the first few elements are kept.

a continuous stream of insertions. Given an index of K levels, for each insertion, we should access K different cubes, and modify each cube index by adding the new element and possibly removing the overflowing ones.

Such a mutable structure has major drawbacks: it causes high cache pollution and it requires costly synchronisation in multi-core architectures. Furthermore, it performs badly with both rotational disks and flash drives. In rotation HDD, the time for moving the magnetic heads to the desired disk partition –known as seek time– is a significant part of the overall time, and thus it penalises accessing to non-consecutive disk sectors. At the same time, as several articles pointed out[5][60], the SSD devices degrade under a heavy load of random writes.

Indeed, NAND Flash Memories have unique erase-before-write and wear-out characteristics, so that a block, usually 64-128 pages, must be erased as a whole before writing a new page. Erasing is slow and it degrades the device, which is why the SSD controller performs out-of-places updates. The result is fragmentation, which under an update-heavy workload, requires writing several times the same page to disk as Chen et al.[24] measured, this fragmentation can lead up to 14 times slower performance.

LSM-tree indexes, such as the one used in Cassandra, have been pointed out as an alternative to the widely adopted B-tree as they do not suffer of write amplification[110]. LSM-tree designs exploit the hierarchy between the volatile and secondary memory, using the first to optimise accesses to the second. Elements are kept in memory until a threshold is reached and then the items are stored ("flushed") into a disk file in a single sequential write. A background process, called *compactions*, aggregates single files in larger ones.

For all the cited reasons, we decided to follow the LSM-tree approach to create our dynamic D8tree. We modified the database Apache Cassandra to implement

our index. We made two changes: we added a trigger and we altered the logic of the flushing algorithm. Figure 3.7 shows the overall architecture. On the right, we can see that the client sends the request to the designed node where the trigger replicates the query to the higher cubes. Cassandra stores each insertion into an in-memory structure called Memtable keeping the records ordered by their priority. When a Memtable reaches its threshold size, we modify the *flushing* algorithm to write only the first elements to disk. In particular, Figure 3.7 shows how the particle we inserted went in different positions of the three cubes so that in the higher one "2", it is discarded when flushing the content into an SSTable on disk. Meanwhile, the compaction process takes care to unite the small SSTables into larger ones, thus reducing the index size.

3.8 Architecture

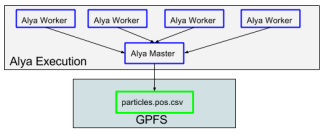
This section provides a brief summary of how we integrated Alya with Qbeast and it describes how we connected Qbeast with ParaView, allowing to query the system in real-time.

The original Alya architecture is shown in Figure 3.8a: a single master node manages the simulation output. The Alya master collects all particles from the workers at each time step, and then it appends the new results into a CSV file which is stored on a distributed file system. While the CSV format has inarguably advantages given its simplicity and human-readability, sending all results to one single worker is a major performance bottleneck. We tested two different integration prototypes: master-slave (Figure 3.8b) and peer-to-peer (Figure 3.8c).

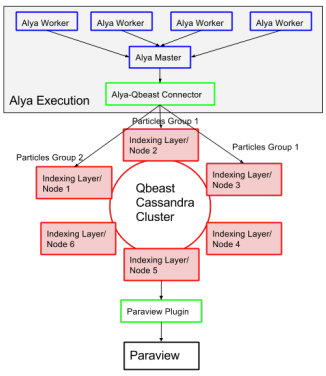
Writing a high-performance communication protocol can be challenging, thus we decided to connect Alya to Qbeast using the official Cassandra driver. We created the *Alya-Qbeast connector* as a C++ library wrapper for Alya, which is written in Fortran 90. The connector uses the C++ Datastax [23] driver to connect through an asynchronous protocol to Cassandra. The driver manages failover and reconnection in case of messages lost or node crashes, and it takes care of delivering the request to the correct node. Cassandra takes care of persisting, indexing, and efficiently writing to disk all the parallel requests sent by Alya. We used the same Alya-Qbeast connector for both implementations: in the first version, only the Alya master invokes the connector while in the second one each slave uses the connector independently.

As Cassandra takes care of reordering the results, we can avoid the gathering operation in the Alya Master. Figure 3.8c(c) shows the peer-to-peer architecture. In the master-slave design the master dictates the output time. However, in the peer-to-peer version, the slowest node limits the total output time. Each Alya worker simulates a sub-domain of the whole simulation volume, and therefore it is common that in some phases of the simulation there are workers with a higher number of particles, which consequently need more time to complete the simulation step.

(A) The original architecture



(B) master-slave version



(C) peer-to-peer version

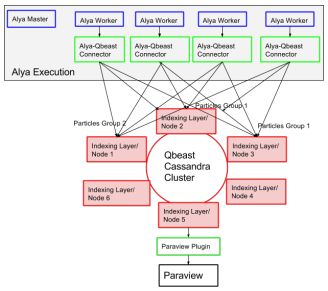


FIGURE 3.8: The three different architectures we tried in this article: (a) The original set up where an Alya master node receives and writes all the information, (b) The Alya master node is connected to QBeast nodes, and (c) all Alya workers push information to QBeast nodes independently.

In the experiments section, we present a detailed comparison between these two Connector implementations.

The C++ Cassandra driver uses an asynchronous protocol thus allowing to efficiently send several requests to Cassandra in parallel. However, as issuing too many requests leads to system instability and performance degradation we added the parameter **parallelism-level** to limit the number of operations that are in execution at the same time. Another option is to reduce the number of queries by grouping them in larger ones, called batch statements. The trade-off is between network bandwidth and latency. As this aspect influences performance, we defined the parameter **batch-size** that indicates how many requests group into a batch statement.

The overall system is composed of: 1. ParaView 2. the ParaView-Qbeast plugin 3. the Qbeast Query Engine: a distributed middle layer that queries the D8-Tree index exploiting the Cassandra data locality. 4. the Cassandra nodes: store and serve requests on the D8-tree index.

Usually, ParaView runs on personal computers and renders the data locally, so that if too many particles are loaded together the system collapses. While it is possible to distribute the rendering on multiple machines, it is costly both in terms of time and resources. However, in many cases a random sample of the results is enough for an interactive analysis of the simulation: we allow the user to choose the right trade-off between the level of detail and system responsiveness. By tuning the *precision* and *max-results* parameters, the user defines the percentage of data to visualize or sets an upper bound to the number of elements to fetch. Thanks to the D8-tree [31], this can be efficiently implemented with Cassandra.

The process of sampling a dataset, also called data-thinning, would normally require the following steps: 1. Read the whole data set 2. Randomly discard in memory 99 elements every 100 3. return the 10000 filtered element. In such a way, reading the whole dataset or just a sample requires almost the same time.

In contrast, the D8-Tree has a response time proportional to the number of elements returned: getting one random part out of ten requires one tenth of the time to read the whole data. The D8-tree uses a random priority to organize the data into cubes, so that in the higher cube we find the elements with higher priority. Therefore, if we want to get the k% of a dataset we just have to: 1. read the root cube 2. select all elements with $priority \leq k$ 3. if point 2 returns new items, we descend into the 8 cube children and for each of them we go back to point 2. 4. if all queries have completed, we return the results to the client.

In Figure 3.9 we show some screen-shots taken during the simulation of particles flowing through a human nose. With this visualizations we were able to obtain insights about how the simulation was proceeding before it completed.

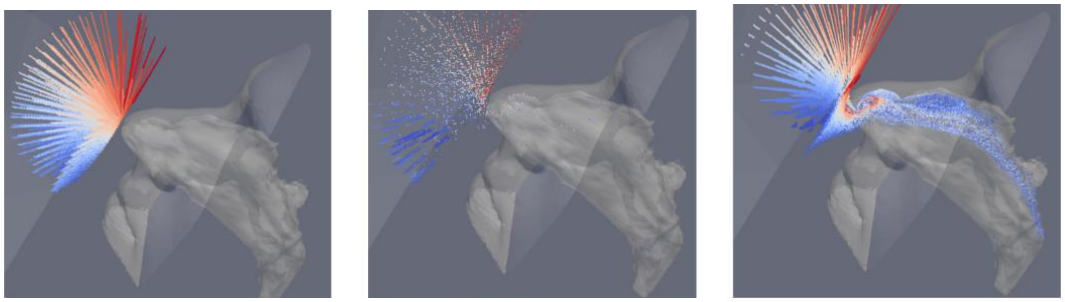


FIGURE 3.9: Screen shots of real-time visualization of particles flowing into the respiratory system in a rapid air intake simulation.

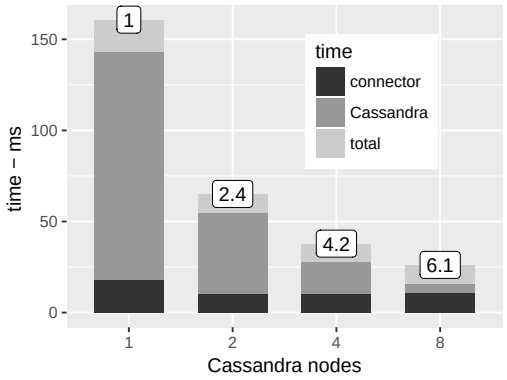
3.9 Experiments

In this section we present the tests we carried out on our Alya-Qbeast integration. We analyze its performance compared with the original Alya implementation, and we discuss the parameters that mostly influence performance. All tests simulated the same particle respiratory system and ran on the BSC-CNS Marenostrum 3 Supercomputer. Each node in Marenostrum 3 is equipped with two Intel SandyBridge-EP E5-2670 20M 8-core for a total of sixteen cores at 2.6 GHz processor base frequency and 32 Gb of DDR3-1600 DIMMS ram. Each node uses the IBM GPFS file system [62] running on the Infiniband FDR10. Both Alya and Cassandra started at the same time in the supercomputer: Cassandra employs an internal BSC's library module, that allows creating a Cassandra cluster using a queue job system.

Horizontal system scalability The first experiment tests the scalability of Cassandra: we increased the number of Cassandra nodes to measure the relative performance improvement. We simulated 6.75 million particles moving during 10 time steps using 256 Alya MPI workers. We disabled the Qbeast indexing, and we used the peer-to-peer version.

For each Cassandra cluster configuration, we had to properly configure the Connector. Indeed, to fully take advantage of the additional nodes, we need to increase

(A) Cassandra scalability details



(B) GPFS VS Cassandra output time

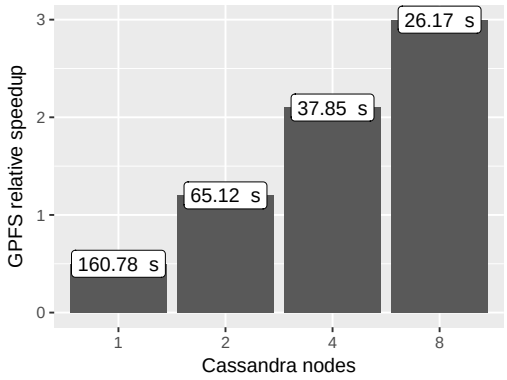


FIGURE 3.10: System scalability

the parallel insertions. For example, with 4 Cassandra nodes, the best performance is with 200 concurrent requests.

Figure 3.10a shows that from 1 to 4 nodes, Alya scaled perfectly, in fact the response time halved when the Cassandra nodes doubled. However, with 8 Cassandra nodes, the relative speed-up resulted in only 6 times, instead of the ideal 8. To understand what limits performance, we broke down the output time into Connector and Cassandra. The Connector only includes the time to transform a particle from the Alya format into the Cassandra format. The latter considers the database processing time. Figure 3.10a shows the different time components of an insertion and how they change when increasing the number of Cassandra nodes. We can see the time taken by Cassandra decreases while the time required by the Connector remains stable: with 8 nodes the first reduces of 9.2 while the second improves of only 1.6. With more than 4 nodes the Connector is the major performance bottleneck and it does not allow the system to scale. Future work will improve the connector implementation and help reduce this fixed performance cost.

GPFS versus Alya-Qbeast This test compares the performance of the original Alya with the peer-to-peer Alya-Qbeast version. The original Alya design has a master process that collects the results from all workers, and then it writes the results into a file on GPFS.

The test simulated 6.75 million particles, 256 Alya MPI, 10 time steps. The original Alya took 80.6 seconds to complete the output step of one iteration. By enabling the Connector and using only one Cassandra node, the output step took 160.78 seconds, while increasing the number of Cassandra nodes reduced the response time to 65 seconds with 2 nodes and to just 26 seconds with 8 nodes. In Fig. 3.10b we show how Alya runs 3 times faster thanks to Cassandra's linear scalability.

Alya master-slave versus peer-to-peer connector This experiment compares the master-slave and the peer-to-peer Alya connector versions: both write data to Cassandra. The aim is to understand how much the original Alya architecture is penalised by the master-slave approach. For this experiment we considered about 300 thousand particles, simulating their flow during 10 time steps. Alya used 64 MPI

workers and insertions were handled by a single Cassandra node with the Qbeast trigger enabled. The D8tree maximum depth was 5.

To fairly compare the two implementations, we had to find the optimal Connector configuration for the parameters *batch-size* (optimal value 5) and *parallelism-level* (optimal value 10). With the optimal settings, the Alya-Qbeast peer-to-peer version doubles the performance of the master-slave version. Even though we did not perform extensive tests of all possible settings, we had the possibility to estimate that a master-slave approach requires about 84% more than the peer-to-peer one.

Dynamic indexing overhead In order to index simulation data in real time, the Qbeast trigger needs to duplicate each insertion multiple times so that it propagates up to the higher tree levels. To measure its overhead, we ran a new test with the same condition of the previous scalability test but enabling the indexing. With the D8tree maximum depth set to 5, the execution took 105.7 seconds. Compared to 37.85 seconds of the non-indexed version, it proved to be 1.8 times slower. As expected, indexing on real-time added an overhead: as maximum depth equalled 5, Qbeast replicated each insertion 5 times. However, we experienced a smaller performance detriment, as long as the Qbeast periodically filters in memory the inserts that would not fit into the index, thus dramatically reducing the amount of data written to disk and decreasing the algorithm overhead from 5 to 1.8.

Query performance experiment This last test compares the query capability when using the D8-tree or when storing data into CSV files. For our experiment, we chose an important query in our visualisation system: A small sampling over a large simulation area. We decided to perform a query that returns 1.5% of the particle present in the whole space domain, over a dataset of about ten million particles. The simulation considered 54000 particles during 200 time steps. We used one Cassandra node and 12 Alya nodes connected through MPI. We ran the particle simulation with the trigger enabled to generate the D8-tree index, and we used our ParaView Plug-in to query and visualise the random sample of particles. We obtained the response in 4.19 seconds.

ParaView does not allow this kind of query so we used Apache Spark to read and filter on memory the results from the Cassandra database to compare. Apache Spark required about 210 seconds to load and filter the sample in memory, while with our system we were able to retrieve and visualise the results in only 4.19 seconds. This massive speed-up –52 times faster– enables the user to analyse interactively a simulation.

3.10 Summary

In this Chapter, we presented our second contribution, the D8tree, and we presented both its first static-data implementation and our prototype architecture that integrates Alya, Qbeast, and ParaView allowing real-time interactive explorative analysis and visualization of large simulation data. On static data, our tests showed the

D8Tree can be up to 47 times faster than PostGIS serving data-thinning queries, and it has also proved to be substantially more rapid for the other kinds of requests examined. Using Qbeast for real-time indexing, our work improved the storage of simulations, boosting performance and analytical capabilities. Our tests demonstrated that key-value databases are a viable alternative to plain file storage for simulation persistence, as they allowed to improve the write performance (in one case by up to 65.7%) by simply adding more database nodes. Also, we showed that it is possible to maintain at real-time an index over the simulation results so that it is feasible to visualize the early results of a running simulation. Indeed, while in our prototype the indexing slows down the execution by 31%, it enables extremely fast arbitrary-approximated query on the simulation. Compared with Apache Spark, we achieved a 52-factor speedup.

The results of this prototype were satisfactory, yet we saw that the system had several areas of improvement. First of all, each operation needs to be broadcasted to all levels of the tree, which adds a considerable overhead both in terms of the number of transactions that each node has to proceed, as well as regarding disk space. In Chapter 4, we will study how it is possible to improve the architecture, reducing the number of copies that the index is producing. To do so, we will use the analytical model we previously introduced in Section 2.9 to model the performance of a key-value database thus helping us understanding which are the limiting factors of our initial architecture.

3.11 List of publications

Artigues, A., Cugnasco, C., Becerra, Y., Cucchietti, F., Houzeaux, G., Vazquez, M., ... & Labarta, J. (2017). ParaView+ Alya+ D8tree: Integrating High Performance Computing and High Performance Data Analytics. Procedia Computer Science, 108, 465-474.

Cugnasco, C., Becerra, Y., Torres, J., & Ayguadé, E. (2016, January). D8-tree: A de-normalized approach for multidimensional data analysis on key-value databases. In Proceedings of the 17th International Conference on Distributed Computing and Networking (p. 18). ACM.

Chapter 4

The AOTree: a write, and eventually read, optimized MIS

This Chapter contains our third contribution, the AOTtree. At first, we will address some of the shortcomings of the D8tree, in particular, the need to replicate several times the elements present in less-populated areas that causes a high overhead in terms of transactions and disk space. To this end, we will first describe the D8tree's limitations, and then we will use the analytical model to study what can be changed in the indexing algorithm without compromising query performances. Based on this analysis, we will propose a new theoretical indexing algorithm; the **Outlook-Tree** that reduces the disk space requirements without compromising the query time. Then, as the OutlookTree does not work well under heavy write workload, we will propose the Asymptotic Outlook Tree, in short AOTree, our novel structure that overcomes the shortcomings of a distributed tree with an innovative lazy optimization approach. The description will focus on the key aspects required to ensure consistency of the overall system during concurrent reads, writes, and optimizations. Finally, we will introduce the results of comparative experiments of the AOTree versus GPFS files and PostgreSQL.

4.1 Indexing algorithms

While there is a wide variety of secondary indexing algorithms, they all tend to organize data in hierarchical structures such as trees and this is an issue if we want the index to be distributed in multiple machines. In such cases, expensive operations as distributed transactions and locks are required to preserve the consistency of the data across different machines. Let's take as an example the Quad-tree, a 2-dimensional space-partitioning indexing algorithm that divides the domain into partitions of fixed sizes. The algorithm principle is simple: first, create a partition - a square -, and store data inside. Then, when the number of elements stored reaches a threshold, split the partition into smaller equally-sized parts. Finally, redistribute the items into the smaller squares. During the splitting phase, we have to create a lock on the first partition, create the smaller squares in remote nodes, move all the data into them and finally release the locks. While the cost of such operations in a single

machine can be neglectable, they are prohibitive when storing the index on multiple servers. Distributed locks are not only an obstacle for the system availability; they also increase response latency and diminish throughput.

Aside from updating the structure, there is also the problem of querying it. A hierarchy means that all interrogations must access to the same root elements: all queries in a database system must pass through the same single server, sharply constraining the overall cluster performance.

In Chapter 3, we presented our work of integrating Alya and D8tree [31], proving that it is possible to sustain the writing throughput of a simulation while indexing the data in a pure peer-to-peer fashion. The D8tree uses de-normalization to relieve the need of distributed transactions and enable a uniform workload distribution between the cluster nodes. The main idea is to build the index on a perfect 8-ary tree[1] with a configurable maximum height. Once they reach their maximum capacity, the nodes only keep a sample of high priority data in the node domain.

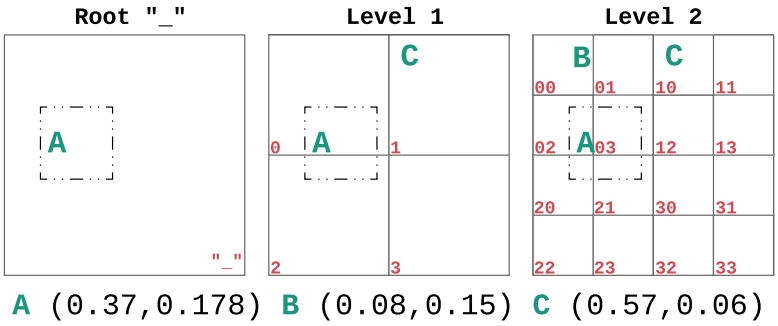


FIGURE 4.1: A 3-levels D8tree with partition max size = 1.

Figure 4.1 shows an example of a 3-levels D8tree where partitions can contain at most one element, and we are storing three 2-dimensional data points - A,B,C - of decreasing random priority. When inserting a new item, we store it into the smallest area where it fits (e.g, "02" for A), then we try to insert the element in the "father", which name is the cube prefix ("0" for A). If the father has reached its limit, we select the elements with the higher priority ($A > B > C$).

If we use a key-value data store, a convenient way to store the partitions is to use the z-order [115] identifier as key, and the list of the elements that are stored in the partition as value. Using this approach, the Listing 4.1 shows the entries we would store in the database to implement the example in Figure 4.1.

LISTING 4.1: Figure 4.1 implementation on a key-value data store

```
kv['00'] = [B]
kv['02'] = [A]
kv['10'] = [C]
kv['0'] = [A]
```

[1] A k-ary tree with all leaf nodes at the same depth. All internal nodes have degree k. [85]

```
kv['1'] = [C]
kv['_'] = [A]
```

In this simple example, we can notice that the number of times an element is stored depends both on its random priority and the number of items - density- present in its area. For instance, C has a lower priority than B, but it gets replicated one time more, as the domain of the partition "0" contains two elements while "1" contains just one. This shows how the elements stored in less populated areas are more likely to be replicated.

The fixed structure of the D8tree allows choosing different paths to complete a query. Let's suppose we are interested in all the data in the range $0.3 < x < 0.6$ and $0.15 < y < 0.40$ (the dashed blue rectangle in the Figure 4.1). We can start from the root "_" and then decide to proceed further down after analyzing what we found. Or we can go directly to level 1, reading "0" and "2", or to level 2 by issuing 4 requests for "02","03","20","21".

Another benefit of this approach is that we can create the index once and for all and then leave the Query Engine to optimize the execution at run-time.

Executing efficiently approximate queries is a key feature of the D8tree. The index structure creates pre-computed samples enabling huge speedups for approximate queries. Indeed, storing random samples of each space partition is a fundamental piece of the D8tree, as it allows an interactive exploration of a vast dataset and it efficiently distributes the data across nodes. To insert a new item, we use a hash of its identifier as a random priority. The benefit of this approach is that when we know the maximum priority stored in a cube, we can estimate how many items are present in all descendant cubes; in its space domain. For instance, if a partition contains one thousand elements and the maximum items' priority we find is smaller than 1% of all the possible random values, we can roughly estimate the population of this area. Indeed, if 1 is to 100, as 1000 is to the number of elements in the domain, we can estimate $\approx 100,000$ elements. More precisely,

$$1000 : number_{elements} = 1 : 1000$$
$$number_{elements} = \frac{1000 * 1000}{1} = 100000$$

Knowing the index fan-out, the Query Engine can use this estimation to decide whether to "jump" *short*, *long* or *regular*.

Figure 4.2 shows the three kinds of jump that can occur on a simplified one-dimensional tree generated using a gamma distribution. The percentage in the tree on the left indicates how much data the node contains over the whole node domain. Let us assume that the Query Engine has just retrieved an index's node containing 38% of the required data in a specific area and now it has to decide how to proceed the navigation. Since the cube contains 38% of the data, we can estimate that we

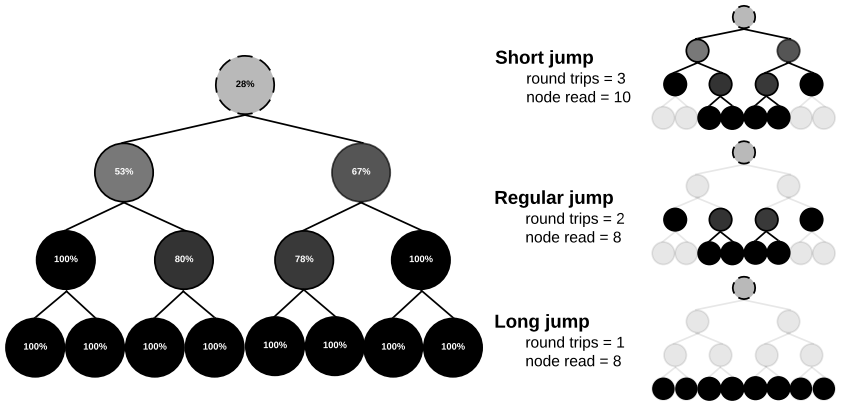


FIGURE 4.2: The picture shows an example of short, regular and long jumps.

would need to access at least $\frac{1}{38\%} = 2.63$ full cubes if the data distributes uniformly. In this optimistic hypothesis, we can find all data $\lceil \log_2 2.36 \rceil = 2$ levels below.

We refer to a *"regular jump"*, if the algorithm accesses to the cubes two levels below, as the estimation predicts. If less, it is a *"short jump"* and a *"long jump"* if more. After a jump we land on some cubes; if they do not contain all the data that is needed, the Query Engine iterates the jumping process. Figure 4.2 shows how the three policies differ regarding the number of iterations - round trips - and access patterns. In Figure 4.2, all data that falls on the left of the root belongs to the domain $[0, 0.5)$, but we can retrieve all elements in different ways. We can read everything at once from one single cube - *"short jump"*-, read two partitions with a *"regular jump"*, or we can visit 4 partitions with the *"long jump"*. Issuing more requests to read the same data adds an overhead. On the other hand, they can run in parallel on multi-cores and servers with possible performance speedup.

Therefore, depending on the hardware architecture, the Query Engine might opt for a type of jump or another. Which strategy is favorable depends on several aspects, as the percentage of data we need, how the data distributes, how we consume the data (all together, in batches, with incremental precision etc..) and last but not least, it depends on the characteristics of the system, such as the number of nodes or the type of hardware used.

4.1.1 D8tree drawbacks

We found out that, although the outstanding query performance of the D8tree, the overhead that it introduces regarding the number of transactions and I/O requests for each insertion limits the overall system performance in write-intensive scenarios.

Figure 4.3 shows the implementation of the D8tree on Cassandra [13]. A D8tree indexing a 3D space creates a perfect 8-ary ($2^{dimensions}$) tree, where each level L has 8^L (only 4^L in the figure for space constraints) partitions of the space domain. The

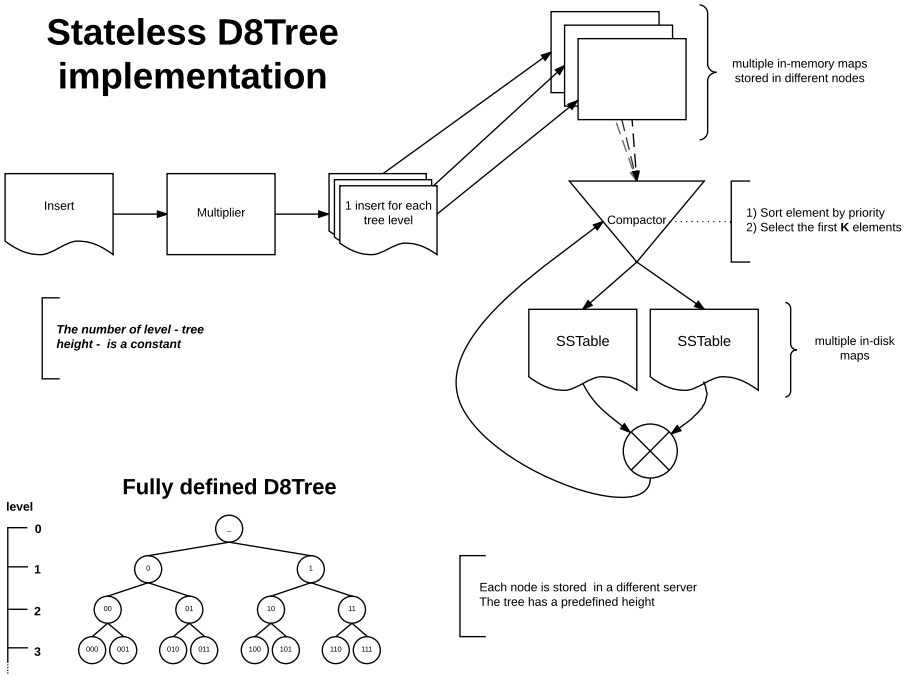


FIGURE 4.3: The original architecture for runtime D8tree indexing.

algorithm stores every insertion in the corresponding partitions in each level of the tree. When one node of the cluster receives a client request, a component called Multiplier takes care of redistributing the data. First, it calculates in which partition (cube) of the lowest level of the index the data should fall in. Then, it sends the update to the cube and all its ancestors, forwarding them to the corresponding server. The drawback of this approach is that we have to issue a new request for each level of the tree, with a considerable overhead regarding network and CPU usage. However, not all the requests end up in the disk. Apache Cassandra, similarly to other NoSQL databases, delays writes to disk to improve the disk throughput. Cassandra stores the incoming requests into an in-memory mutable structure called *Memtable*. When the *Memtable* exceeds a configurable threshold, the requests go into an immutable on-disk structure, called *SSTable*. A background process called *Compactor* takes care of merging the various *SSTables* into larger ones to reduce the number of I/O requests during reads. In Chapter 3, we modified the *Compaction* process so that if there are more elements per cube than allowed, it sorts each entry by their random priority and then discards the ones with a lower priority. The same process takes place when a *Memtable* is dumped into a *SStable*. The advantage of this approach is that it is possible to build a D8tree without having a shared state between nodes or threads. Indeed, since the compaction works as an idempotent monoid, there is no constraint in the order we process the single data parts as it does not change the final results. However, the order the compaction tasks place influences the performance.

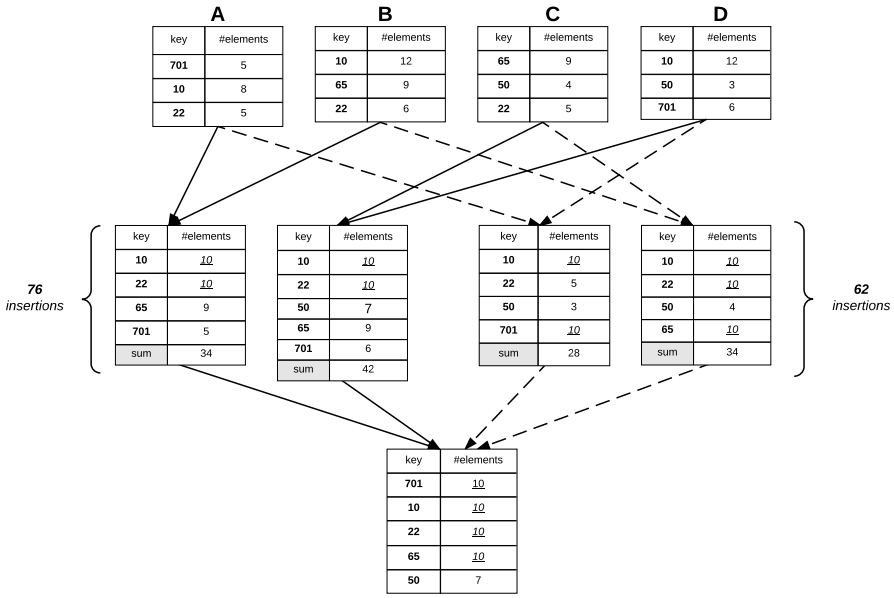


FIGURE 4.4: How the order influences compaction performance

Figure 4.4 shows a small numerical example: on the left, performing

$$compact(compact(A, B), compact(C, D))$$

requires 76 insertions, while on the right

$$compact(compact(A, D), compact(C, B))$$

only requires 62. Merging all four *Memtables* at onces would require storing only 47 elements. The order in which the *SSTables* and *Memtables* are merged depends on the time the data is received. If we send many elements that go in the same partition in a small time interval, it is likely that the *Compaction* will be able to drop a larger percentage of the data compared to the case the same elements are received in a large time span. Indeed, given its stateless design, the *Compactor* knows only about the small subset of the data residing in memory, resulting in a conservative evaluation of which data it has to filter; an overestimation that causes flushing to disk unnecessary data. Even though the background compaction eventually discards the redundant pieces, it can result in a considerable I/O overhead.

The second problem concerning the disk space usage is a direct consequence of the original D8tree design: generating a perfect 8-ary tree. It means that a 3D index with 10 levels might have up to $\sum_{n=0}^{n=10} 8^n = \frac{8^{10+1}-1}{8-1} \approx 1.2 * 10^9$ nodes and that an element can be replicated up to 10 times. While the number of cubes is large, it can be easily managed with a distributed key-value data store, especially if we encode the cubes containing no data as misses.

In Chapter 3, we showed that when the data is uniformly distributed in the domain, the space overhead caused by replicating the element at different levels for a 3D index is about

$$\lim_{Z\to\infty} \frac{\sum_{i=1}^{Z-1} K * 8^i}{\sum_{j=1}^{Z} K * 8^j} = \lim_{Z\to\infty} \frac{8^Z - 8}{8 * (8^Z - 1)} = \frac{1}{8} = 12.5\% \tag{4.1}$$

However, the cost can be much higher in real applications, as it strongly depends on the data distribution. For instance, in different use cases, we observed replication overhead ranging from 60% to one order of magnitude higher.

4.2 D8tree performance analysis

Since the goal of this work is to reduce the storage, network and CPU overhead of the D8tree, we analyzed which parts of the index could be modified without losing performance in read operations. The D8tree builds an index without any assumption about the performance characteristics of the underlying system: it does not take into account the overhead of accessing the data location or sending a remote request. As a result, we could find an optimal query plan with the same index structure even on two clusters with completely different technologies and performance. For instance, a system where all data fits in the primary memory might benefit from using smaller cubes rather than a cluster using rotational disks. Similarly, different network technologies have different costs to send and handle remote requests.

In the D8tree, the Query Engine is free to decide whether to perform a *"long jump"* up to the maximum height of the tree. However, this level of freedom comes to the cost that the index structure must provide a large combination of querying paths, and that causes a high usage of resources.

In particular, it is supporting the *jump long* that causes the higher overhead in the index, as it requires to copy elements in a high-number of smaller cubes. Therefore, to reduce the index size, we decided to analyze which could be a reasonable upper bound for the *jump long* so that we can reduce the index size with minor performance degradation. To this end, we used the analytical model presented in Section 2.9 that estimates the performance of a distributed key-value database when varying the number of servers.

Figure 4.5 shows the optimal size of the partition we should use to retrieve different amounts of items from a cluster with increasing number of servers. Obviously, the smaller the partitions are, the more requests we have to issue.

For instance, the smallest query **10K**, the one that returns ten thousand items, should be divided into $\lceil \frac{10000}{160} \rceil = 63$ partitions when the database runs on a single node, while it should have $\lceil \frac{10000}{75} \rceil = 134$ partitions on a database of 10 nodes.
Also, the model shows us that the optimal row size is 160 elements if all data resides in a single server, independently from the query dimension. Therefore, we

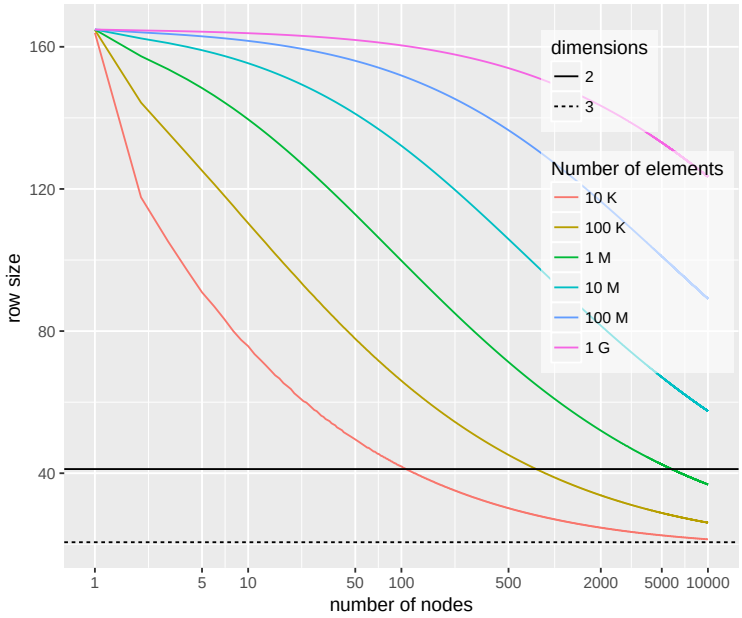


FIGURE 4.5: How the optimal row size changes for different queries and cluster sizes.

should use this value as the default cube size when building a D8tree as it is the row granularity at which the database works best.

When we do a *regular jump*, we try to guess at which level of the index we can find all the data. Ideally, at this level, the partitions have approximately the optimal row size, 160 in our example. On the other hand, the *long jump* would go one, or more, levels downwards where the cubes contain $\approx \frac{1}{2^{dim}}$ of the father's data as they split along all the dimensions. Therefore, in a 2-dimensional scenario, a cube is approximately one-quarter of their father size, in a 3D case one-eighth, in 4D $\frac{1}{16}$ and so on. At the same time, we will issue 4, 8 and 16 times more requests respectively, thus achieving better workload distribution. We can use our model to find out when the *long jump* is beneficial. In Figure 4.5, the horizontal solid line marks at what point a *long jump* in useful for the 2-dimensional case. Similarly, the dotted line indicates that point for the 3D case. In the first case, we would have a performance improvement running a few queries, but only in clusters of 100 nodes and more. On the side, with 3 dimensions there is no benefit until we have clusters of 10 thousand nodes or more. In none of the cases, we observed a benefit for *long jumps* of two levels or more.

The result of such analysis is that we can have a better understanding of what makes sense to replicate, and what does not, in a realistic database cluster scenario. For instance, Figure 4.5 shows that we have no improvement replicating more than the *regular jump* for 3 and more dimensions with the tested hardware and software. In the case of 2D, we might have a benefit replicating one, but only one.

Even though these results are based on a particular hardware/software configuration, we can reasonably assume that future clusters will still have a limited -

countable - benefit from a *long jump*, thus we can accommodate future architectures adding a configuration parameter, which we call *technological boost*.

4.3 The OutlookTree

Following the previous assumptions, we improved the original D8tree design by replacing the idea of a global "index max-height" with the idea that every single cube has its own "max-height". Therefore, from a cube perspective, we will only replicate the data up to the offspring nodes necessary to perform a *regular jump*. We named this new kind of tree, the *outlookTree*, as we duplicate the data only up to the *"reasonable outlook"* of each cube, with the assumption that we will do at maximum *regular jumps*.

Definition 4.3.1 *The **outlookTree** is a **K**-ary unbalanced rooted tree **T(D,S,B)** where **D** is the number of dimensions and where the nodes have either zero or $\boldsymbol{K} = 2^D$ children. Each node is a K^Lth disjoint partition of the D-dimensional space –node's domain– where **L** is the **depth**, meaning the distance between the node and the root. Each node contains up to **S** of the elements that fall into its domain. If a node reaches the threshold **S**, it maintains a random sample of the whole data, with **P** representing the ratio between the data contained and the data that falls into its domain.*

*The outlookTree guarantees that any node is the root of a perfect **K**-ary tree with height equals to the node's outlook which is defined:*

$$\boldsymbol{O} = \lceil \log_K \frac{1}{P} + B \rceil = B + \lceil -\log_K P \rceil \tag{4.2}$$

*The positive constant **B** represents a configurable technology boost.*

Figure 4.6 shows a graphical comparison of three kinds of indexes produced over a dataset composed of ten thousand random elements with a mixed uniform and Zipf distribution. The maximum cube size is one hundred elements. The Quad-tree is displayed on top: the white nodes contain data while the colored ones are references to other nodes. A common way to implement Quad-trees on top of key-value databases is to use a unique ID for each space partition. Each cube uses the father's ID as the prefix, as it represents a disjoint fraction of the father's domain.

Therefore, queries start from the root node "_" and proceed downward visiting all partitions interested by the query. Let us suppose we are interested in a random sample containing 20% of the elements of a particular area. As the QuadTree splits the space into regular partitions, we can use the query predicate to calculate the smallest cube that might contain the required information. Supposing that this cube is "03". Since we do not know the state of the index, node "03" could contain data, be a reference to other nodes, or not exist at all. Thus, all queries must start from the root node. In our example, we would access in sequence the reference nodes "_", "0", "03", and read the data stored in "030","031","032","033". Finally, we would filter in memory the result by randomly selecting one element every twenty.

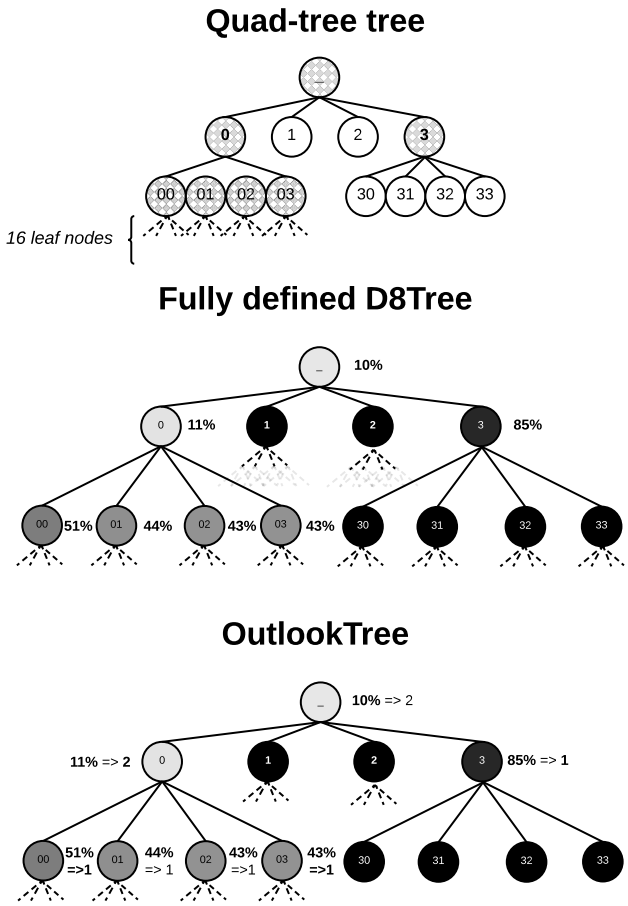


FIGURE 4.6: Comparison between the Quad-tree, the D8tree and the outlook-tree.

The mid section of Figure 4.6 shows a D8tree built on the same data set. The node color represents the percentage of data present in the cube: black means the node contains all the elements that fall into its domain, while the lighter color indicates that the node has only a fraction of them. Serving the previous query with the D8tree is faster, as we can directly go to node "03" since we know that it exists and that it contains at least a fraction of the data we need. In this case, it comprises the sample we need, thus no further steps are required. As drawback the total number of nodes is much higher. In Figure 4.6 , the D8tree has $4^4 - 1 = 255$ data nodes, while the QuadTree has $16 + 4 + 2 = 22$ data nodes and $4 + 2 + 1 = 7$ metadata ones.

The bottom of Figure 4.6 shows the OutlookTree, configured with B equal to 0. The value after the arrow $(=>)$ is O, the outlook. If a node has outlook O, all the nodes at O levels of distance are defined. Consequently, the previous query would start from node "_", then go directly to node "03", as the outlook=2 guarantees that it exists. The advantage is that only some parts of the tree are a perfect 2-ary tree; there are only 37 data nodes, 85% less compared to the D8tree.

Figure 4.7 shows an example of how three the indexes can be implemented on

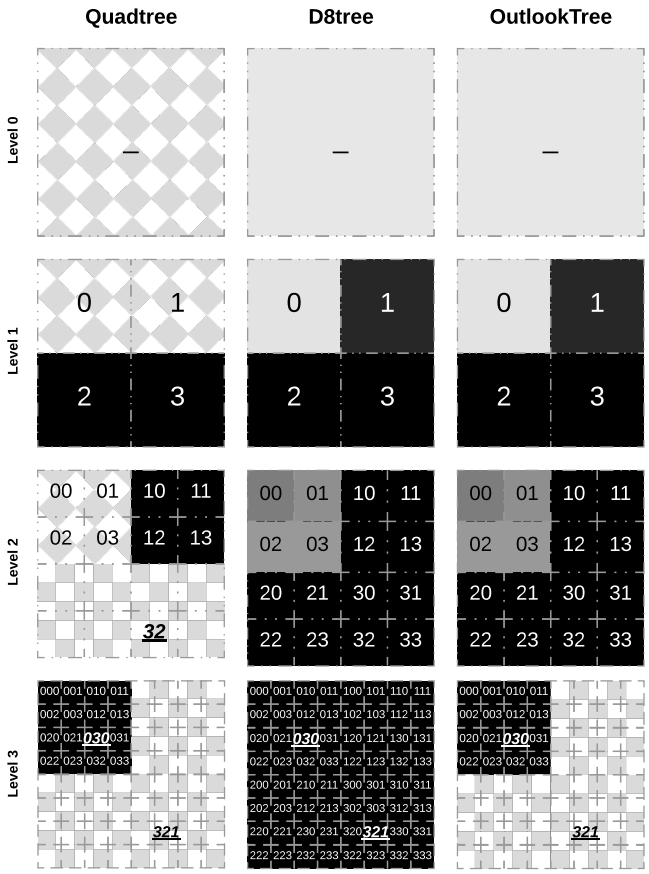


FIGURE 4.7: A graphical representation of how the data is organized in three MI algorithms.

a key-value database, using as the key a space-filling curve - the z-order. The area with a square pattern represents non-defined nodes– a miss–, while in the QuadTree the rhombus pattern illustrates the metadata nodes.

Let us imagine we have two queries about two different areas and that the smaller node that contains the queries' domain are nodes "321" and "030" respectively. Both queries are easily implemented with the D8tree as it guarantees that each node up to level 3 is defined and that we can directly access the two nodes. On the other hand, we cannot use the same approach with the QuadTree and the OutookTree. Indeed, a Top-Down navigation of the QuadTree requires for query "321" to go through nodes "_" and "3" and filter in memory around one-quarter of the data. Instead, query "030" goes down the whole tree, moving through nodes "_","0","030".

The same approach with the OutlookTree is slightly more efficient, as once we access to the root node we get its outlook - 2- and thus we can jump directly to level 2, saving a roundtrip in both queries.

One might argue that while we reduced the number of nodes, we reintroduced the performance bottleneck caused by forcing all queries to pass through the root node. However, in both the QuadTree and the OutlookTree, it is possible to alleviate the stress on the higher nodes by caching some information. Furthermore, as we will describe in details later, the OutlookTree allows using a compacted and loosely

updated in-memory data structure that enables the use of approximated data structures. Indeed, while in the QuadTree all nodes must have a consistent view of the cached information to avoid losing data, our architecture works correctly also in case of outdated caches or missed updates.

4.4 The AOTree: eventually building the OutlookTree

The previous section described the benefits of the *OutlookTree* design over the *D8tree*, stressing out how it was possible to reduce the disk space and the number of transactions by "cutting" part of the tree. On the other hand, the fact that the outlook of each cube might increase during the index lifetime makes the *OutlookTree* structure dynamic, thus hard to keep both consistent and fast in a distributed environment.

The "theoretical" outlook of a cube is calculated using Formula 4.2; it has a monotonic value that increases as soon as new data goes into the index. However, strictly following the *OutlookTree* definition, every time the outlook of a cube increases, we should forward all its data to its descending nodes. A straightforward implementation would require distributed locks and transactions, similar to the ones needed for the QuadTree, which would cause the same limits regarding system scalability and availability. For this reason, we designed an architecture that builds a sub-optimal version of the index that eventually converges to a full *OutlookTree*. In this way, we are able to implement the index without locks while optimizing the index while serving queries, in a process that we call *ReadOptimization* (RO). Once an RO completes forwarding the data of a cube to its descendants, we can increase the node's "*committed outlook*", which is the one we can use during reads. Alternatively, we use the term "*committed level*" to indicate up to which level a particular partition of the tree is optimized.

The first difference we can notice, comparing the original D8tree architecture in Figure 4.3 and the architecture we implemented for the *OutlookTree* in Figure 4.10, is that we introduced two new components; the *RangeEstimator* and the aforementioned *ReadOptimizer*.

The *RangeEstiamator* (RE) has three main duties. Firstly, it reduces the number of transactions by sending a copy of the data only to cubes where it might fit. Secondly, it ensures that the cubes' committed outlook is guaranteed. Lastly, it ensures no data is lost during a *ReadOptimization*. To achieve its goals, the RE uses an in-memory data structure to estimate in which nodes to insert the new items.

As shown in Listing 4.2, at a high level the *RangeEstimator* is a function that returns two values, r_{from} and r_{to} which are used to calculate where to send the data.

LISTING 4.2: A psuedo-code of RangeEstimator application

```
cube = '023221230010931...'
r_from, r_to = range_estimator(cube)
insert_into_cubes = []
```

```
for i in range(r_from, r_to):
    insert_into_cubes.append(cube[0:i])
```

The *RangeEstimator* generates a random priority for each element using a hash of the item's unique identifier. Then, it uses the priority to calculate from which level onwards the element should be inserted. It starts from the root: it compares the node's maximum random priority, and if the element's priority is higher, it iterates the process only to the child that could contain the elements. The process continues until it finds a node with a smaller max-priority, thus defining r_{from}.

In the meantime, the *RangeEstimator* calculates up to which level to propagate the insertion; the value r_{to}.

In a first implementation, we simply used Formula 4.2 to calculate the outlook, updating the write_rand only once we had redistributed the data. However, the system incurred in a deadlock under heavy updates. Indeed, we had the undesired situation where we had to redistribute the data before updating the write_rand, thus without reducing the amount of elements a cube can contain. This resulted in the cubes growing faster than the system was able to replicate them.

For this reason, we now distinguish between the *"transient"* write_rand, which can be continuously updated as it influences only the r_{from}; and the committed outlook that influences the r_{to}.

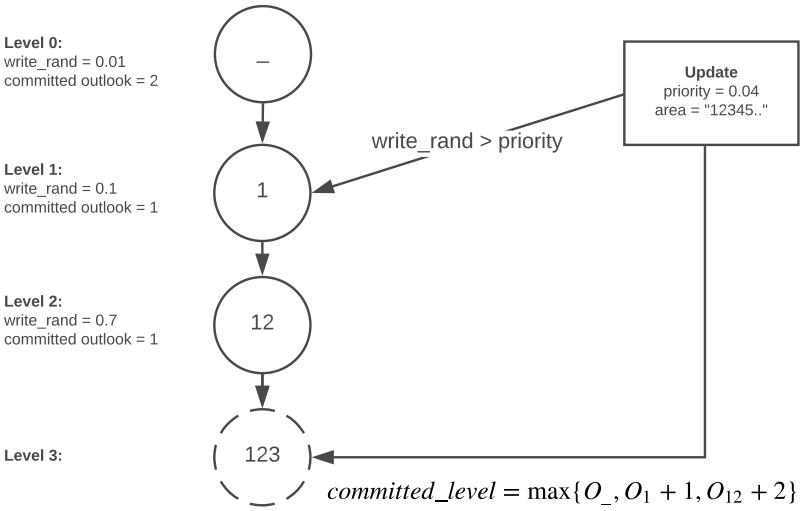


FIGURE 4.8: An example of insertion range estimation.

Figure 4.8 shows a simplified versions of the *RangeEstimator* that uses the write_rand and committed outlook. We can see how by inserting an element with priority equal to 0.04, the algorithm goes down the levels of the index to find the first cube that has a higher priority - cube "1" in this case. At the same time, the algorithm calculates the first not-overflown node - the dashed "123" - and the maximum of the *committed outlook* of all the visited nodes. Therefore, in this case, the insertion would be propagated to the nodes "1","12" and "123".

Distinguishing between the transient and committed status ensures consistency between concurrent read and write operations, but it does not guarantee that the *ReadOperation* does not lose data. Figure 4.9 shows an example of a Lost Update (P4)2.2.12, described in Section 2.2.3, that can occur in any tree-based indexing algorithm when we have a node that has reached its maximum size, and we have to break it into new sub-partitions. The problem is that without a lock, we cannot read and then update the index without losing data. In the image, the data from cube "..212" has to be propagated into its children. Therefore the "splitter" process reads all the items from the first cube and divides them between the ones that go into "..2121" or "..2122". In the meantime, a concurrent new insertion that goes into node "..212", would not be propagated to the children nodes, resulting in inconsistency. On the other hand, if we update the outlook value of node "..212" before starting the copy, we will have all simultaneous insertions propagated to its children. As a downside, the new outlook would also allow queries to go directly to nodes "..2121" and "..2122", which do not contain the whole data yet. The result would be that concomitant queries would have missing data.

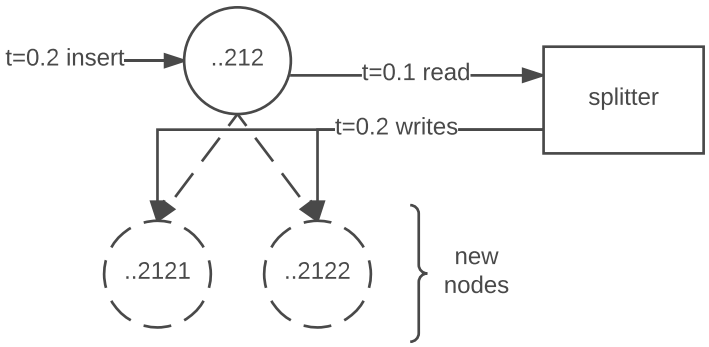


FIGURE 4.9: Possible Lost Update (P4) during copy.

A straightforward implementation requires a 2PL strategy: locking the cube while the data is read and copied into the children. A lock would put on hold all read and write operations on the part of the distributed index with an unsustainable performance cost. To avoid this situation, we designed a protocol that allows lock-free data copy and index creation.

As said before, when ingesting new data our system builds a sub-optimal version of *OutlookTree* that requires additional steps during a read. The proper *OutlookTree* is then lazily generated during reads. After a query, as we have already retrieved data from the disk, we perform a background *ReadOptimization* that redistributes the data, thus reducing the overhead of future read operations.

The consistency of the whole system is maintained by the *RangeEstimator*. By changing the r_{to} in a timely matter, we guarantee that all items inserted go directly to the new children while we are "read optimizing" a cube. Using the example in Figure 4.9, we ensure that while the splitter is copying the data from "..212" to

its offspring, the *RangeEstimator* propagates all concurrent new insertions also to "..2121" and "..2122". However, this change does not influence read operations, nor prematurely changes the node's outlook. We do so by "announcing" a cube to all servers of the cluster before visiting it. If all nodes acknowledge it before we retrieve the cube, we can optimize it.

To correctly implement this mechanism, we must distinguish between five states of a cube:

leaf: a cube that has not reached its maximum capacity yet.

full: it has surpassed its target capacity and thus it will store only the new values with a priority lower than a specific value, the *write_rand*.

announced: A query is going to visit a full cube, and we might optimize it. To avoid Lost Updates, we must ensure that r_{to} includes the cube's children so that concurrent insertions do not get lost.

replicated: The cube has been optimized. It has a *committed outlook*.

visitable: The cube contains all the ancestors' elements that enter its domain. A cube is visitable if it is either the root of the index or if all its ancestors up to the root are replicated.

To describe the four states of a cube, we use three variables:

write_rand: the priority of the last item that fits into the cube.

announcement_time: the time all servers acknowledged the cube's announcement.

committed_outlook: a variable that keeps track of which *ReadOptimizations* occurred.

The *write_rand* is calculated by taking advantage of all the times the database has to sort data to maintain its Log-structured merge-tree (LSM) architecture. When the Compaction completes, we broadcast the new write_rand values to all servers. A cube is stored in a single primary server, but it can have replicas; in case of conflicting values, the smaller write_rand wins.

4.4.1 Querying the AOtree

The first step in executing a query is to calculate its area of interest, and then we find the cube with the smallest domain that can contain all searched information. We call this cube the Minimum Bounding Cube, MBC. In the case of the D8tree, queries can start directly from the MBC or any of its descendants as the data is replicated up the index's max-height. Diversely, in the AOtree data is only eventually and opportunistically replicated, so we must ensure that the query starts from a "visitable" cube.

A naive way to ensure we do not miss any entry is to start all queries from the root cube, and once we know its outlook, we can proceed down to its replicated

descendants. The downside is that the root node - and similarly the higher nodes of the tree - could become a performance bottleneck in the system. We avoid so by storing into an arbitrary sized in-memory trie the outlooks of the highest index's nodes. As the outlook has monotonic crescent value, we do not have to worry about inconsistency as a missing or outdated value would only cause to visit a higher-than-needed number of index nodes.

The in-memory trie provides the known *committed level* for the area described by the MBC, which could be smaller, greater or equal to the level of the MBC. In the first case, we cannot ensure that the MBC is visitable and we must find the first ancestor that it is and start from there. If it is equal, the MBC is visitable and thus we can directly start the query from it. In the last case, when the *committed level* is greater than the MBC, we are free to decide from which level downward to start the index exploration.

For example, if our MBC "012345" is visitable, but we know that part of the tree has a *committed level* of 2, we will first have to visit cube "012", from which we will gather the correct *committed level* and thus we will proceed to cube "012345". On the other hand, if we know the *committed level* is 7, we could also start with cubes "01234500", "01234501", etc.

In case we are free to decide from which level of the tree to start our exploration, we need to take into consideration three main aspects.

A replication constraint: we must stay within the committed level.

A cubes estimation: if we can estimate the number of results of a query, we can calculate the minimum number of the nodes we will have to visit to find all the required data.

A maximum parallelization threshold: as multiple queries can run at the same time; we must ensure system stability by limiting the number of concurrent requests.

Cubes domain estimation

An essential step in planning the exploration of the AOtree index is understanding how many cubes we will need to visit to gather all the data we need. While we proceed with the index navigation, we can increase the precision of our estimation. We can use the write_rand of each visited cube to estimate the fraction of elements that it contains over the ones that fall into its domain and that its offspring store. We can estimate the percentage of data contained in a cube as:

$$cube_{\%} = \frac{write\_rand}{MAXIMUM_{write\_rand}} * 100$$

For instance, if the write_rand were store as a byte, the $MAXIMUM_{write\_rand}$ would have been 255. Therefore, a cube with write_rand 123 would have $cube_{\%} \approx 48\%$.

Once we have the query required precision $query_{\%}$ and the $cube_{\%}$, we can calculate the minimum number, the lower bound, of the cubes we must visit to satisfy a query given the index dimensionality **D**. We can calculate the $cubes_{LB}$ using the following formula:

$$cubes_{LB} = \lceil \log_D (\frac{query_{\%}}{cube_{\%}}) \rceil$$

For example, in a 3 dimensional index (D=3), if we have visited cube "0123" that contains 1% of the data in its domain, and we are looking for 10% of the data in that area,

$$cubes_{LB} = \lceil \log_3 (\frac{10\%}{1\%}) \rceil = \lceil 2.096 \rceil = 3$$

Therefore, in this case where the *committed level* ≥ 6, with 6 being the sum of the $cubes_{LB}$ and the level of the starting cube "0123", we can freely decide to continue to the cubes in level 6, "0123***", if that respects the maximum parallelization threshold.

Overall process summary

When we put all together, the overall process can be can be summarized in the following steps:

1. If a cube reaches its size limit, the Compaction will eventually calculate its write_rand, and it will propagate it to all nodes. The cube changes from **leaf** to **full**.

2. At query time, the *QueryCoordinator* (QC) decides from which cubes to start. If the first cubes to visit can be optimized, the QC announces to all servers which cubes could be "optimized". The time all nodes acknowledged the fact the cube might be replicated in the future is stored as a *announcement_time*. The cube is now **announced**.

3. During a query, we put into the *ReadOptimization* queue all cubes we read that can be optimized.

4. The *ReadOptimization* process picks a cube from the queue and checks if the data has been retrieved after the announcement time. If so, it propagates the data to the descendants. Otherwise, it drops the operations as we could miss concurrent updates.

5. Once the *ReadOptimization* completes, the node committed_outlook is set to 1 and the cube is **replicated**. From this moment forward, the Compaction can drop all elements with a priority greater than write_rand. Also, queries can decide to skip the node and access directly to its offspring.

6. When a cube updates its committed_outlook, it notifies its father. Each cube keeps tracks of the sons' completed replications. When all sons achieved a minimum committed_outlook of K, the father committed_outlook increases of K, and the information propagates to the ancestors recursively.

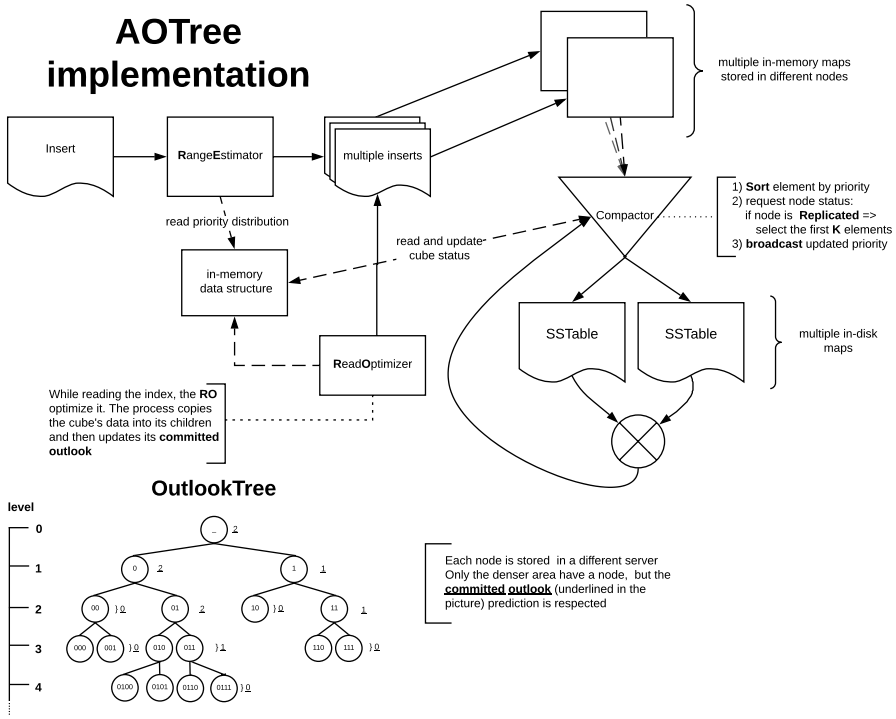


FIGURE 4.10: The OutlookTree implementation schema.

4.4.2 Distributed transaction

Any distributed system has to deal with two main problems: message delivery and peer consensus. The first concerns the amount of overhead we are willing to pay to ensure a message we sent has been received. In increasing order of guarantees - and therefore overhead -, a message can be delivered *"at most once"*, *"at least once"*, and *"exactly once"*. While the first has no overhead, the other have a more complex logic, a higher memory footprint, and they require sending additional messages.

The *RangeEstimation* uses the write_rand to calculate the r_{from}. As long as the write_rand is monotonic decrescent, if a server loses an update, it will observe a value greater than the real one. Consequentially, it will estimate a smaller r_{from}, resulting in an additional operation but not in data loss. For such reasons, these messages can be delivered in a best-effort manner, without requiring acknowledgment, in a *"at most once"* manner.

Let us suppose that we are inserting two almost identical elements i_1 and i_2 that should go into $cube_{012}$. The two operations are processed by two different servers, n_1 and n_2 that have inconsistent metadata as they lost part of the communication.

Therefore, the value of $outlook(cube_{012})$ differs in the two nodes: for example we might have $o_1 = 2$ and $o_2 = 1$. As a result, i_1 will be propagated to $cube_{0123}$ and $cube_{01234}$, while i_2 only to the $cube_{0123}$. In both cases the data is stored in at least one node, but as the outlook is used also to create the query plan, a query executed in n_1 might decide to skip $cube_{0123}$ and read directly $cube_{01234}$, therefore missing i_2.

To avoid this situation, the actual implementation of the *RangeEstimator* (RE) uses a combination of various data structures keeping track of the cubes that have been announced or replicated. Starting from the root, the RE searches for the first cube that is not in the replication_set and that can contain the update ($priority(item) < write\_rand$).

Differently from the write_rand, we cannot underestimate a cube's outlook as it could result in data loss. To ensure we do not lose data performing a *ReadOptimization*, when a node adds a cube to the replication_set we need all peers to acknowledge they have done the same. In this case, we need *"at least once"* delivery, as we need to know when all servers have received the update, but duplicated messages pose no threat. By knowing the time all nodes have received the last update that changed the outlook, we can avoid any inconsistency that may happen during a *ReadOptimization*. Indeed, if part of the servers observe a different outlook we might occur in the previously cited problem of Lost Update (P4).

The second problem is consensus, as a cube could have replicas and different servers can try to update the write_rand value. However, in such situation we do not need complex algorithms as the Paxos protocol described in Section 2.2.4. Indeed, in this case we can simply use the rule in which the smaller value wins as write_rand has a monotonic decreasing value.

4.4.3 Memory footprint

To achieve high-performance any database system needs to reduce at a minimum the quantity of I/O necessary for each operation. To this end, modern databases optimize the way the data is indexed and stored to the disk to maximize the throughput, while durability is guaranteed by grouping many operations in a single write into the commit log. As a result, each insertion results in less than one I/O operation on average. Designing our system, we did not want to increase the I/O, and therefore we need to keep in-memory a small set of information about the index structure to ensure its consistency. The *RangeEstimation* requires few information to be replicated in each server; thus even if we increase the number of machines in our cluster, our index might be limited by the amount of memory on the single computer, hindering the scalability of the system. Therefore, reducing the size of the globally shared information is paramount.

A straightforward implementation of the *RangeEstimation* would need to know the last write_rand and committed_outlook for each cube to calculate both the r_{from} and r_{to}. Even though the quantity of information for each cube is relatively small, each server would need to have a full copy of write_rands of the whole cluster, thus

limiting the system scalability. To avoid so, we studied a novel in-memory structure that could approximate the *RangeEstimation* with a smaller memory requirement.

To preserve the index consistency, we must ensure that

$$\overset{\approx}{r}_{from} \leq r_{from}$$
$$\overset{\approx}{r}_{to} \geq r_{to}$$

where $\overset{\approx}{r}_{from}$ and $\overset{\approx}{r}_{to}$ are the approximation of r_{from} and r_{to}. Indeed, if we underestimate the r_{from}, the system will propagate the data to a cube that cannot accommodate it; thus the Compactor eventually will remove it. When overestimating the r_{to}, we will insert the element in a node where it is not yet reachable by any query. It is a temporary waste of space, but it can be recovered either if the index grows or during compaction. In both cases, no data is lost.

Recalling the outlook definition 4.2, the outlook increases when the write_rand decreases. Therefore, if we want to ensure $\overset{\approx}{r}_{to} \geq r_{to}$, we need $\overset{\approx}{write}\_rand \leq write\_rand$. On the other hand, to guarantee $\overset{\approx}{r}_{from} \leq r_{from}$, we need $\overset{\approx}{write}\_rand \geq write\_rand$, which is in contrast with what the r_{to} requires.

For this reason, we decided to use two different functions to calculate $\overset{\approx}{r}_{from}$ and $\overset{\approx}{r}_{to}$. We will call the r_{from} estimator $\hat{r}_{from}$ defined as:

$$\hat{r}_{from}(i) = \min\{j : \forall cube_j \ni i \wedge priority(i) \leq \hat{\phi}(cube_j)\}$$
$$\hat{\phi}(cube) = \overset{\approx}{write}\_rand : \overset{\approx}{write}\_rand \geq write\_rand$$

where i is the item we are going to insert in the index. On the other hand, the r_{to} estimator $\hat{r_{to}}$ is defined as:

$$\hat{r_{to}}(i) = \max\{\hat{\theta}(cube_j) : \forall cube_j \ni i\}$$
$$\hat{\theta}(cube) = \overset{\approx}{outlook} : \overset{\approx}{outlook} \geq outlook$$

The advantage of this formulation is that we can implement $\hat{\phi}$ by keeping in memory only a subset of an arbitrary size of the write_rand values. For instance, we can keep the smallest ones, as they are the ones that ensure the greatest I/O save as they allow avoiding the greater number of insertions. Secondly, a smaller write_rand indicates a denser area and thus we have a higher probability that future insertions will interest that zone.

Similarly, we can use for $\hat{\theta}$ any Approximate Membership structure, such as the Counting Quotient Filter (CFQ) [82] or Bloom Filters [25]. In such a way, we can arbitrarily reduce the memory footprint at the cost of a higher indexing overhead.

Furthermore, the data used by the *RangeEstimation* can be integrated with the outlook trie used by the *QueryCoordinator* to reduce the space used in the replication_set. If a cube falls into the committed_outlook of another cube, we already know that it is a *replicated cube*, thus we can avoid to store it in the replication_set.

For instance, if cube "012" has committed_outlook 5, we can avoid to save all cubes "012*****". In a 3D index, this could potentially save $8^5 = 32768$ entries.

The committed_outlook format is a binary structure that keeps track of which descendants has been optimized. It has two parts: a global counter that indicates the committed_outlook, and an offspring mask that keeps track of the children delta outlook. When a *ReadOptimization* completes, the counter equals 1. If later all the cube's children get optimized as well, the counter is set to 2, meaning that all the data contained in the cube is also replicated two levels downwards. The second part is the offspring mask: it allocates few bits to store the outlook of the children nodes. In the current implementation, we use 4 bytes: 1 for the counter and 3 for the mask. In a 3 dimensional index, as each cube has $2^{dimensions} = 8$ children, we can allocate 3 bits for each son, so that we can keep track of a delta outlook up to 7. The higher the dimensional cardinality the fewer bits can be allocated for each node. However, as long as high-dimensional indexes have a larger fan-out, the tree grows shallower with smaller outlooks.

4.5 AOTree testing

This section contains the tests we ran to validate the performance of the AOtree implementation provided by the new version of Qbeast. At first, we will introduce the scalability results generated by an open source benchmark tool. Secondly, we will discuss the performance of a real HPC application that has been adapted to use our system, focusing on its performance profile and the problematics involved in integrating an MPI based code with a TCP based database. Lastly, we will propose a performance comparison of the time required to run the HPC application using as storage Qbeast, Cassandra, PostgreSQL and a single file on GPFS.

We ran our tests at the Barcelona Supercomputing Center, in the Marenostrum IV. Each one of the computing nodes we used contains two sockets with an Intel Xeon Platinum 8160 24C processor at 2.1 GHz for a total of 44 cores for a node. The primary memory is composed by 12x8 GB DDR4-2667, 96GB in total, with a ratio of 2GB per core. All nodes are interconnected with a 100Gb Intel Omni-Path Full-Fat Tree and a 10Gb Ethernet [1]. We stored the database data into the local scratch, a SATA 240GB Intel s3520 SSD. The disks are rated for sequential read up to 320 MB/s; sequential write up to 300 MB/s, random read 65000 IOPS and random write 16000 IOPS [63]. For comparison, we also stored data in GPFS. Table 4.1 shows how the scratch SSD disks and the GPFS compared in terms of IOPS and bandwidth when increasing the size of a block of write. We ran the tests using the fio [40] benchmarking tool.

We can see how the two storage devices behave differently; the SSD has better IOPS for small size writes, while stripping layout of the GPFS ensures better throughput for large sequential writes.

	Block size		
	4k	64k	64M
GPFS	1337 IOPS - 5.5 MB/s	1058 IOPS - 69.4 MB/s	25.42 IOPS - 1719 MB/s
scratch	27071 IOPS - 98.6 MB/s	16250 IOPS - 99.8 MB/s	4.77 IOPS - 321 MB/s

TABLE 4.1: Performance comparison of GPFS and local SSD disks in Marenostrum IV.

4.5.1 Synthetic tests

Our first tests aimed to estimate a lower and upper bound for the performance of our system. Therefore, we tested an increasing number of database nodes versus a set of clients that were performing random insertions. We used the Cassandra stress test tool to perform the benchmark. We configured the system to perform random insertions with a Gaussian distribution, inserting in a table with the same structure as the one we used to store the results of the Alya simulation. The model used as partition key the particle identifier and clustered the time. The rest of the values are x, y, z position, speed, accelerator and other physical characteristics of the particles for a total of 15 double and 3 integer number.

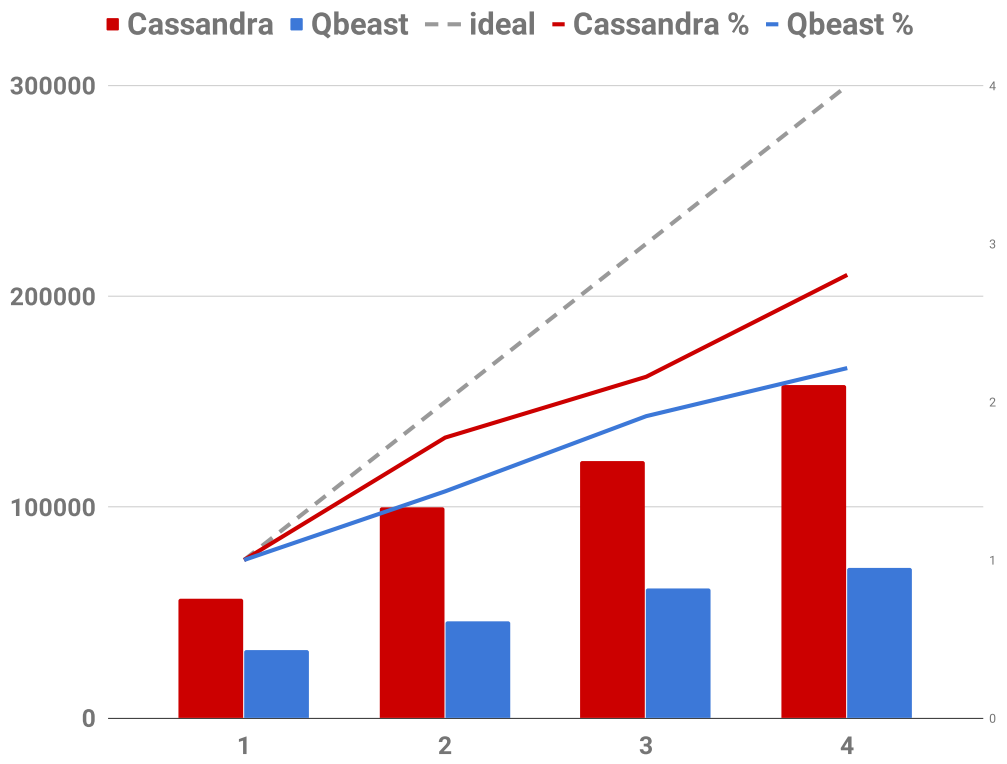


FIGURE 4.11: Cassandra and Qbeast IOPS per node and relative speedup.

On a single Marenostrum node, we achieved $\approx 56K$ IOPS with Cassandra and

$\approx 32K$ IOPS with Qbeast, which is the expected performance as long as we are indexing and thus for each insertion in the main table we have to perform at least one additional insertion on the index table. However, we see that neither Cassandra nor Qbeast present perfectly linear scalability. In the case of Cassandra, we suspect that the difference is mostly due to our testing environment, as we used an equal number of clients and nodes to stress the tests and thus it might be possible the Cassandra simply outperformed the clients. On the other hand, Qbeast shows suboptimal scalability that we suspect is caused by the use of logged batch. Indeed, the current implementation of Qbeast uses a combination of Triggers and logged batches insertion that, as described in Section 5.3, adds a significant overhead both in terms of network and IO.

4.5.2 HPC integration

A nontrivial task is integrating an MPI based application with an asynchronous TCP-based protocol. There are two main problems. The first concerns how to handle the asynchronous communication with a high enough level of parallelism that can exploit the distributed database and thus achieve good performance. To this end, we used the C version of Hecuba [2] an HPC oriented library that we develop in our research group at the Barcelona Supercomputing Center. Hecuba allows efficient use of NoSQL databases in MPI oriented applications by taking care of all the callback and asynchronous messages management. Internally, it keeps a limited size queue of objects to be inserted, so that it can issue many parallel queries that can uniformly distribute among servers. Our implementation also ensures all the data has been inserted before the execution completes and supports self-limiting strategies to avoid collapsing the database under a too heavy workload.

The second aspect concerns how to mix HPC code designed for a single application per core and the database driver that uses a thread pool design. Figure 4.14 shows three traces of an execution of the first timestamps of the Alya simulation that stores data with Qbeast. On the vertical axis we see the different MPI workers, while the horizontal colored bars describe the type of operation the worker was doing in a given period. The red indicates the worker is waiting for an input from other workers, while the blue represents active computation. The upper trace in Figure 4.14 shows that most of the time all workers are waiting for one considerably slower worker to complete. In a physics simulation, it is common to split the space into smaller parts, so that each worker can focus on its domain. After each timestamp, workers share information regarding the particles that moved from a domain to another. The downside of such an approach is that particles may be concentrated in the specific area during part of the simulation.

In our specific use case, we used Alya to study the efficiency of drug inhalers, so the code begins with all drug particles residing in a limited area. Thus, we have most of the data in a few workers and consequentially an initial unbalanced workload.

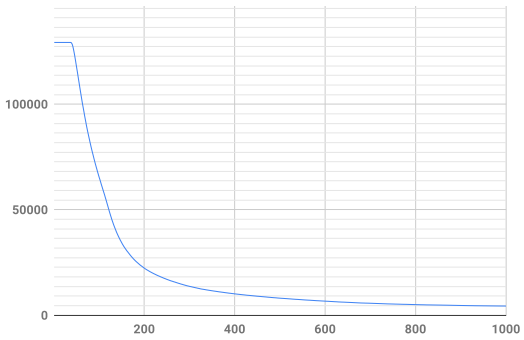


FIGURE 4.12: Particle deposition.

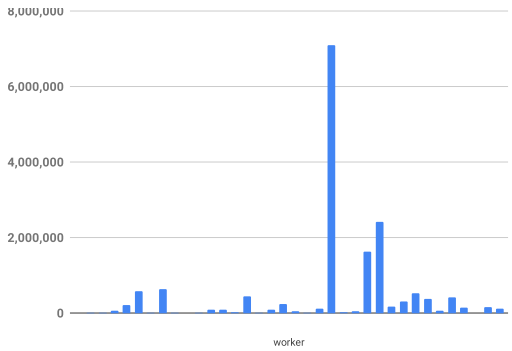


FIGURE 4.13: number of writes per worker.

This effect is visible in Figure 4.13 that shows the number of particles written for each MPI worker. Another interesting behavior of this class of physics simulation is shown in Figure 4.12: particles can deposit on human tissue and thus the number of particles decreases as time goes by. From our point of view, these two characteristics - the initial unbalance of work and the decreasing number of particles - indicate that the most critical parts for I/O are the first few timestamps. Therefore, in our tests, we will focus only on the initial part of the problem as long as it is the most I/O intensive.

Our first integration of Alya and Qbeast showed a poor performance that could not be completely justified by the imbalance of the physics problem. Indeed, in a typical scenario, Hecuba can distribute the insertions on more servers, thus mitigating the unbalance. However, this did not happen with Alya: when a single worker is writing to the database, all other workers are consuming CPU as they use a busy spin strategy to ensure low latency when receiving data for the network. Therefore, the single worker cannot parallelize the insertions as it has to compete for CPU resources with the other nodes.

Figure 4.14 shows the Paraver traces of the first 30 seconds of three different configurations of Alya. Each line represents the state of a worker: blue indicates the worker is computing useful work, red shows the worker is waiting for a message from another worker, while orange suggests the worker is performing an MPI communication task. The numbers indicate when Alya completed persisting a timestamp. The first one on the top shows our first implementation. It is easy to see how the last worker on the bottom is the one slowing down the whole execution. If we measure the time required for the single worker to complete and we divide it by the insertion performed, we can estimate that the system achieved approximately 17K IOPS, which is considerably less than the IOPS that the Qbeast cluster could sustain in the tests we executed. The trace in the middle shows the performance of Alya when disabling the busy spin. It is easy to see how the time required to write to Qbeast reduced from approximately 7.2 seconds to 2.4, which results in $\approx 53K$ IOPS which is an appropriate performance for the cluster we are using. The downside of such an approach is that the cost of an MPI communication increases, eventually

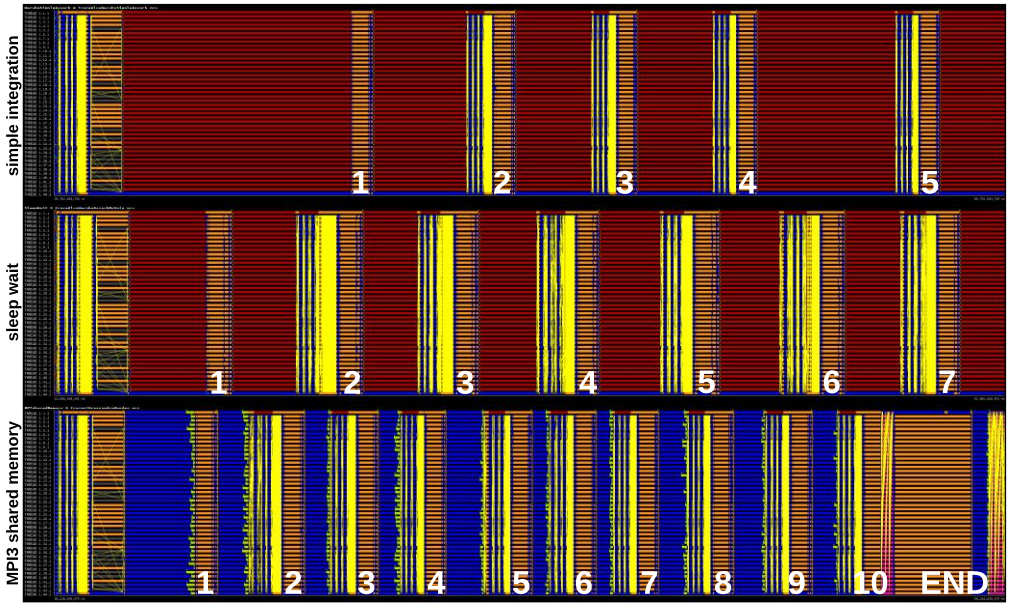


FIGURE 4.14: A bar char of the number of writes per worker.

counterbalancing the advantage of a faster I/O. Indeed, while the second trace is faster in the first ten steps, the first implementation requires less time on more extended runs. For example, our measurements show that the average time the worker spent in the *MPI_Allgatherv* increased by factor 2.4. We can visually observe this slow down by comparing the width of the orange phases in the upper and middle traces in Figure 4.14. To improve I/O without sacrificing the whole execution, we used a hybrid approach. We decided to introduce an additional step in which we shuffle the data between workers on the same node so that each one participates equally in the writing process, and we can take advantage of the used CPU resources. Figure 4.14 shows how this approach pays off in terms of performance. Indeed, while the first configuration completed 5 and 7 timestamps respectively, by using the MPI shared memory shuffling, we finished all the 10 initial steps. This also indicates that more than being limited by the performance of the database, the overhead of sending data is often the major performance bottleneck.

Using the MPI shared memory adds a synchronization step between workers which adds $\approx 27ms$ to each time step, an increase in the I/O time of around 5-7 %. Using shared memory is a sub-optimal solution but it serves the scope of our tests as the general goal is to reduce the number of synchronizations required for I/O. In the future, we will investigate more flexible solutions such as the integration with dynamic scheduling framework such as DLB [44] to ensure nodes can share CPU resources. An alternative approach is to use MPI to send data directly to the database, but it requires to migrate the database architecture from a thread pool design to a worker MPI layout.

Figures 4.15 and 4.16 report the performance of the different backend we tested

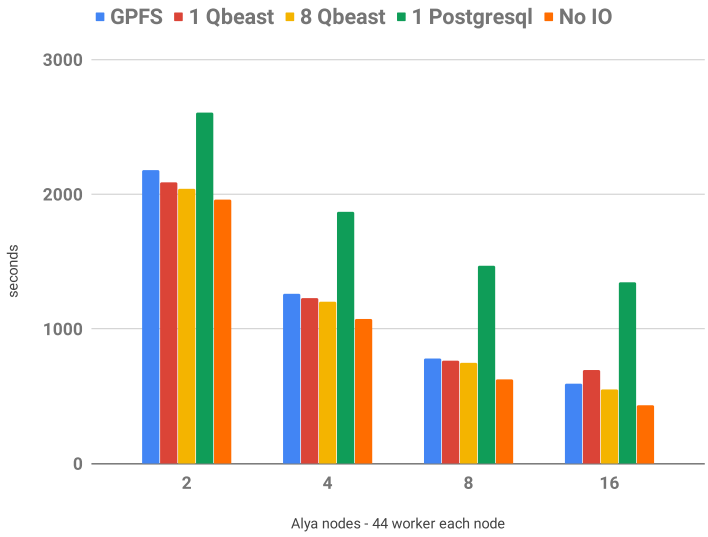


FIGURE 4.15: Time for 1000 steps in Alya.

increasing the number of workers. Figures 4.15 shows the total time when using as backend a file on GPFS, one node of Qbeast, eight nodes of Qbeast, one node of PostgreSQL and no backend at all. Alternatively, Figure 4.16 shows the net time Alya spent performing I/O. There are several insights we can gather from these results. First, we shall note that Qbeast can store and index data faster than the GPFS can write into a not-indexed CSV file. As the reader may be surprised by such a result, we should clarify that Alya uses a master-slave approach to output data into an ASCII file, which is arguably not the most efficient format. However, as we discussed in Section 2.5 an alternatives is MPI/IO which is not the perfect solution either. Indeed, the number of particles changes during the time due to the deposition and they can move from a domain and another, thus making complex to use of Hyperslabing, which would in any way eventually require to shuffle all data between nodes. Alternatively, each worker could write independently in a different file, which would probably be the faster solution, but then the user would have to merge and reassemble the results in a second phase. In any case, the point we want to prove is not that our system can be faster than file storage, but that in applications where the output file is required to have a specific structure to facilitate the analysis, our system can compete, if not be faster, then mere files. Another interesting result is that time required for I/O for one Qbeast node or eight is not proportional, as the second is not one-eighth of the first. Also, by changing the number of Alya workers, the I/O time remains approximately constant, except for one node of Qbeast with 16 Alya nodes that registers a considerable decrease in performance. Such a spike in performance suggests that our system starts to suffer when ratio between database and application nodes is higher than 1:16. At the same time, we see adding more Qbeast nodes does not linearly reduces the number I/O time, suggesting that in our implementation, either in the MPI3 shuffling or in the database communication

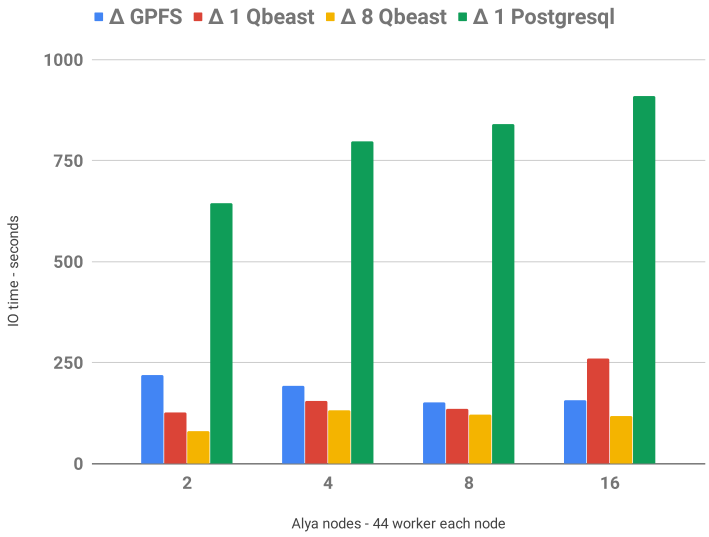


FIGURE 4.16: Net I/O time by backends.

there is a performance bottleneck that we might investigate in future works.

In the Figure 4.15 and Figure 4.16 we can also see the performance of PostgreSQL is considerably lower while ingesting a high rate of writes while building a PostGIS multidimensional index. The speed also decreases when increasing the number of concurrent actors. For a fair comparison, we used the same MPI3 shared memory approach and prepared statements for PostgreSQL. We also store the data in the scratch SSD, as PostgreSQL showed better performance on SSD than GPFS. To improve the throughput, each worker commits its chunk of the insertions after each timestamp, so that the driver can optimize the writing of the single particles. We did not test a distributed version of PostgreSQL as the set up of such a system in a queue based system such as Marenostrum IV can be challenging and it would not have brought additional insights. Indeed, even in this case, PostgreSQL could linearly scale, we would need three PostgreSQL nodes to match GPFS, and five instances to match one node of Qbeast and an HPC scenario in which it might be considered convenient to deploy more storage nodes that application is rare.

To evaluate the performance when reading the data, we used a similar methodology to the one we used in Chapter 3; we selected three typical queries that scientists use when exploring the result of our particle inhalation problem. At first, scientists need to have an overall look at the whole simulation. Secondly, a relevant query is to see which particles deposited in a specific area of the nasal cavity, the olfactory region, where drugs get absorbed faster. Lastly, it is interesting to check how the particles get expedited from the nozzle of the inhaler.

Given the size of the areas of interest, we will gather only a sample of the data. More precisely, a sample of 0.01% of the whole simulation, while 1% of data in the other two queries. In the following, we will identify the queries as “all 0.01%”, “olfactory 1%”, “inhaler 1%”. With such configuration, the three queries return 20,

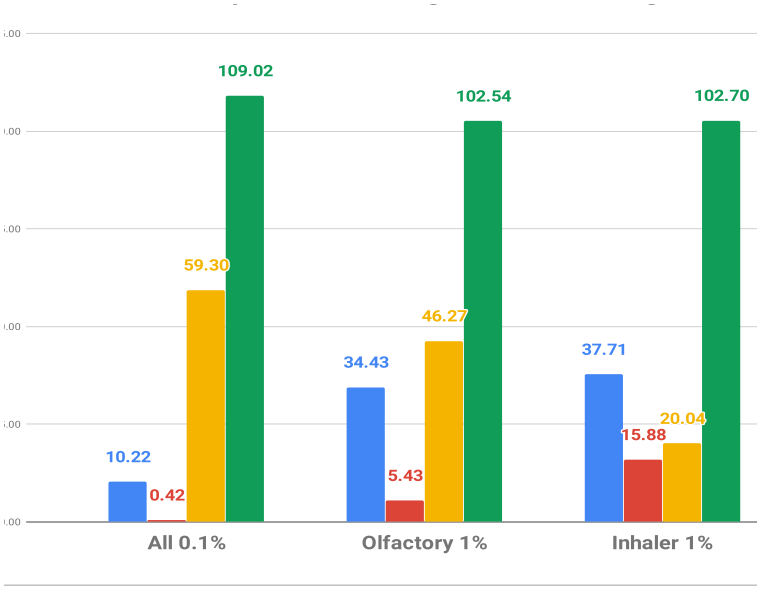


FIGURE 4.17: Qbeast and PostgreSQL response time for the three example queries.

	speedup	optimization runs	iterations	cube visited
All 0.01%	24.51	6	2	10
Olfactory 1%	6.34	10	19	19
Inhaler 1%	2.37	8	61	61

TABLE 4.2: Qbeast speedup improvement after *ReadOptimizations* with the relative number of iterations and index cube visited.

27, 200 thousand results respectively.

Figure 4.17 compares the response time of Qbeast, when the AOTree is optimized and when it is not; and PostgreSQL, either using a 3D or a 4D secondary index. We can see how Qbeast always outperforms PostgreSQL with the optimized AOTree. Also, even with the not-optimized AOTree, Qbeast is faster than the PostgreSQL on two queries out of 3.

Table 4.2 shows the performance improvments of the AOTree after multiple *ReadOptimizations*. The table shows how many Read Optimizations run before optimizing the part of the index interested by the three queries. It is important to note a RO execution for one query most likely benefits also others, thus reducing the overall number of RO required to achieve good performance. The table reports also the different speedup we can achieve in the three queries, ranging from 24.51 X improvement to a "mere" factor 2.37. In query "All 0.01%" we find the highest speedup as the index gets closer to the OutlookTree. Indeed, in such case, the query completes in just two iterations. On the other hand, in the other two examples, we can see the number of iterations is considerably higher, meaning that the algorithm has to continue to explore the index various times before finding all data. After analyzing the structure, we found out the high number of iterations is due to the fact the current implementation fails to increase the outlook of cubes with empty children. Indeed,

as the empty children do not get optimized, they prevent the increasing of the father outlook. As a future work, we will solve this bordering situation and thus we expect even higher performance speed-up for these queries.

4.6 Summary

In this Chapter, we described the performance limitation of the D8tree when dealing with write-intensive applications and we studied how to overcome such constraints proposing a new theoretical index, the OutlookTree. The OutlookTree reduces the disk space and transactional requirements without compromising the performance of reading queries. However, the OutlookTree cannot be built under heavy load as long as any outlook change requires to lock large portions of the tree while moving data from a server to another. Therefore, we proposed the AOtree, which builds a sub-optimal index that uses various optimization strategies to converge to the OutlookTree structure eventually. This Chapter also describes the experiments we performed on the AOTree that demonstrate it is possible to use in HPC advanced functionalities such as MI indexing and sampling with no performance cost, or even improvements according to our tests.

4.7 List of publications

In preparation: Cugnasco, C., Calmet, H., Santamaria, P., Gil, E., Sirvent, R., Becerra, Y., Torres, J., Ayguadé, E., Labarta, J. (2019). Qbeast, the HPC multidimensional database. To be submitted to ICPP2019

The content of this Chapter has been used in the European patent request EP18382698.1, with title DISTRIBUTED INDEXES, applicant BARCELONA SUPERCOMPUTING CENTER - CENTRO NACIONAL DE SUPERCOMPUTACIÓN, and author Cesare Cugnasco and Yolanda Becerra.

Chapter 5

Qbeast

This Chapter describes the forth and last contribution of this thesis: Qbeast. Qbeast is a distributed software that implements the D8tree and the AOTree in symbiosis with Apache Cassandra. Indeed, to ensure the proposed novel algorithms are convenient tools for solving HPC problems, we had to test these algorithms and architectures in realistic scenarios. Two main approaches were possibles. A possible method is to create a mock version of a database with only a few functions available. A mock system has the advantage that it is simpler to understand what influences the overall performance, as there are fewer components to interfere. On the other hand, the results are less significative as long as a real-world application would require the missing features and thus potentially change the system behavior. Therefore, we opt to integrate our algorithms within a widely adopted database and we reused part of the code and architecture of Apache Cassandra.

In this chapter, we will focus on the implementation details, the architecture and the trade-off design decisions we had to take to implement and test the D8tree and AOtree. In particular, we will describe the two main versions of Qbeast, the distributed indexing system that builds and allows to query our indexes. We will call QbeastV1 the versions that create and query D8tree indexes, while QbeastV2 works on the AOTree.

We will also describe Qview, the ParaView plugin we developed to query and visualize the results stored in Qbeast.

5.1 Overall architecture

While theoretically, the process of the query could run in a completely distributed manner, with the first node only acting as an initiator of a chain reaction, we had to centralize the process for technical reasons. The main problem is that in a distributed execution it is complex to tell apart a completed query from one that is waiting for delayed results. As a naive solution, we could simply wait enough time to ensure all sub-queries completed, but that would considerably slow down the execution. As future work, we are considering a more distributed approach to the problem in which we use the index tree structure to spread the responsibility of controlling the process of the query. Also, as we analyzed with teh analytical model in Section 2.12,

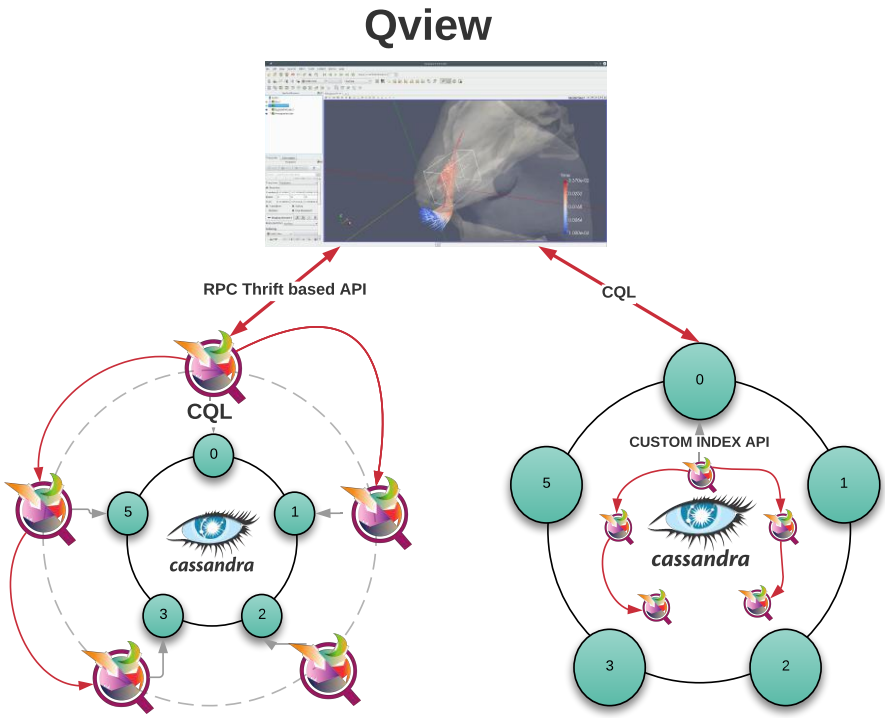


FIGURE 5.1: A broad view of QbeastV1 vs QbeastV2

a single query coordinators limits the perfromance only for cluster of 70 or more nodes.

5.2 Data gathering

The data retrieving phase is not shown in Figure 5.1 for a matter of space. There are two main approaches, a centralized one and a distributed one. In the first one, the client can initialize the query in any of the Qbeast nodes, which then becomes the query coordinator managing all the communication with the nodes. All Qbeast nodes forward the data to the coordinator - on server 0 in the example, which then forward the required data to the client. The advantage of such an approach is that the API is straightforward for the client, as it can simply use an RPC call, wait for the result and gather it. However, when a query grows in size we start to experience problems of memory as the initialization node has to hold all results and it can, therefore, crash on a heavy workload. Also, it adds the overhead of sending, serializing and deserializing the data additional times.

5.2.1 Custom secondary index

In QbeasV2, we used a different approach, and we integrated Qbeast inside Cassandra so that both codes live in the same JVM and they can access the same in-memory object without the need to serialize or network overhead. Furthermore, we abandoned the custom Thrift protocol as we aimed for tighter integration with

Cassandra. Indeed, we used the modular architecture of Cassandra to develop a custom secondary index so that both insertions and query can use the same CQL language and drivers. As long as Qbeast allows randomly sampling, which is not a feature contemplated by the CQL standard, we had to use as a workaround a custom expression. Listing 5.1 shows an example of how using QbeastV2 we can create a table, an index on a subset of its columns, and insert or query data from it.

LISTING 5.1: An example of QbeastV2 schema creation and query

```
CREATE TABLE particle(
  pid int,
  time double,
  x double,
  y double,
  z double,
  PRIMARY KEY (pid, time)
);
CREATE CUSTOM SECONDARY INDEX qbeast_idx
 ON particle(x,y,z) USING 'es.bsc.qbeast.QbeastIndex';
INSERT INTO particle(pid,time,x,y,z)
  VALUES (1,0.2,0.4,0.5,0.6);
SELECT * FROM particle
WHERE x>0.1 AND x<0.9 AND
      y>0.1 AND y<0.9 AND
      z>0.1 AND z<0.9 AND
      expr(qbeast_idx,'precision=0.3')
ALLOW FILTERING;
```

Using the secondary index interface, Cassandra sends back the data to the user and distributes the query to the different nodes. On this side, the client can use two different approaches. If the client simply issues a query to a single node, this node will forward the request to the others, gather the results and forward to the application with a similar matter we discussed before. This approach is useful for small queries, but for larger ones a better way is to gather the data directly from each Qbeast node. We can do so by splitting the query according to the DHT token ring, so that we can send a disjoin query partition to each single node and avoid additional network transmission.

LISTING 5.2: An example of token range partition in QbeastV2.

```
SELECT * FROM particle
WHERE x>0.1 AND x<0.9 AND y>0.1 AND
 y<0.9 AND z>0.1 AND <<0.9 AND
expr(qbeast_idx,'precision=0.3') AND
TOKEN(partid) >= 0 AND TOKEN(PARTID) < 100
ALLOW FILTERING;
```

```
SELECT * FROM particle
WHERE x>0.1 AND x<0.9 AND y>0.1 AND
 y<0.9 AND z>0.1 AND <<0.9 AND
expr(qbeast_idx,'precision=0.3') AND
TOKEN(partid) >= 100 AND TOKEN(PARTID) < 200
ALLOW FILTERING;

SELECT * FROM particle
WHERE x>0.1 AND x<0.9 AND y>0.1 AND
 y<0.9 AND z>0.1 AND <<0.9 AND
expr(qbeast_idx,'precision=0.3') AND
TOKEN(partid) >= 200
ALLOW FILTERING;
```

In Listing 5.2, we can see an example in which we have three nodes: the first node is responsible in the DHT table for all the keys with a hash value between 0 and 100, the second from 200 to 300 and the last the ones from 300 upward. Therefore, the client can send the three queries shown in the listing directly to the node that contains the data avoiding additional data transfer.

However, there is a key difference between the way Qbeast and Cassandra's custom index API work and that can cause some problematics. Indeed, Cassandra assumes that each node has a local file that indexes all and only the entries stored in that node, while in Qbeast the index is randomly distributed.

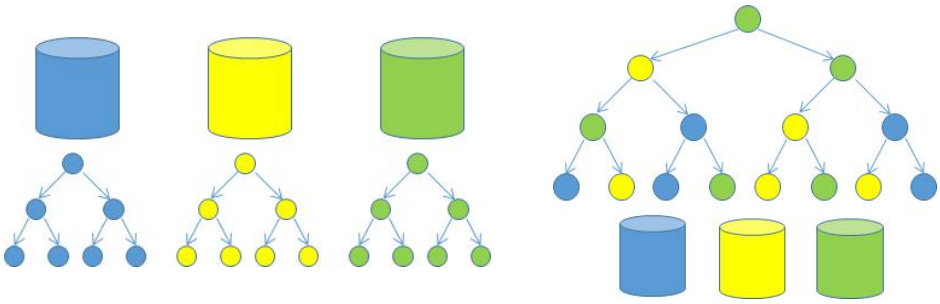


FIGURE 5.2: Locally vs globally stored indexes

Figure 5.2 shows the difference between the two approaches. In the first case, data is randomly distributed between nodes, and each of them builds a local index with only the data it stores. In the second case, we build a global index, and we randomly assign each part of it to a node. The problem of the first approach is that each query gets broadcasted to all nodes. Since database performance is usually limited by the CPU, broadcasting the query means the system cannot scale in terms

of IOPS. The drawback of the second approach is that we must jump from a node to another to complete a query. However, both the D8tree and the AOtree do not have such a disadvantage as the queries can directly access to the node needs, as we discussed in the previous chapters.

An interesting aspect of our implementation is that the Cassandra secondary index API assumes the indexes will all have a local design, thus it propagates the same query to all nodes. On the other hand, Qbeast runs queries distributedly on multiple nodes. Therefore, when Cassandra sends the same query to all nodes, there is the risk of triggering various instances of the same operation. At the moment of writing, we implemented a workaround solution that uses either the query predicate or a user-provided id to tell apart global queries or repeated ones.

5.3 Propagating writes

The simple structure of the D8tree allows the writing process and the reading one to be decoupled. Indeed, once we configured the maximum height of an index, a client can freely navigate through the index without knowledge of the state of a database. At the same time, each insertion has to be broadcasted to all levels of the tree and then, as we discussed before, written to disk only if the compact decides there is enough space to store it in such a space partition.

A D8tree of height K requires K tentative insertions in one cube in each one of the K levels.

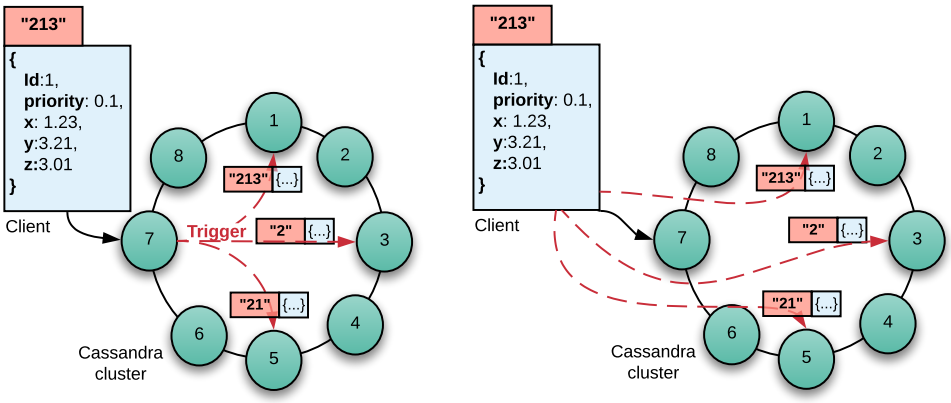


FIGURE 5.3: Different strategies to propagate insertions.

Figure 5.3 shows the different strategies we can employ to store data to propagate K times each insertion. Let us suppose we are writing data on the table particles and the D8tree is built on the table particles_d8tree. In the first case, the client will send a write to the table particle where the Qbeast's trigger will calculate the cube identifiers for all K additional cubes in table particles_d8tree where to send the update. To ensure eventual atomicity, the new mutations generated by the trigger are executed in a logged batch insertion, together with the original one. In such a way,

we can ensure that all updates are eventually persisted. However, such a guarantee comes at a considerable cost.

Indeed, the whole process of inserting using a trigger goes as follows:

1. Node 7 receives the insertion for the table particle and executes the trigger, which converts the simple insertion to a batch one.

2. To ensure that the writes will get eventually stored even if it fails, Node 7 has to send at least two copies of the batch into the local batchlog of two other peers.

3. Once the replicas acknowledge the insertion into the batchlog, Node 7 propagates the insertions to Nodes 1, 3, 5 and waits for their reply.

4. If Nodes 1, 3, and 5 succeed, Node 7 acknowledges the client and sends a message to delete the batchlog of its entry to the two peers.

As a result, using the Trigger with the logged batch in a D8tree of eight K means sending a total of $K * 2 + 4 + 2$ messages, and three communication rounds - client coordinator, coordinator batchlog replicas and coordinator slaves. Also, all the data must be stored in two batch logs so that for each element we perform in total $K * 3$ I/O operations.

In the second case, we have the client sending one update to table particles and K insertions to particles_d8tree. In total $2 * (K + 1)$ messages are exchanged in parallel in a single communication round. Cleary, the second method is more efficient, but it reverses to the client the responsibility to ensure all insertions have succeeded and to retry again in case they are not. Also, in deployments in which the client is far away from the database cluster or it uses a slower network, the first approach might be still favorable.

In Qbeast, we use the first approach for the D8tree, while the current implementation of the AOtree uses the second approach. Indeed, in the AOtree, the trigger uses the RangeEstimator to reduce the number of writes. The RangeEstimator uses some information such as the replication set, and cached outlooks that are stored inside Qbeast and that are not currently shared with the clients.

On the other hand, the additional overhead of the batch is mitigated by the fact that the AOtree requires far fewer copies of the same data, thus alleviating the extra cost of logged batches. As an example, in the Alya use case we usually set the D8tree K to 10, while in the AOtree we observed an average replication of approximately 2. Using the previous formulas we can calculate that for each insertion the D8tree without trigger requires 22 messages and multiplies 11 times the I/O, while the AOtree with the trigger only needs 10 messages and 5 times the I/O. Nevertheless, in future work, we might investigate smarter client driver that can furthermore reduce the message and the I/O to the minimum possible: six messages and 3 times the I/O.

5.3.1 Priority calculation

In both versions of Qbeast, the priority of a cube is computed optimistically by taking advantage of the compaction process of Cassandra. In both cases, we modified the BigTableWriter class of Cassandra to intercept both the sorted compaction and the merge of Memtables and SStables. We take advantage of the already sorted data to discard the exceeding elements in the D8tree. In the AOTree we update the write_rand and also drop exceeding data but only if the cube is optimized. While in the D8tree the cutting strategy is simple, in the AOtree we need considerable coordination between different parts. Indeed, we must consider all the different states that a cube can have in the AOTre.

Furthermore, to alleviate the drawbacks previously described in Chapter 4 and illustrated in Figure 4.4, we also intercept calls that update the Memtables. However, while compaction is a relatively infrequent event, we have an update to the Memtables at each insert which means that when intercepting such calls we must ensure to add little overhead. For such a reason in QbeastV2, the function that decides whether to cut a Memtables uses approximated information to avoid costly infra-thread synchronizations.

5.4 Integration with distributed computing framework

Apache Cassandra and Qbeast offer basic read, write, and index lookups operations, while it does not support any query that requires intensive memory usage, such as aggregations or joints. For such cases, we need an additional system that can perform data manipulation in a scalable and distributed manner, and that can well integrate with the underlying database.

We have integrated our system with two different distributed computing frameworks, Hecuba with PyCOMPSs[106], and Apache Spark[117].

5.4.1 PyCOMPSs and Hecuba integration

PyCOMPSs is a framework developed in the Barcelona Supercomputing Center that aims to simplify the development of distributed parallel applications written in Python. PyCOMPSs is built on top of COMPS and allows users to parallelize sequential applications by simply annotating the Python functions that should run in parallel. The system takes care of all the dependencies between tasks and ensures they are executed on different machines respecting the tasks' dependencies and moving data from a server to another when necessary. Moreover, PyCOMPSs can dynamically adapt the tasks dependency graph during the execution thus better mitigating any work unbalance or improving the performance of algorithms with irregular execution patterns.

In our research group, we created Hecuba, a library that extends the PyCOMPSs capability with additional data management functionalities. Hecuba reduces the

data movement between computing nodes by storing data into Apache Cassandra and Qbeast. Listing 5.3 shows how Hecuba can parse filtering predicates over an indexed collection and push down the filtering to Qbeast that can use its index to reduce the data to filter in memory.

LISTING 5.3: Qbeast and Hecuba integration.

```
from hecuba import *
class Particles(StorageDict):
      '''
      @TypeDef <<pid:int,time:>,ptype:int,x:double,y:double,z:double>
      @Index_on(x,y,z)
      '''
sim1 = Particles("my.sim")

area_of_interest = filter(lambda ((pid,time),(ptype,x,y,z)):
        0.1 < x < 0.3 and 0.3 < y < 0.7
        and 0.1 < x < 0.13 and ptype ==3, sim1.items())

# the filtering predicate gets reduced to
red_filter=lambda ((pid,time),(ptype,x,y,z)): ptype ==3
```

In the example in Listing 5.3, as we are creating a Qbeast index on x, y, z, but not on part_type, Hecuba runs a 3D query on Qbeast and then filters in-memory all elements with a type different than 3.

5.4.2 Apache Spark integration

In addition to PyCOMPSs, we have also integrated Qbeast with Apache Spark due to its up-growing popularity and its versatility. Spark started as a research project at UC Berkeley AMPLab in 2009 and was open sourced in early 2010 moved to Apache foundation in 2013. A wide range of contributors now develop the project (over 1200 developers from 300 companies). Apache Spark is a fast and general purpose engine for large-scale data processing, an improvement to the map-reduce Hadoop model. Apache Spark comes with a vast software environment. Above all projects that grow around MLlib, Spark Streaming, and Spark SQL are the ones more interesting. MLlib is a powerful Machine Learning library that offers a growing set of scalable tools such as classification, regression, clustering, collaborative filtering, dimensionality reduction, and optimization. Spark Streaming[116] allows streaming analysis and data manipulation using the concept of micro batching on programmable windows of time. Finally, Spark SQL[12] allows performing data analysis by issuing SQL queries that are converted into map-reduce tasks by the framework and executed in parallel on Spark. It also supports bytecode generation, which reduces the overhead of pipelining multiple operations as it creates more efficient and cache friendly code.

LISTING 5.4: An example of integration between Qbeast and Spark.

```
val particles=spark.read
```

```
                    .format("org.apache.spark.sql.cassandra")
                    .options(Map(
                        "table" -> "particles",
                        "keyspace" -> "exp1"
                       ))
                   .load()
// API approach
particles.filter('x >= 0.01 && 'x<0.02 && 'y > 0.1 )
    .groupBy('source)
          .count().show()
// SQL approach
particles.createGlobalTempView("particles")
spark.sql("""
SELECT    type,count(*)
FROM    particles
WHERE x >= 0.01 AND x<0.02 AND y > 0.1
GROUP BY type
""").show()
```

We have a prototype that integrates Qbeast with Spark. Our solution uses the Cassandra Spark connector to connect the two systems, with the addition of a push-down optimization that allows filtering operations on the indexed columns to run on Qbeast. Listing 5.4 shows an example of how it is possible to run an aggregational query using Spark and Qbeast. The listing shows two different ways to trigger the query. The first uses the DataFrames API, while the second uses a SQL query that is then interpreted by Spark SQL. In this example, our integration intercepts the predicate " x >= 0.01 AND x<0.02 AND y > 0.1" and sends it to Qbeast, which used the AOTree to reduce the required I/O drastically.

Currently, we are working on also intercepting operations that require a random uniform sample of a data set so that it can run efficiently with a fraction of cost with Qbeast.

5.5 Qview

In this thesis, we have diffusively cited Qview, the plug-in that we use to query and visualize 3D data stored in Qbeast. The first version of Qview used a custom designed RPC API to issue the query in Qbeast and visualize the data, while the latest version is directly integrated with the Cassandra C++ driver, allowing queries on non indexed tables. Qview takes care of opening an SSH tunnel Qbeast and convert the ParaView area of interest in a Qbeast query. Then, it gathers the data and turns it into the ParaView format. Thanks to the integration with Hecuba, the user can also run arbitrary queries that run in the supercomputing, store the results in Qbeast and then use our plugin to visualize them with the desired level of details.

5.6 Summary

We presented two different strategies to implement the D8tree and AOtree, and we discussed the advantages and disadvantages of each one of them. Developing QbeastV2, we took some architectural choices that might have penalized performance but that have notably simplified the integration of our system in existing installations. Indeed, with the current design, it is possible to use Qbeast without the need to rewrite any legacy code. Moreover, it can be easily installed in existing clusters without disruption. Future work will focus on raising the technology readiness level of QBeast by increasing its robustness and stability. Also, we will consider the development of our indexing system for different databases.

Chapter 6

Conclusions

In this thesis, we proved that alternative storage solutions based on NoSQL technologies are possible and convenient when dealing with scientific simulations as they improve performance and user productivity.

Additionally, we proposed Qbeast; a novel distributed system for multidimensional indexes with arbitrary approximated precision that we proved is a viable solution to store, visualize and analyze large scientific simulations.

This document described in detail the path we took to develop the Qbeast architecture, the algorithms that we designed and the research background the guided our investigation.

As part of the first contribution, we analyzed the use of NoSQL technologies for scientific HPC applications, and we found out how important the data model is for performance and scalability and that scientific application requires MIS functionalities to simplify the research workflow. Then, we built an analytical model that allows studying how the data model influences performance in distributed systems. In the meantime, thanks to a comprehensive analysis of the available storage and data management solutions, we concluded that there are no existing solutions that provide scalable MIS functionalities for HPC.

This finding led to the second contribution, the development of the D8tree which proved to be considerably faster of PostgreSQL for a wide array of queries. Then, via Qbeast, we integrated the D8tree with Alya, and we studied how Qbeast improves the scientists' workflow.

As a third contribution, we studied the performance shortcomings of the D8tree with the goal of enabling its use also in write-intensive applications. Thanks to this analysis, combined with the analytical model proposed in the first contribution, we developed the OutlookTree, which reduces the storage footprint of the D8tree without sacrificing the query performance. Finally, we proposed the AOTree, a write and eventually read optimized indexing algorithm that allows building a relaxed version of the OutlookTree without distributed locks and transactions, thus improving query performance even in write-intensive HPC environments. As a final contribution, we discuss some architectural choices we have taken developing Qbeast and how it can integrate with external computational platforms as Apache Spark and

PyCOMPSs.

The AOtree represents the final stage of an evolutionary development that took place in the last years of our research. Its design is the result of years of experience and hard trial and error work, and it is a major achievement for our research.

Indeed, the AOtree demonstrates that it is not only possible to bring to the HPC world advanced functionalities such as MI indexing and sampling, but also that they can come with no performance cost, rather improvement according to our tests.

We believe its design will influence the architecture of the future data storage systems and it will simplify the way HPC simulations are managed. For such reasons, in October 2018 we submitted a patent request to protect the most critical aspects of our architecture. The patent covers both the architecture we presented in Chapter 4 and the developments that we foresee in the future.

Future works While the AOTree is a milestone in our research, we believe it opens multiple lines of work. To cite a few; smarter index optimizations, support of high dimensional space, stratified sampling and machine learning algorithm optimized to work on MIS. In Chapter 4, we presented only the ReadOptimization as optimization policy, but others are possible. For example, in our patent application, we also describe policies aiming to predict future queries or to optimize area containing popular items. Also, we are studying how to adapt popular machine learning algorithms such as the K-NN, Kmeans, and the DB-scan, to take advantage of the nature of the AOtree to reduce both the algorithm complexity and I/O requirements.

It is our belief Qbeast will influence the design of the future distributed databases and improve the scientific workflow in the HPC community. A future line of work is tighter integration with HPC, improving the compatibility with the MPI and OpenMP libraries, and supporting in-situ analysis by entrusting Qbeast to decide which data to persist or to analyze *in loco*.

We also believe that Qbeast can play an essential role outside HPC. Indeed, thanks to the support of the European Community we are investigating how to bring our research to the broader community of Business Intelligence. Indeed, our project proposal for Quake - QBeast Utility Analysis to marKet and Enterprise - has been accepted under the grant *Horizon 2020 Future and Emerging Technologies Innovation Launchpad*. Started in April 2018, project Quake is focusing on the study of possible commercial exploitations for Qbeast. Quake financed the creation of our patent, and it is supporting the development of a business prototype and a sound business plan endorsed by a detailed market study. As part of the prototype, we used our system to index social network fashion images using both neural autoencoders and multi-dimensional deep learning classification. In the next months, we will gather users feedback to improve the accuracy of our classification and dimensional reducing algorithms and we will tests which ones provide the best user experience.

Bibliography

[1] *BSC MareNostrum Technical Information*. (https://www.bsc.es/marenostrum/marenostrum/technical-information).

[2] *Hecuba*. (https://github.com/bsc-dd/hecuba/).

[3] Daniel J Abadi. "Consistency tradeoffs in modern distributed database system design: CAP is only part of the story". In: *Computer* February (2012), pp. 37–42.

[4] Sameer Agarwal et al. "{BlinkDB}: Queries with bounded errors and bounded response times on very large data". In: *EuroSys* (2013), pp. 29–42. DOI: 10.1145/2465351.2465355.

[5] Nitin(University of Wisconsin-Madison) Agrawal et al. "Design Tradeoffs for SSD Performance". In: *ATC* (2008), 57–70. ISSN: 13837621. DOI: 10.1016/j.sysarc.2014.07.003.

[6] James Ahrens et al. "An image-based approach to extreme scale in situ visualization and analysis". In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE Press. 2014, pp. 424–434.

[7] James Ahrens et al. "An Image-Based Approach to Extreme Scale in Situ Visualization and Analysis". In: *International Conference for High Performance Computing, Networking, Storage and Analysis, SC* 2015-Janua.January (2014), pp. 424–434. ISSN: 21674337. DOI: 10.1109/SC.2014.40.

[8] James Ahrens et al. "In situ mpas-ocean image-based visualization". In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, Visualization & Data Analytics Showcase*. 2014.

[9] Anders Andrae. *Total Consumer Power Consumption Forecast*. 2017. URL: https://www.researchgate.net/publication/320225452_Total_Consumer_Power_Consumption_Forecast.

[10] D. Antons and F. T. Piller. "Opening the Black Box of "Not Invented Here": Attitudes, Decision Biases, and Behavioral Consequences". In: *Academy of Management Perspectives* 29.2 (2015), pp. 193–217. ISSN: 1558-9080. DOI: 10.5465/amp.2013.0091. URL: http://amp.aom.org/cgi/doi/10.5465/amp.2013.0091.

[11] Lars Arge et al. “The Priority R-tree: A Practically Efficient and Worst-case Optimal R-tree”. In: *ACM Transactions on Algorithms* (2008). ISSN: 15496325. DOI: 10.1145/1328911.1328920.

[12] Michael Armbrust et al. “Spark SQL”. In: *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data - SIGMOD '15* (2015), pp. 1383–1394. ISSN: 07308078. DOI: 10.1145/2723372.2742797. URL: http://dl.acm.org/citation.cfm?doid=2723372.2742797.

[13] Antoni Artigues et al. “ParaView + Alya + D8tree: Integrating High Performance Computing and High Performance Data Analytics”. In: *Procedia Computer Science* 108 (2017), pp. 465–474. ISSN: 18770509. DOI: 10.1016/j.procs.2017.05.170.

[14] Antoni Artigues et al. “Scientific Big Data Visualization : a Coupled Tools Approach 1 . Background : The tools”. In: (2014), pp. 4–18. DOI: 10.14529/jsfi140301.

[15] Norbert Beckmann et al. *The R*-tree: an efficient and robust access method for points and rectangles*. 1990. DOI: 10.1145/93605.98741.

[16] Petra Berenbrink et al. “Balanced Allocations: The Heavily Loaded Case”. In: *SIAM Journal on Computing* 35.6 (2006), pp. 1350–1385. ISSN: 0097-5397. DOI: 10.1137/S009753970444435X. URL: http://dx.doi.org/10.1137/S009753970444435X.

[17] Hal Berenson et al. “A critique of ANSI SQL isolation levels”. In: *Proceedings of the 1995 ACM SIGMOD international conference on Management of data - SIGMOD '95*. Vol. 7. June 1995. New York, New York, USA: ACM Press, 1995, pp. 1–10. ISBN: 0897917316. DOI: 10.1145/223784.223785. URL: http://roderic.uv.es/bitstream/handle/10550/41689/100573.pdf?sequence=1&isAllowed=yhttp://portal.acm.org/citation.cfm?doid=223784.223785.

[18] John Biddiscombe and Jerome Soumagne. “Parallel computational steering and analysis for hpc applications using a paraview interface and the hdf5 dsm virtual file driver”. In: *Proceedings of the 11th ...* (2011). ISSN: 18727654. DOI: 10.1016/j.ejogrb.2016.02.011. URL: http://dl.acm.org/citation.cfm?id=2386244.

[19] “Bigtable: A distributed storage system for structured data”. In: *7th Symposium on Operating Systems Design and Implementation (OSDI '06)*. USENIX Association, 2006.

[20] D Borthakur. “The hadoop distributed file system: Architecture and design”. In: *Hadoop Project Website* (2007), pp. 1–14. DOI: 10.1109/MSST.2010.5496972. URL: http://cloudcomputing.googlecode.com/svn/trunk/??/Hadoop_0.18_doc/hdfs_design.pdf.

[21] Eric Brewer. “Spanner, TrueTime & The CAP Theorem”. In: (2017), pp. 1–7.

[22] Michael J. Cahill, Uwe Röhm, and Alan D. Fekete. "Serializable isolation for snapshot databases". In: *ACM Transactions on Database Systems* 34.4 (2009), pp. 1–42. ISSN: 03625915. DOI: 10.1145/1620585.1620587. URL: http://portal.acm.org/citation.cfm?doid=1620585.1620587.

[23] *C/C++ Driver for Apache Cassandra*. https://goo.gl/Q5pq5sr.

[24] Feng Chen et al. "Understanding intrinsic characteristics and system implications of flash memory based solid state drives". In: *ACM SIGMETRICS Performance Evaluation Review* 37.1 (2009), pp. 181–192. ISSN: 0163-5999. DOI: 10.1145/1555349.1555371. URL: http://portal.acm.org/citation.cfm?id=1555349.1555371.

[25] Saar Cohen and Yossi Matias. "Spectral bloom filters". In: *Proceedings of the 2003 ACM SIGMOD international conference on on Management of data - SIGMOD '03*. New York, New York, USA: ACM Press, 2003, p. 241. ISBN: 158113634X. DOI: 10.1145/872757.872787. URL: http://portal.acm.org/citation.cfm?doid=872757.872787.

[26] Jeremy Condit et al. "Better I/O through byte-addressable, persistent memory". In: *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles - SOSP '09*. Vol. 168. 2. New York, New York, USA: ACM Press, 2009, p. 133. ISBN: 9781605587523. DOI: 10.1145/1629575.1629589. URL: http://portal.acm.org/citation.cfm?doid=1629575.1629589.

[27] Gao Cong, Christian S Jensen, and Dingming Wu. "Efficient retrieval of the top-k most relevant spatial web objects". In: *Proceedings of the VLDB Endowment* (2009). DOI: 10.14778/1687627.1687666.

[28] James C Corbett et al. "Spanner: Google's globally distributed database". In: *Google's Globally Distributed Database* 31.3 (2013), pp. 1–22. ISSN: 0734-2071. DOI: 10.1145/2491245. URL: http://dl.acm.org/citation.cfm?id=2491245%5Cnpapers3://publication/doi/10.1145/2491245.

[29] Raul de la Cruz, Hadrien Calmet, and Guillaume Houzeaux. "Implementing a XDMF / HDF5 Parallel File System in Alya". In: *Whitepaper of PRACE-1IP project* (2011), pp. 1–8. URL: http://www.prace-ri.eu/IMG/pdf/Implementing_a_XDMF_HDF5_Parallel_File_System_in_Alya-2.pdf.

[30] Cesare Cugnasco et al. "Aeneas: A Tool to Enable Applications to Effectively Use Non-relational Databases". In: *Procedia Computer Science* 18 (2013), pp. 2561–2564. ISSN: 18770509. DOI: 10.1016/j.procs.2013.05.441. URL: http://linkinghub.elsevier.com/retrieve/pii/S187705091300584X.

[31] Cesare Cugnasco et al. "D8-tree: a de-normalized approach for multidimensional data analysis on key-value databases". In: *Proceedings of the 17th International Conference on Distributed Computing and Networking - ICDCN '16*

(2016), pp. 1–10. DOI: 10.1145/2833312.2833314. URL: http://dl.acm.org/citation.cfm?doid=2833312.2833314.

[32] Cesare Cugnasco et al. "Exploiting Key-Value Data Stores Scalability for HPC". In: *Proceedings of the International Conference on Parallel Processing Workshops*. IEEE, Aug. 2017, pp. 85–94. ISBN: 9781538610442. DOI: 10.1109/ICPPW.2017.25. URL: http://ieeexplore.ieee.org/document/8026073/.

[33] Alfredo Cuzzocrea, Il-Yeol Song, and Karen C Davis. "Analytics over large-scale multidimensional data: the big data revolution!" In: ... *14th international workshop on Data* ... (2011), pp. 101–104. DOI: 10.1145/2064676.2064695. URL: http://dl.acm.org/citation.cfm?id=2064676.2064695%5Cnpapers3://publication/doi/10.1145/2064676.2064695.

[34] Giuseppe DeCandia et al. "Dynamo: Amazon's Highly Available Key-value Store". In: *Proceedings of the Symposium on Operating Systems Principles* (2007), pp. 205–220. ISSN: 01635980. DOI: 10.1145/1323293.1294281. URL: http://dl.acm.org/citation.cfm?id=1323293.1294281.

[35] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. "Consensus in the presence of partial synchrony". In: *Journal of the ACM* 35.2 (Apr. 1988), pp. 288–323. ISSN: 00045411. DOI: 10.1145/42282.42283. URL: http://portal.acm.org/citation.cfm?doid=42282.42283.

[36] E. F. E F. Codd. "Derivability, redundancy and consistency of relations stored in large data banks". In: *ACM SIGMOD Record* 38.1 (June 1969), p. 17. ISSN: 01635808. DOI: 10.1145/1558334.1558336. URL: http://portal.acm.org/citation.cfm?doid=1558334.1558336.

[37] Ahmed Eldawy and Mohamed F. Mokbel. "The era of Big Spatial Data". In: *2016 IEEE 32nd International Conference on Data Engineering, ICDE 2016* 10.12 (2016), pp. 1424–1427. ISSN: 15516245. DOI: 10.1109/ICDE.2016.7498361. URL: http://dl.acm.org/citation.cfm?doid=3137765.3137828.

[38] Nathan Fabian et al. "The ParaView coprocessing library: A scalable, general purpose in situ visualization library". In: *1st IEEE Symposium on Large-Scale Data Analysis and Visualization 2011, LDAV 2011 - Proceedings* (2011), pp. 89–96. DOI: 10.1109/LDAV.2011.6092322.

[39] Ronald Fagin, Amnon Lotem, and Moni Naor. "Optimal aggregation algorithms for middleware". In: *Journal of Computer and System Sciences* (2003). DOI: 10.1016/S0022-0000(03)00026-6.

[40] *Fio, flexible I/O tester*. (https://linux.die.net/man/1/fio).

[41] Mike Folk, Gerd Heber, and Quincey Koziol. "An overview of the HDF5 technology suite and its applications". In: *Proceedings of the EDBT/* ... March (2011), 36–47. DOI: 10.1145/1966895.1966900. URL: http://doi.acm.org/10.1145/1966895.1966900%5Cnhttp://www.rasdaman.com/

ArrayDatabases-Workshop/Slides/5-hdf5_oo.pdf%5Cnhttp://dl.acm.org/citation.cfm?id=1966900.

[42] Sarah F. Frisken and Ronald N. Perry. "Simple and Efficient Traversal Methods for Quadtrees and Octrees". In: *Journal of Graphics Tools* 7 (2002). ISSN: 1086-7651. DOI: 10.1080/10867651.2002.10487560.

[43] *Ganglia Monitoring System*. http://ganglia.info/. Accessed: 2016-07-29.

[44] Marta Garcia, Jesus Labarta, and Julita Corbalan. "Hints to improve automatic load balancing with LeWI for hybrid applications". In: *Journal of Parallel and Distributed Computing* 74.9 (2014), pp. 2781–2794. ISSN: 07437315. DOI: 10.1016/j.jpdc.2014.05.004.

[45] Gartner. "Real-time Insights and Decision Making using Hybrid Streaming , In-Memory Computing Analytics and Transaction Processing". In: (2016), pp. 1–9.

[46] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. "The Google file system". In: *ACM SIGOPS Operating Systems Review* 37.5 (2003), p. 29. ISSN: 01635980. DOI: 10.1145/1165389.945450.

[47] Seth Gilbert and Nancy Lynch. "Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services". In: *ACM SIGACT News* 33.2 (June 2002), p. 51. ISSN: 01635700. DOI: 10.1145/564585.564601. URL: http://portal.acm.org/citation.cfm?doid=564585.564601.

[48] Inigo Goiri et al. "ApproxHadoop: Bringing Approximations to MapReduce Frameworks". In: *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems* 1 (2015), pp. 383–397. DOI: 10.1145/2694344.2694351. URL: http://doi.acm.org/10.1145/2694344.2694351.

[49] Jim Gray and Leslie Lamport. "Consensus on Transaction Commit". In: 1.July (2004). ISSN: 03625915. DOI: 10.1145/1132863.1132867. URL: http://arxiv.org/abs/cs/0408036.

[50] A Guttman. *R-trees: a dynamic index structure for spatial searching*. ACM Press, 1984.

[51] Frank T. Hady et al. "Platform Storage Performance with 3D XPoint Technology". In: *Proceedings of the IEEE* 105.9 (2017), pp. 1822–1833. ISSN: 00189219. DOI: 10.1109/JPROC.2017.2731776.

[52] Dan Han and Eleni Stroulia. "HGrid: A Data Model for Large Geospatial Data Sets in HBase". In: *2013 IEEE Sixth International Conference on Cloud Computing* (). DOI: 10.1109/CLOUD.2013.78.

[53] R. Hernandez et al. "Experiences of Using Cassandra for Molecular Dynamics Simulations". In: *Parallel, Distributed and Network-Based Processing (PDP), 2015 23rd Euromicro International Conference on* (2015), pp. 288–295. ISSN: 1066-6192. DOI: 10.1109/PDP.2015.43.

[54] Roger Hernandez et al. "Automatic query driven data modelling in cassandra". In: *Procedia Computer Science* 51 (2015), pp. 2822–2826. ISSN: 18770509. DOI: 10.1016/j.procs.2015.05.441.

[55] Adam Hospital et al. "BIGNASim: A NoSQL database structure and analysis portal for nucleic acids simulation data". In: *Nucleic Acids Research* 44.D1 (2016), pp. D272–D278. ISSN: 13624962. DOI: 10.1093/nar/gkv1301.

[56] Guillaume Houzeaux, Michel Aubry, and Mariano Vázquez. "Extension of fractional step techniques for incompressible flows: The preconditioned Orthomin(1) for the pressure Schur complement". In: (). ISSN: 0045-7930. DOI: DOI:10.1016/j.compfluid.2011.01.017. URL: http://www.sciencedirect.com/science/article/B6V26-520J96C-2/2/ab5d4e88c36e26eac7c15f5edfcbb159.

[57] Guillaume Houzeaux et al. "Parallel uniform mesh multiplication applied to a Navier-Stokes solver". In: *Computers and Fluids* 80.1 (2013), pp. 142–151. ISSN: 00457930. DOI: 10.1016/j.compfluid.2012.04.017. URL: http://dx.doi.org/10.1016/j.compfluid.2012.04.017.

[58] Mark Howison et al. "H5hut : A High-Performance I / O Library for Particle-based Simulations". In: (2010).

[59] Mark Howison et al. "Tuning HDF5 for Lustre file systems". In: *IASDS '10 Proceedings of the Workshop on Interfaces and Abstractions for Scientific Data Storage* 5 (2012). URL: http://escholarship.org/uc/item/46r9d86r.pdf.

[60] Xiao-yu Hu and Robert Haas. "The Fundamental Limit of Flash Random Write Performance". In: *Writing* (2010).

[61] *Hubble Essentials: Quick Facts*. (http://hubblesite.org/the_telescope/hubble_essentials/quick_facts.php).

[62] *IBM General Parallel File System*. https://goo.gl/yWmVXw.

[63] *Intel SSD DC S3520 Series*. (https://ark.intel.com/products/93012/Intel-SSD-DC-S3520-Series-240GB-2-5in-SATA-6Gb-s-3D1-MLC-).

[64] Jianwei Li et al. "Parallel netCDF: A High-Performance Scientific I/O Interface". In: *Supercomputing* (2003), pp. 1–11. DOI: 10.1109/SC.2003.10053.

[65] Ibrahim Kamel and Christos Faloutsos. "Hilbert R-tree: An Improved R-tree using Fractals". In: *International Conference on Very Large Databases (VLDB)* (1994).

[66] Jonathan G Koomey and D Ph. "Growing in Data Center Electircity Use 2005 to 2010". In: (2011).

[67] Marcel Kornacker, C Mohan, and Joseph M Hellerstein. "Concurrency and recovery in generalized search trees". In: *SIGMOD '97 Proceedings of the 1997 ACM SIGMOD international conference on Management of data* 26.2 (1997), pp. 62–72. ISSN: 01635808. DOI: 10.1145/253262.253272. URL: http://portal.acm.org/citation.cfm?doid=253262.253272.

[68] Ravi Kanth V. Kothuri, Siva Ravada, and Daniel Abugov. "Quadtree and R-tree indexes in oracle spatial: a comparison using GIS data". In: *the 2002 ACM SIGMOD international conference on Management of data* (2002). DOI: 10.1145/564691.564755.

[69] *Kryo: Java serialization and cloning*. https://goo.gl/R8jacQ.

[70] Laksham Avinash and Prashant Malik. "Cassandra: a decentralized structured storage system". In: *ACM SIGOPS Operating Systems Review* (2010), pp. 1–6. ISSN: 01635980. DOI: 10.1145/1773912.1773922. URL: http://dl.acm.org/citation.cfm?id=1773922.

[71] Avinash Lakshman and Prashant Malik. "Cassandra". In: *ACM SIGOPS Operating Systems Review* 44.2 (Apr. 2010), p. 35. ISSN: 01635980. DOI: 10.1145/1773912.1773922. URL: http://portal.acm.org/citation.cfm?doid=1773912.1773922.

[72] Leslie Lamport. "Time, clocks, and the ordering of events in a distributed system". In: *Communications of the ACM* 21.7 (July 1978), pp. 558–565. ISSN: 00010782. DOI: 10.1145/359545.359563. URL: http://portal.acm.org/citation.cfm?doid=359545.359563.

[73] Leslie Lamport and others. "Paxos made simple". In: *ACM Sigact News* 32.4 (2001), pp. 18–25.

[74] John MacCormick et al. "Kinesis". In: *ACM Transactions on Storage* (2009). DOI: 10.1145/1480439.1480440.

[75] Nathan Marz and James Warren. *Big Data: Principles and best practices of scalable real-time data systems*. New York; Manning Publications Co., 2015. ISBN: 9781617290343.

[76] Kirk McKusick and Sean Quinlan. "GFS: evolution on fast-forward". In: *Communications of the ACM* 53.3 (2010), pp. 42–49. ISSN: 00010782. DOI: http://doi.acm.org.ezp-prod1.hul.harvard.edu/10.1145/1594204.1594206. URL: http://portal.acm.org/citation.cfm?id=1666420.1666439.

[77] *MemSQL*. http://www.memsql.com.

[78] C. Mohan et al. "ARIES: a transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging". In: *ACM Transactions on Database Systems* 17.1 (Mar. 1992), pp. 94–162. ISSN: 03625915. DOI: 10.1145/128765.128770. URL: http://portal.acm.org/citation.cfm?doid=128765.128770.

[79] Thomas Neumann, Tobias Mühlbauer, and Alfons Kemper. "Fast Serializable Multi-Version Concurrency Control for Main-Memory Database Systems". In: *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data - SIGMOD '15* (2015), pp. 677–689. ISSN: 07308078. DOI: 10.1145/2723372.2749436. URL: http://dl.acm.org/citation.cfm?doid=2723372.2749436.

[80] Salman Niazi et al. "HopsFS : Scaling Hierarchical File System Metadata Using NewSQL Databases This paper is included in the Proceedings of the 15th USENIX Conference on". In: *Fast* (2017).

[81] Shoji Nishimura et al. "MD -HBase: Design and implementation of an elastic data infrastructure for cloud-scale location services". In: *Distributed and Parallel Databases* 31 (2013). DOI: 10.1007/s10619-012-7109-z.

[82] Prashant Pandey et al. "A General-Purpose Counting Filter". In: *Proceedings of the 2017 ACM International Conference on Management of Data - SIGMOD '17* (2017), pp. 775–787. ISSN: 16130073. DOI: 10.1145/3035918.3035963. URL: http://dl.acm.org/citation.cfm?doid=3035918.3035963.

[83] Andrew Pavlo and Matthew Aslett. "What's Really New with NewSQL?" In: *ACM SIGMOD Record* 45.2 (2016), pp. 45–55. ISSN: 01635808. DOI: 10.1145/3003665.3003674. URL: http://dl.acm.org/citation.cfm?doid=3003665.3003674.

[84] M. Pease, R. Shostak, and L. Lamport. "Reaching Agreement in the Presence of Faults". In: *Journal of the ACM* 27.2 (Apr. 1980), pp. 228–234. ISSN: 00045411. DOI: 10.1145/322186.322188. URL: http://portal.acm.org/citation.cfm?doid=322186.322188.

[85] *perfect k-ary tree*. https://goo.gl/sV7K45.

[86] Torben Kling Petersen. "Inside The Lustre File System". In: *SEAGATE Technology paper.* (2015).

[87] Henning Piezunka and Linus Dahlander. "Distant search, narrow attention: How crowding alters organizations' filtering of suggestions in crowdsourcing". In: *Academy of Management Journal* 58.3 (2015), pp. 856–880. ISSN: 00014273. DOI: 10.5465/amj.2012.0458. URL: http://pubs.acs.org/doi/abs/10.1021/ie010187i.

[88] Jean-pierre Prost et al. "MPI-IO/GPFS, an Optimized Implementation of MPI-IO on Top of GPFS". In: *SC '01: Proceedings of the 2001 ACM/IEEE Conference on Supercomputing*. November 2001. 2001, pp. 0–14. ISBN: 1-58113-293-X. DOI: 10.1109/SC.2001.10047.

[89] Aw Richa. "The power of two random choices: A survey of techniques and results". In: *Combinatorial ...* 1.1 (2001), pp. 1–60. URL: http://books.google.com/books?hl=en&lr=&id=ZGgnFGfHGroC&oi=fnd&pg=PA255&dq=The+Power+of+Two+Random+Choices+:+A+Survey+of+Techniques+and+Results&ots=wxTNXanuWg&sig=YeKtV6TSR966W8cVEgkdVOEgZtg\nhttp://books.google.com/books?hl=en&lr=&id=ZGgnFGfHGroC&oi=fnd&.

[90] Marzia Rivi et al. "In-situ visualization: State-of-the-art and some use cases". In: *PRACE White Paper* (2012), pp. 1–18.

[91] Pamela Rogerson-revell. "BeeGFSintro". In: 47.4 (2010), pp. 162–166. DOI: 10.1177/0021943610377298.

[92] A Rosenberg. "Improving query performance in data warehouses". In: *Business Intelligence Journal* 11.1 (2006), p. 7.

[93] Anish Das Sarma et al. "Consistent thinning of large geographical data for map visualization". In: *ACM Transactions on Database Systems* (2013). DOI: 10.1145/2539032.2539034.

[94] Frank Schmuck and Roger Haskin. "GPFS: A shared-disk file system for large computing clusters". In: *Proceedings of the First USENIX Conference on File and Storage Technologies* January (2002), 231–244. ISSN: 02692813. DOI: 10.1111/j.1365-2036.2004.02077.x. URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.19.7147&rep=rep1&type=pdf.

[95] *ScyllaDB achieves Cassandra feature parity, adds HTAP, cloud, and Kubernetes support*. https://goo.gl/3QctwZ.

[96] Timos K Sellis, Nick Roussopoulos, and Christos Faloutsos. "The R+-tree: A Dynamic Index for Multi-dimensional Objects". In: *International Conference on Very Large Databases (VLDB)*. 1987. ISBN: 0-934613-46-X.

[97] Konstantin Shvachko et al. "The Hadoop Distributed File System". In: *2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*. Vol. 454. IEEE, May 2010, pp. 1–10. ISBN: 978-1-4244-7152-2. DOI: 10.1109/MSST.2010.5496972. URL: http://www.aanda.org/articles/aa/abs/2006/30/aa5130-06/aa5130-06.htmlhttp://ieeexplore.ieee.org/document/5496972/.

[98] Konstantin V Shvachko. "HDFS Scalability: The limits to growth". In: *; login:: the magazine of USENIX & SAGE* (2010).

[99] Vishal Sikka et al. “Efficient transaction processing in SAP HANA database”. In: *Proceedings of the 2012 international conference on Management of Data - SIGMOD '12*. New York, New York, USA: ACM Press, 2012, p. 731. ISBN: 9781450312479. DOI: 10.1145/2213836.2213946. URL: http://dl.acm.org/citation.cfm?doid=2213836.2213946.

[100] Bogdan Simion et al. “The Price of Generality in Spatial Indexing”. In: *BigSpatial '13 Proceedings of the 2nd ACM SIGSPATIAL International Workshop on Analytics for Big Geospatial Data* (2013), pp. 8–12. DOI: 10.1145/2534921.2534923.

[101] Václav Snášel et al. “Geometrical and topological approaches to Big Data”. In: *Future Generation Computer Systems* 67 (2017), pp. 286–296. ISSN: 0167739X. DOI: 10.1016/j.future.2016.06.005. URL: http://dx.doi.org/10.1016/j.future.2016.06.005.

[102] Michael Stonebraker and Ariel Weisberg. “The VoltDB Main Memory DBMS”. In: *IEEE Data Eng. Bull.* (2013), pp. 21–27. ISSN: 00237205. DOI: 10.3141/2046-07. URL: http://sites.computer.org/debull/a13june/voltdb1.pdf.

[103] Michael Stonebraker et al. “The End of an Architectural Era: (It's Time for a Complete Rewrite)”. In: *Proceedings of the 33rd International Conference on Very Large Data Bases*. 2007. ISBN: 9781595936493. URL: http://dl.acm.org/citation.cfm?id=1325851.1325981.

[104] Amy Tai et al. “Replex : A Scalable , Highly Available Multi-Index Data Store”. In: *Atc* (2016), pp. 337–350.

[105] Farhan Tauheed et al. “Accelerating Range Queries for Brain Simulations”. In: *2012 IEEE 28th International Conference on Data Engineering* (). DOI: 10.1109/ICDE.2012.56.

[106] Enric Tejedor et al. “PyCOMPSs: Parallel computational workflows in Python”. In: *International Journal of High Performance Computing Applications* 31.1 (2017), pp. 66–82. ISSN: 17412846. DOI: 10.1177/1094342015594678.

[107] Martin Theobald, Gerhard Weikum, and Ralf Schenkel. “Top-k query evaluation with probabilistic guarantees”. In: *on Very large data bases-Volum* (2004). DOI: doi:10.1016/B978-012088469-8.50058-9.

[108] Ankit Toshniwal et al. “Storm@twitter”. In: *Proceedings of the 2014 ACM SIGMOD international conference on Management of data - SIGMOD '14*. New York, New York, USA: ACM Press, 2014, pp. 147–156. ISBN: 9781450323765. DOI: 10.1145/2588555.2595641. URL: http://dl.acm.org/citation.cfm?doid=2588555.2595641.

[109] Mariano Vazquez et al. “Alya: Towards Exascale for Engineering Simulation Codes”. In: (). URL: http://arxiv.org/abs/1404.4881.

[110] Peng Wang et al. "An efficient design and implementation of LSM-tree based key-value store on open-channel SSD". In: *Proceedings of the Ninth European Conference on Computer Systems - EuroSys '14* (2014), pp. 1–14. ISSN: 21601968. DOI: 10.1145/2592798.2592804. URL: http://dl.acm.org/citation.cfm?doid=2592798.2592804.

[111] Ling-Yin Wei et al. "Indexing spatial data in cloud data managements". In: *Pervasive and Mobile Computing* (2013). DOI: 10.1016/j.pmcj.2013.07.001.

[112] Sage A Weil et al. "Ceph: A Scalable, High-Performance Distributed File System". In: *OSDI '06 Proceedings of the 7th symposium on Operating systems design and implementation*. 2006, pp. 307–320. ISBN: 1-931971-47-1. URL: https://dl.acm.org/citation.cfm?id=1298485.

[113] Sage A. Weil et al. "Dynamic Metadata Management for Petabyte-Scale File Systems". In: *Proceedings of the ACM/IEEE SC 2004 Conference: Bridging Communities* 00.November (2004). DOI: 10.1109/SC.2004.22.

[114] Youjip Won et al. "Barrier Enabled IO Stack for Flash Storage". In: (2017). ISSN: 15522938. URL: http://arxiv.org/abs/1711.02258.

[115] *Z-order*. https://en.wikipedia.org/wiki/Z-order.

[116] Matei Zaharia et al. "Discretized streams". In: *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles - SOSP '13*. 1. New York, New York, USA: ACM Press, 2013, pp. 423–438. ISBN: 9781450323888. DOI: 10.1145/2517349.2522737. URL: http://dl.acm.org/citation.cfm?doid=2517349.2522737.

[117] Matei Zaharia et al. "Resilient Distributed Datasets: A Fault-Tolerant Abstraction for\rIn-Memory Cluster Computing\r\n\n". In: (). ISSN: 00221112. DOI: 10.1111/j.1095-8649.2005.00662.x. URL: https://www.usenix.org/system/files/conference/nsdi12/nsdi12-final138.pdf.

[118] Matei Zaharia et al. "Spark : Cluster Computing with Working Sets". In: *HotCloud'10 Proceedings of the 2nd USENIX conference on Hot topics in cloud computing* (2010), p. 10. ISSN: 03642348. DOI: 10.1007/s00256-009-0861-0.

[119] Xiangyu Zhang et al. "An efficient multi-dimensional index for cloud data management". In: *Proceeding of the first international workshop on Cloud data management - CloudDB 2009* (). DOI: 10.1145/1651263.1651267.

[120] Yinghua Zhou et al. "Hybrid Index Structures for Location-based Web Search *". In: (2005). DOI: 10.1145/1099554.1099584.