UNIVERSIDADE ESTADUAL DO CEARÁ

CENTRO DE CIÊNCIAS E TECNOLOGIA

PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

MESTRADO ACADÊMICO EM CIÊNCIA DA COMPUTAÇÃO

CAIO HOLANDA COSTA

SCALABILITY PATTERNS FOR DISTRIBUTED NOSQL DATABASES

FORTALEZA – CEARÁ

2017

CAIO HOLANDA COSTA

SCALABILITY PATTERNS FOR DISTRIBUTED NOSQL DATABASES

Dissertação apresentada ao Curso de Mestrado Acadêmico em Ciência da Computação do Programa de Pós-Graduação em Ciência da Computação do Centro de Ciências e Tecnologia da Universidade Estadual do Ceará, como requisito parcial à obtenção do título de mestre em Ciências da Computação. Área de Concentração: Ciências da Computação

Orientador: PhD. Paulo Henrique Mendes Maia

Co-Orientador: PhD. Nabor das Chagas Mendonça

FORTALEZA – CEARÁ

2017

Dados Internacionais de Catalogação na Publicação

Universidade Estadual do Ceará

Sistema de Bibliotecas

Costa, Caio Holanda Costa.
Scalability patterns for distributed nosql databases [recurso eletrônico] / Caio Holanda Costa Costa. - 2017.
1 CD-ROM: il.; 4 ¾ pol.

CD-ROM contendo o arquivo no formato PDF do trabalho acadêmico com 128 folhas, acondicionado em caixa de DVD Slim (19 x 14 cm x 7 mm).

Dissertação (mestrado acadêmico) - Universidade Estadual do Ceará, Centro de Ciências e Tecnologia, Mestrado Acadêmico em Ciência da Computação, Fortaleza, 2017.
Área de concentração: Ciências da Computação.
Orientação: Prof. Dr. Paulo Henrique Mendes Maia.
Coorientação: Prof. Dr. Nabor das Chagas Mendonça.

1. nosql. 2. banco de dados. 3. padrões. 4. escalabilidade. 5. benchmark. I. Título.

CAIO HOLANDA COSTA

SCALABILITY PATTERNS FOR DISTRIBUTED NOSQL DATABASES

Dissertação apresentada ao Curso de Mestrado Acadêmico em Ciência da Computação do Programa de Pós-Graduação em Ciência da Computação do Centro de Ciências e Tecnologia da Universidade Estadual do Ceará, como requisito parcial à obtenção do título de mestre em Ciências da Computação. Área de Concentração: Ciências da Computação

Aprovado em: 22 de Fevereiro de 2017

BANCA EXAMINADORA

PhD. Paulo Henrique Mendes Maia (Orientador)
Universidade Estadual do Ceará – UECE

PhD. Nabor das Chagas Mendonça (Co-Orientador)
Universidade de Fortaleza - UNIFOR

Dr. Marcial Porto Fernandez
Universidade Estadual do Ceará

Dr. Javam de Castro Machado
Universidade Federal do Ceará - UFC

À minha esposa, por me permitir realizar esse sonho. Foi seu imenso esforço de assumir sozinha todas as responsabilidades que me deu a oportunidade de cursar esse mestrado. Cada vitória é também sua, pois eu não teria conseguido dar um passo sequer sem seu apoio incondicional!

AGRADECIMENTOS

Agradeço aos meus pais, foram eles que, com muito sacrifício e renúncia, me trouxeram até aqui. Eles me apoiaram nas grandes decisões da minha vida, e compreenderam minha ausência durante o período de realização deste trabalho.

Agradeço a todos os professores e funcionários da UECE envolvidos no programa de Mestrado Acadêmico em Ciências da Computação. Sua dedicação mantem acessa a esperança de um Brasil melhor. Agradeço ao meu co-orientador, professor Nabor Mendonça, da Universidade de Fortaleza (UNIFOR). Suas observações, conselhos e ajuda foram indispensáveis.

E, finalmente, agradeço especialmente ao meu orientador, professor Paulo Henrique, da Universidade Estadual do Ceará (UECE). Sua paciência, incentivo, apoio, confiança e fé foram essenciais para que, apesar de todos os obstáculos, eu conseguisse produzir esse trabalho. Considero-o mais que um professor e orientador. Considero-o um amigo e exemplo.

"...it ain't about how hard you hit. It's about how hard you can get hit and keep moving forward. How much you can take and keep moving forward. That's how winning is done!"

(Rocky Balboa)

RESUMO

Para lidar com a crescente demanda de armazenamento e provimento de dados das atuais aplicações on-line, empresas como Amazon e Google desenvolveram bancos de dados não relacionais verdadeiramente distribuídos que se tornaram conhecidos como bancos de dados NoSQL. A fim de prover de maneira simples uma grande capacidade de escalabilidade horizontal, esses bancos de dados têm como modelo de dados o domínio da aplicação e abrem mão de alguns princípios de bancos de dados relacionais. No entanto, adotar um banco de dados NoSQL distribuído e comprometer-se com seu modelo de dados pode não fornecer o nível de escalabilidade necessário para alguns cenários. Nesses casos, é necessário implementar a abordagem de modelagem de dados correta para alavancar a escalabilidade do banco de dados. Portanto, este trabalho cataloga quatro padrões de modelagem de dados que visam melhorar a escalabilidade de aplicativos baseados em bancos de dados NoSQL distribuídos: UUID Key, Index Table, Enumerable Keys e Fan-out on Write. Os padrões propostos incluem testes baseados em cargas de trabalho que comparam o desempenho dos padrões com abordagens mais intuitivas e menos escaláveis, a fim de proporcionar uma melhor compreensão das melhorias e desvantagens dos padrões. O código-fonte das cargas de trabalho e das ferramentas desenvolvidas para executar os testes dos padrões estão compartilhados em um repositório público de forma que estudantes e desenvolvedores possam personalizá-los e testá-los em seus próprios ambientes.

Keywords: NoSQL. banco de dados. padrões. escalabilidade. benchmark. cargas de trabalho. YCSB

ABSTRACT

In order to cope with the ever increasing storage service demand of current online applications, companies like Amazon and Google have developed truly distributed non-relational datastores that become known as NoSQL databases. Those databases rely on a domain driven data model and give up some principles of relational databases in order to provide an easy and great scaling-out capacity. However, adopting a distributed NoSQL database and committing to its data model may not provide the scalability level required for some scenarios. In those cases it is necessary to employ the correct data modeling approach in order to leverage database scalability. Therefore, this work catalogs four data modeling patterns that aim to improve the scalability of applications based on distributed NoSQL databases: UUID Key, Index Table, Enumerable Keys e Fan-out on Write. The proposed patterns include workload-based tests that compare the performance of the patterns against more intuitive and less scalable approaches in order to provide a better understanding of the improvements and drawbacks of the patterns. The source code of the workloads and tools developed to support the patterns tests are available in a public repository so students and developers can customize and test them in their own scenarios.

Keywords: NoSQL. database. pattern. scalability. benchmark. workload. YCSB

LIST OF ILLUSTRATIONS

LIST OF TABLES

LIST OF SOURCE CODES

LIST OF ABBREVIATIONS AND ACRONYMS

ACID	Atomicity, Consistency, Isolation and Durability
AP	Available and Partition-Tolerant
API	Application Programing Interface
AWS	Amazon Web Services
CAP	Consistency, Availability, Partition tolerance
CF	column-family
CP	Consistency and Partition-Tolerant
CQL	Cassandra Query Language
CSV	Comma Separated values
DBA	Database Administrator
DO	document-oriented
DSL	Domain Specific Language
EC2	Elastic Compute Cloud
FOR	Fan-out on Read
FOW	Fan-out on Write
GSI	Global Secondary Indexes
GUID	Globally Unique Identifier
HCI	Human-Computer Interaction
IK	Incrementing Key
IOPS	Input/Ouput per Second
JEE	Java Enterprise Edition
JSON	JavaScript Object Notation
KV	Key-value
LBA	List-Based Association
MVC	Model-View-Controller
NoSQL	Not Only SQL
OLTP	On-line Transaction Processing
OO	object-oriented
RFC	Request For Comments
RSS	Rich Site Summary
SPOF	Single Point of Failure

SQL	Structured Query Language
SSH	Secure Shell
UML	Unified Modelling Language
UUID	Universally Unique Identifier
XDCR	Cross Datacenter Replication
YCSB	Yahoo! Cloud Serving Benchmark

SUMMARY

1 INTRODUCTION

1.1 MOTIVATION

With the beginning of the 21st century, web applications requirements dramatically increased in scale. Applications like social networks, media streaming services, e-commerce and storage services have become part of people's daily lives. Coping with the data volume growth and the ever increasing number of concurrent users has required more computing resources. Consequently, database systems had to scale in order to handle the huge load without impairing services and applications performance.

Scaling out database systems has become the appropriate and definitive solution because scaling up was an expensive and practically limited approach. However, scaling out traditional relational database systems by employing the master-slave architecture is not a scalable solution since all the write requests must be submitted to the master, impairing write-scalability. In order to provide a truly scalable solution, it was necessary to partition data across multiple nodes of a shared-nothing database cluster (ELMASRI; NAVATHE, 2010). However, relational database systems were not designed to be executed on clusters (SADALAGE; FOWLER, 2012), since the relationships between tables and the ACID (Atomicity, Consistency, Isolation and Durability) principles do not favor data sharding (COSTA et al., 2015).

Driven by their own scalability requirements, companies like Amazon and Google developed non-relational databases based on a data model that do not rely on the principles that make difficult to shard relational databases. Not Only SQL (NoSQL) databases, as they become known, do not persist business domain entities as multiple related records (tuples), but rather store them as an independent single unit of information, or an aggregate, as named by Sadalage & Fowler (2012). Since aggregates are independent, they are easily distributed across the nodes of a database cluster, thus favoring data sharding.

Aggregate-oriented[1] NoSQL databases provide great horizontal scalability by strongly relying on the aggregate concept and giving up widespread features implemented by relational databases. The responsibility of giving semantic meaning for data and maintaining its integrity is shifted to the application. Therefore, developers and software architects must refrain their minds to correctly design the models of the application domain in order to adequate them for

[1] Graph databases are also included in the NoSQL umbrella term. However, as explained in Chapter 2, they are not addressed in this work.

that non-relational paradigm.

However, there are some scenarios in which committing to the aggregate data model or using the features provided by a determined NoSQL database product may not provide the expected or required level of scalability, or even impair the system scalability. For instance, in a popular web blog application, it is common for some posts to become hot topics, and hundreds, even thousands, of users try to submit comments at the same time. In that case, storing a post and its comments in a document-oriented NoSQL database as a single aggregate may cause scalability problems. In situations like that it is necessary to come up with an alternative approach for modeling and storing data in the database to provide greater scalability.

NoSQL database books usually focus on presenting the aggregate-based data model, and the advantages and restrictions imposed by that data model. However, they do not address modeling strategies that provide optimized scalability for determined scenarios, neither warn readers about scalability issues that may arise when using a NoSQL datastore in certain situations.

Software engineers and developers familiar to relational databases and learning the general concepts of NoSQL databases, or learning how to use a specific NoSQL database product, tend to merge some common techniques of relational databases based applications with the aggregate data model of NoSQL databases. It is also common for NoSQL databases novice users to be restricted to the features offered by the database that they are working with and do not apply non-trivial alternative approaches already widespread in the NoSQL community.

There is a lack of academic works that guide NoSQL database users, specially the novice ones, towards modeling approaches that leverage the scalability of NoSQL databases and avoid schemas that do not favor it. On the other hand, multiple experienced professionals from the NoSQL database community made available in multiple medias, well-proven modeling strategies that can leverage the scalability of applications that rely on aggregate-oriented NoSQL databases.

Therefore, in order to guide NoSQL databases newcomers to implement solutions that provide better scalability than the basic approaches described by NoSQL books and products documentations, this work catalogs four well-proven best practices and modeling approaches employed by the NoSQL community and describe them as patterns. The first three patterns describe fundamental best practices and modeling schemas that should be implemented by applications based on NoSQL databases that require a greater level o scalability, and the fourth pattern describes a modeling schema addressed to near real-time event streams applications. The

four patterns presented in this work are addressed to aggregate-oriented NoSQL databases and OLTP (On-line Transaction Processing) applications.

However, the patterns described in this work include an innovative practical aspect. Instead of just presenting fictitious scenarios to describe the solution introduced by the pattern, this work not only uses real scenarios, but also implemented workloads used to compare the performance of the proposed patterns against more intuitive approaches that possibly would be the applied solution by inexperienced developers. For each pattern presented in this work, two workloads are implemented: one for the naive approach and the other one for the scalable approach recommend by the pattern. Both workloads are executed against a NoSQL database deployed in a cloud distributed environment of the category for which the pattern is indicated for, and the resultant metrics are compared regarding their scalability. The workloads are a relevant technological contribution since the quantitative comparison between the approaches provides a better understanding of the scalability improvement provided by the pattern and its drawbacks, and makes easier for the reader to analyse the pattern advantages and disadvantages.

The Yahoo! Cloud Serving Benchmark (YCSB) framework has been used in order to implement the workloads and manage their executions during the patterns tests. However, the YCSB standard implementation does not provide some features required by the patterns tests. Therefore, in addition to the workloads, the features required by the patterns tests have been developed by extending the YCSB framework. An additional utility tool, called YCSBtoCSV, has been developed in order to facilitate and make less error-prone the extraction of the relevant metrics from the log files generated during the patterns tests.

The source code of the workloads developed for testing the patterns are available in a public repository[2] so students and developers interested in the subject may customize the workloads by providing different parameters, or modifying the source code, and execute them in their own environment, in order to evaluate the adoption of the pattern or for learning purposes. The source code of the YCSB extensions and the source code of the YCSBtoCSV tool are also available in the same public repository in which the workloads are stored.

1.2 OBJECTIVES

The main goal of this work is to catalog four patterns addressed to leverage the scalability of applications that store their data in aggregate-oriented NoSQL databases. The

[2] https://bitbucket.org/caiohc/

following list enumerates the specifics objectives of this work, which are the steps necessary to accomplish the main goal:

- Identifying the best-practices and well-proven modelling schemas employed by the NoSQL databases users community to be cataloged as scalable patterns.
- Implementing the features required by the patterns tests that are not provided by the standard implementation of the YCSB framework.
- Implementing the YCSB workloads required for each pattern test.
- Implementing the utility tool for collecting the resultant metrics from the patterns tests.
- Executing the patterns tests and collecting the results in order to describe the patterns.

1.3 OUTLINE

Chapter 2, named Theoretical Background, presents the subjects that this work relates to: aggregate-oriented NoSQL databases and patterns. That chapter explains the fundamental concepts that provide the great scalability capacity of NoSQL databases, the aggregate-oriented NoSQL databases common characteristics, and presents the types of aggregate-oriented NoSQL databases. As last topic, Chapter 2 explains what patterns in the context of software engineering are, presents the essentials elements of a pattern and its templates.

In Chapter 3, the main work that are related to this one are presented. The listed work included in that chapter have the same subject or apply similar methodology. Chapter 3 helps to situate this work in the research scenario related to databases patterns, modeling approaches and custom benchmarking tests.

Chapter 4 describes how the research was developed and gives special attention to its practical aspect. Since the innovative element of this works is the scalability tests included as a section of the patterns, Chapter 4 explains how the tests have been implemented, the tool used to execute the tests, the extensions developed for it, an auxiliar tool created to collect the metrics generated by the tests, and the tests environment.

Chapter 5 presents the four scalability patterns resultant from this research: the UUID Key pattern, the Index Table pattern, the Enumerable Keys pattern, and the Fan-out on Write pattern. The patterns are described using the proposed form that includes the charts that compares the pattern with its counterpart naive approach regarding their scalabilities.

Chapter 6 presents the conclusions of this work and suggests possible future work. Appendix A presents the result of the 95th percentile latency metric for each pattern.

2 THEORETICAL BACKGROUND

This chapter presents NoSQL databases and patterns because they are concepts necessary to comprehend the research described in this work. The first section explains aggregate-oriented databases and the second section gives a brief explanation of patters in the software engineering context.

2.1 NOSQL DATABASES

2.1.1 Motivation

Nowadays, online applications like social networks, ecommerce, and media streamings are part of people lives. Consequently, they serve thousands, even millions, of concurrent users and generate great volumes of data. Pioneers companies like Amazon, Facebook and Google have developed new datastores when they noticed that traditional relational databases alone were not enough too handle the unprecedented storage service demand of those applications.

In order to increase the capacity of a relational database, the simplest approach is to scale up, i.e., install more hard disks, more RAM memory, and more powerful CPUs. However, there is a practical limit for scaling up a database system since the hardware resources are limited by the current technology. Additionally, it is an expensive approach. On the other hand, increasing the capacity of a distributed system by adding more nodes is cheaper and virtually unlimited because more computational nodes can be added to the database system as the demand increases. The last approach is known as scaling out.

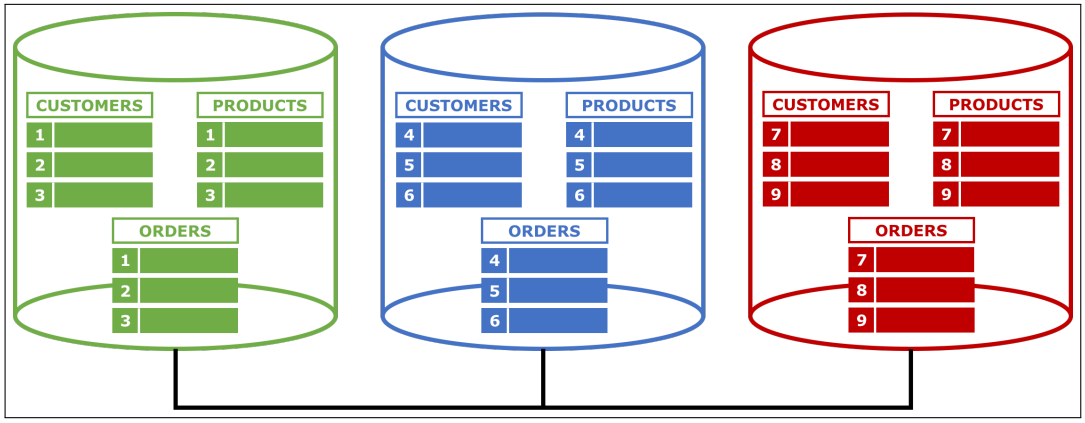
Therefore, in order to handle the ever increasing storage demand, the above-mentioned companies developed non-relational distributed databases that leverage the scaling out approach. Those databases and others built with the same purpose became known as Not Only SQL (NoSQL) databases.

2.1.2 Data Sharding

The master-slave architecture is the most used approach to scale out relation databases. However, it only provides read scalability since all the write requests must be submitted to the master node. In order to provide read and write scalability, data must be sharded across multiple nodes (COSTA et al., 2015). When a database is sharded, its tables are partitioned, as

equally as possible, across multiple nodes, as illustrated in Figure 1.

Figure 1 – Sharded database has its data distributed among the cluster nodes.



Source: Created by the author

Each node shown in Figure 1 stores a subset of each table's data (`customers`, `orders` and `products`). Since each node holds just a subset of data, read requests must be targeted to the right node. Similarly, each new record must follow a placement algorithm that determines in which node the record must be stored. Consequently, the read and write loads are balanced among the cluster nodes.

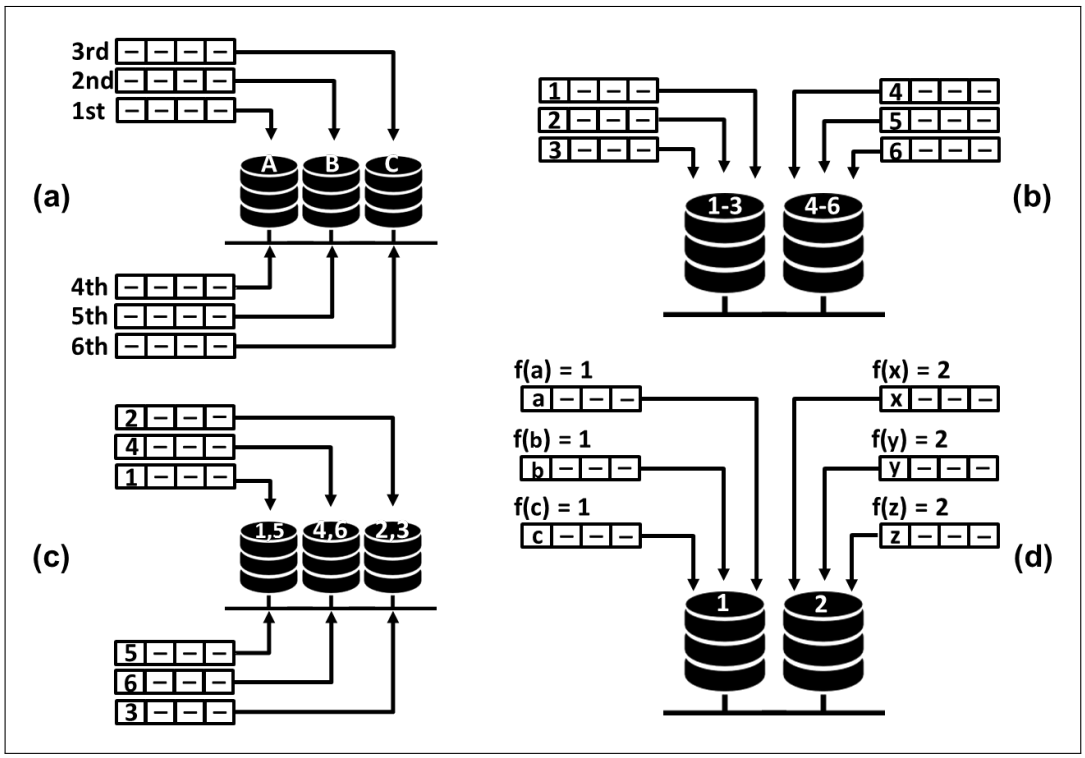
Each of the data subsets (a.k.a data chunks) stored in the cluster nodes are called data shards. In order to determine the shard where a record must be stored, one of its fields must be chosen as shard key. A shard key is the record field that serves as input parameter for the placement algorithm that determines the shard (consequently, the node) where the record should be stored. Therefore, all records of a table must have a shard key.

Figure 2 illustrates the four sharding approaches (DEWITT; GRAY, 1992):

- the round-robin approach (a) distributes the records of a table among the nodes in a round-robin fashion;
- the range approach (b) consists on assigning a shard key range for each shard;
- in the list approach (c), a set (not a range) of shard key values is assigned to each shard;
- in the hashing approach (d), the shard key of the record is used as input parameter of a hash function, and the result determines the target shard.

The hash sharding approach (Figure 2d) is the most employed one since its randomness tends to distribute better the load among the nodes of the cluster, resulting in better scalability. All NoSQL databases capable of scaling out implement data sharding. Therefore, all of them provide the key-value interface previously demonstrated:

Figure 2 – Sharding methods.



Source: (COSTA et al., 2015)

- in order to retrieve a record, a key (shard key) is supplied to the database;
- in order to store a record in the right shard (cluster node), the key (the shard key) and its value (the record itself) is supplied to the database.

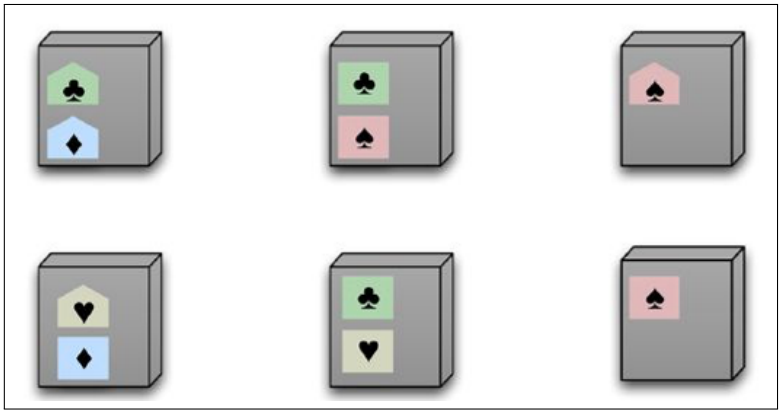
2.1.2.1 Replication

Hardware and network failures are common for large clusters composed of commodity servers. Therefore, in order to prevent data shards from becoming inaccessible, replication is fundamental to NoSQL databases.

Figure 3 shows a master-slave replication for a NoSQL cluster. The house shaped icons represent master shards, while the square shaped icons represent replica shards. As illustrated in the figure, a node can be the master for a shard and a slave for another shard at the same time. The left-most node in the bottom line is the master node for the heart shard and the slave node for the diamond shard at the same time.

When master-slave replication is employed, if a node that stores a master shard fails, the database will not accept a write request for that shard until a failover operation completes, i.e., a replica shard is elected as the new master. Therefore, some NoSQL databases implement

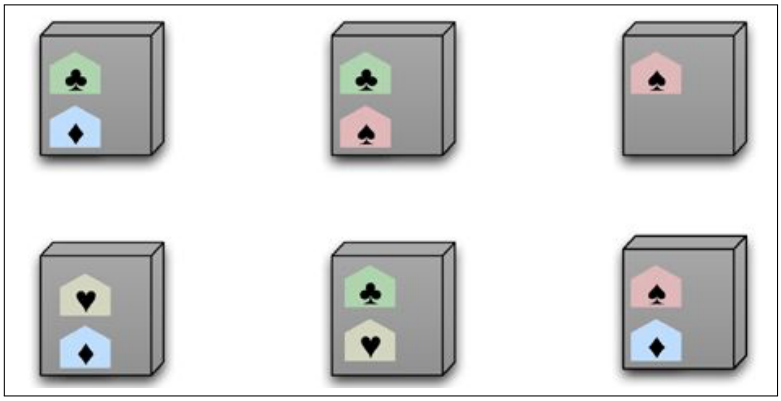
Figure 3 – NoSQL database cluster implemented with master-slave replication.



Source: (SADALAGE; FOWLER, 2012)

peer-to-peer replication. Figure 4 represents a NoSQL cluster in which all nodes are masters.

Figure 4 – NoSQL database cluster implemented with peer-to-peer replication.



Source: (SADALAGE; FOWLER, 2012)

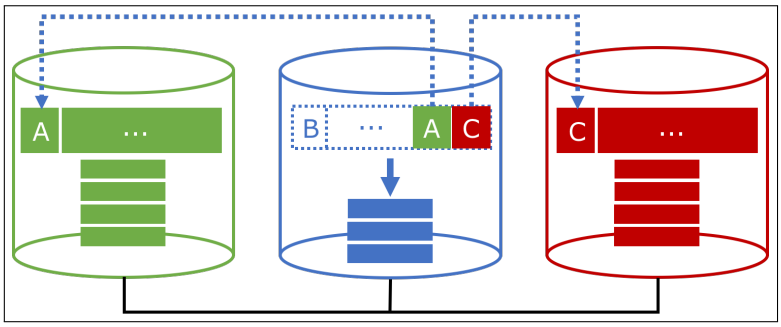
2.1.3 Characteristics of NoSQL Databases

In order to provide read and write scalability by implementing automatic data sharding and fast key-value operations, NoSQL databases present a set of common characteristics. Firstly, they are not aware of relationships that may exist between the stored records. For relational databases, a referential integrity constraint imposes that a record in one table that refers to another table must refer to an existing record in that table (ELMASRI; NAVATHE, 2010). In a sharded database cluster, related records may be stored in different nodes. Consequently, a referential integrity validation would require network communication between nodes.

For instance, Figure 5 depicts a hypothetical sharded database cluster that implements referential integrity. In order to store a record that has two foreign keys that refer to records

in different tables, the database must query two other nodes to check if the referenced records actually exist. That additional step makes data sharding complex and expensive. Therefore, NoSQL databases do not implement referential integrity in order to facilitate data sharding across the cluster. In a NoSQL database, the record in Figure 5 would be stored even if the referenced records did not exist.

Figure 5 – Hypothetical referential integrity validation in a sharded database cluster.



Source: Created by the author

NoSQL databases do not implement the transaction concept as relational databases do. As known in the relational databases domain, a transaction handles multiple retrievals and updates as an atomic unit of work against the database (ELMASRI; NAVATHE, 2010). In a sharded database, data is split across multiple nodes. Consequently, a set of related operations (retrievals and updates) are submitted to different nodes, making complex and expensive to treat those operations as a single unit of work. However, in NoSQL databases, every operation against a single record is atomic.

NoSQL databases do not implement join operations like relational databases do, since it also implies in scattering requests across the cluster. As previously mentioned, related records may be stored in different nodes and, this way, a join operation would trigger requests to multiple nodes in order to gather the requested records. In order to retrieve related records that reside in different tables, the application must submit multiple requests to the database. Couchbase implements an SQL-like language called N1QL that provides a JOIN operator. However, that operator is just a convenience since the node that accepts the request submits multiple requests to the cluster.

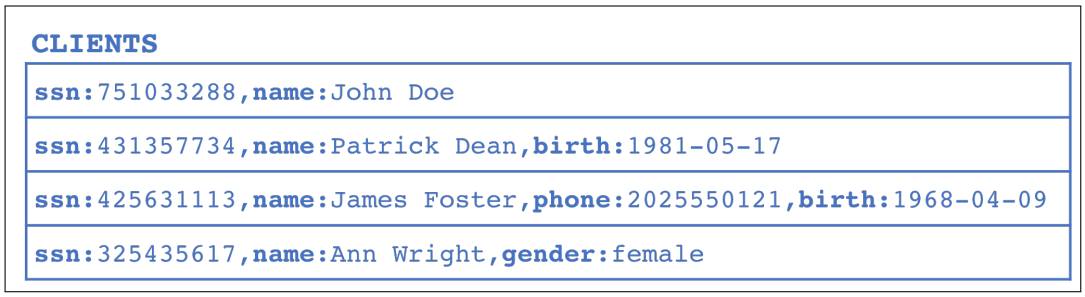
Although every operation executed in a master shard is replicated to its replicas, sometimes a record retrieved from a replica shard is not the most recent copy. Update or insert requests take a while to be executed by all the replicas, but eventually, they will be executed and

confirmed. Therefore, NoSQL databases are eventually consistent.

Most[1] NoSQL databases do not make any restrictions to the structure of the records stored in a table, i.e., they are schemaless. That means that the records of a table do not need to have the same set of fields. Figure 6 shows a schemaless collection that holds documents with different attributes. Those are artifacts of document-oriented databases and, if compared with relational databases, collections, documents and attributes can be understood as tables, records and fields, respectively

The four documents in the clients collection have different structures. The first and last documents do not have the `birth` attribute, and only the last document has the `gender` attribute. That feature is appropriate for storing unstructured data, and is one of the reasons why a NoSQL database may be chosen for a project. Data semantics and types are managed by the application. Therefore, there is an implicit schema imposed by the application.

Figure 6 – Collection of a schemaless document-oriented database.



Source: Created by the author

2.1.4 The CAP Theorem

The CAP theorem (BREWER, 2000) states that a stateful distributed system can guarantee only two of the three following properties at the same time: Consistency (C), Availability (A) and Partition-Tolerance (P). In the CAP theorem context, availability means that if a node can be reached, then it can accept write and read requests. Although NoSQL databases are partition-tolerant, there is a trade-off between consistency and availability. Considering a cluster composed by two nodes that cannot communicate due to a network partition, in order to keep both nodes accepting write requests, consistency must be impaired, since the cluster nodes cannot synchronize. On the other hand, in order to keep data consistency, there can be no writes

[1] Not all NoSQL databases are schemaless. For instance, Cassandra is a very popular NoSQL database, however, its proprietary CQL interface is not schemaless.

in both sites.

Considering the CAP theorem, NoSQL databases are classified as:

- mostly Consistency and Partition-Tolerant (CP), which trades-off availability for consistency;
- and mostly Available and Partition-Tolerant (AP), which trades-off consistency for availability.

The word *mostly* is used because that choice as not binary as it was at the time the CAP theorem was first introduced (BREWER, 2012). NoSQL databases provide settings and features that allow to tune the availability-consistency trade-off.

2.1.5 Types of NoSQL Databases

There are four types of NoSQL databases: key-value, document-oriented, column-family, and graph databases. However, while the first three have been created to provide great scalability by providing data sharding across a cluster, the last one is not distributed and was created to implement complex relationships between entities, like very extended friendship and likes graphs of social networks. Since the purpose of graph databases is not related to horizontal scalability, they are not addressed in this work. For the remainder of this text, the NoSQL expression does not include graph databases.

Key-value, document-oriented and column-family databases are known as aggregate-oriented databases (SADALAGE; FOWLER, 2012). Since those databases are unaware of any relationship between the stored records and do not implement join operations, it is common to aggregate information related to different business domain entities in a single record. That approach allows to reduce the number of requests submitted to the database since a record can aggregate all the necessary information for an application use case. Therefore, the aggregate expression designates the minimum unit of information that represents a business domain entity that can be stored in a NoSQL database. Often, an aggregate is compared to a record in a relation database.

2.1.5.1 Key-value databases

Key-value (KV) databases are the simplest and most scalable type of aggregate-oriented NoSQL databases. For a pure KV database, an aggregate is just a big blob of meaningless bits, i.e., the aggregate content is opaque to the database. Consequently, the only way to retrieve

an aggregate it's by a lookup based on its key.

KV databases are mostly used as in-memory cache layers. Examples of KV databases are: Memcached[2], MemcacheDB[3] and Redis[4]. The other categories of NoSQL databases are built upon key-value stores principles. However, they are more specialized (LAMLLARI, 2013).

2.1.5.2 Document-oriented databases

As the name suggests, in document-oriented (DO) databases, aggregates are known as documents. Differently from a KV database, a DO database is able to see the structure of the documents. Therefore, in addition to allowing aggregate lookups based on the shard key, a DO database allows client applications to submit queries based on the fields that compose the documents. It is also possible to retrieve parts of the document instead of the whole document, and create indexes based on the contents of the document. Additionally, the fields of a document may have types for which the database provides custom operations, e.g., incrementing an integer field.

Documents that represent the same business domain entity type are stored in the same collection. A collection is analogous to a table in a relational database. As a schemaless NoSQL database, documents of the same collection can have different structures. It is also possible to nest a document, or multiple documents, inside another document. Many DO databases use JavaScript Object Notation (JSON) to represent documents. Examples of DO databases are: Couchbase[5], MongoDB[6], and Elasticsearch[7].

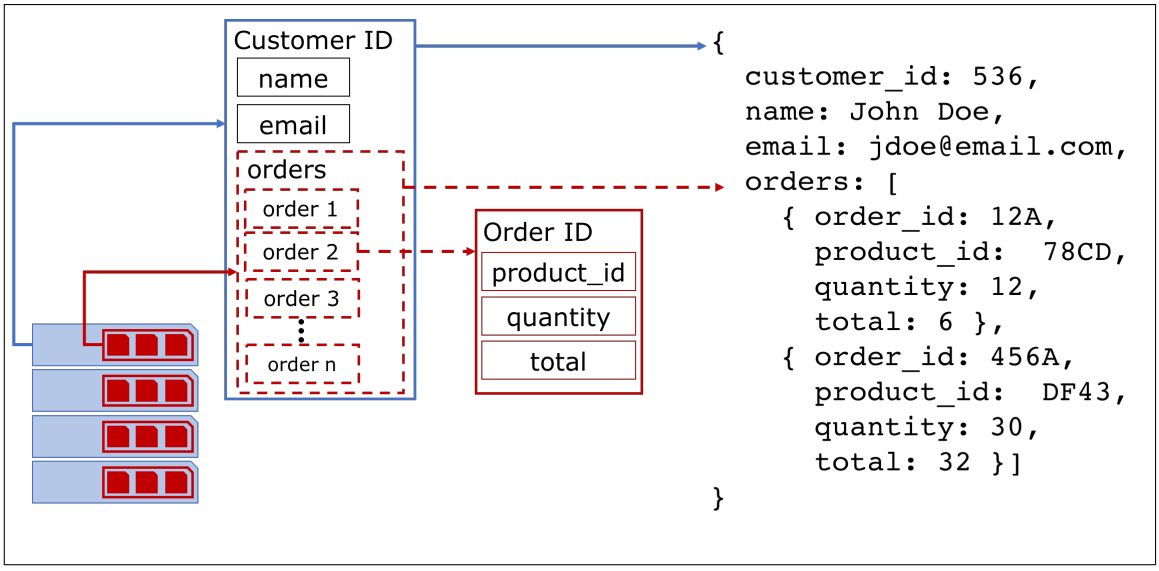
The following example helps to consolidate the aggregate concept and illustrates the use of DO databases. For an e-commerce application, there is an one-to-many association between customers and their orders. In a relational database, those entities would be stored in two tables and the database would be aware of the relationship between those table records. However, NoSQL databases do not implement relationships between aggregates. Therefore, in order to retrieve a customer document and its orders with a single query, the orders documents can be nested as a list of document in the customer document as illustrated by Figure 7.

Figure 7 shows the orders of a customer modeled as a list of documents attribute of

[2] https://memcached.org
[3] http://memcachedb.org
[4] https://redis.io
[5] https://www.couchbase.com/
[6] https://www.mongodb.com
[7] https://www.elastic.co/products/elasticsearch

the customer document. That is the meaning of an aggregate, i.e., to aggregate the necessary information in a single storage unit. Figure 7 also demonstrates the JSON representation of a customer document. The *id* attribute is the shard key and is used to retrieve the document with a key-value request. Additionally, DO databases are able to use the other attributes of a document as search keys, for instance, the name attribute can be used as search key to find a customer document.

Figure 7 – Customer and its orders stored as an aggregate in a DO database.



Source: Created by the author

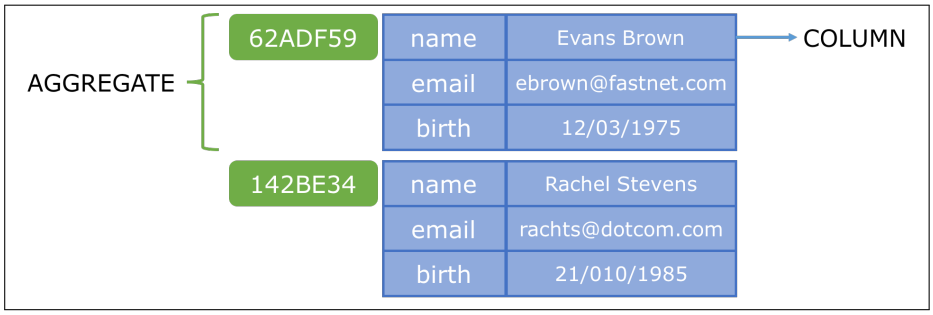
The same approach can be used to store the customers and their orders in a KV database. However, the only way to retrieve the customer aggregate is by a key-value request using the customer *id* as key. Once the application gets the aggregate, it can parse the JSON and use the information.

2.1.5.3 Column-family databases

In column-family (CF) databases, aggregates are called records or rows and, similarly to relational databases, they are composed by multiple columns. Records that represent the same business model entity type are grouped in a column family. A column family is analogous to a relational database table. However, very differently from a relational database and following the schemaless nature of NoSQL databases, records of the same column family can be composed by different column sets. That means that, the number of columns, their types, and their names can be different for records in the same column family.

The e-commerce example used in the previous subsection will be used here in order to explain better CF databases. Figure 8 shows two records of the customers column family. That type of column family is known as static column family because its rows present the same set of columns. For both records shown in Figure 8, the customer *id* is the shard key (the green cells). Therefore, it determines the record placement.

Figure 8 – Customers static column family.



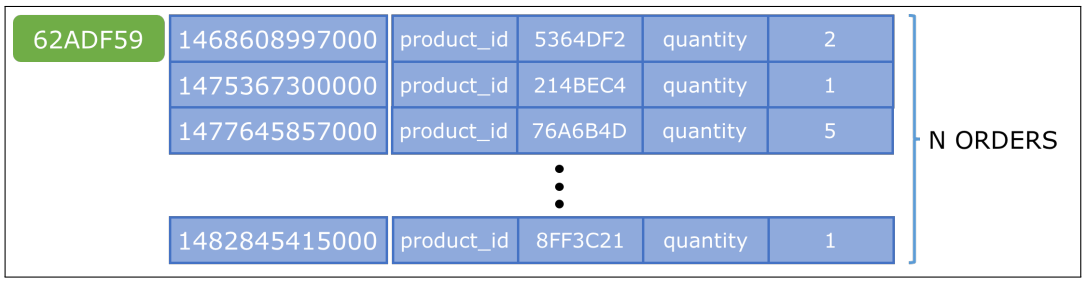
Source: Created by the author

CF databases implement a second type of column family known as dynamic column family, which is able to store records with different structures and each record can contain an arbitrary number of columns. That is the type of column family that must be used in order to store all the orders of a customer as a single aggregate. Since dynamic column families can be composed by an arbitrary number of columns, each order of a user can be stored as a set of columns of the same record. Therefore, dynamic column families are very efficient in representing one-to-many associations.

Figure 9 illustrates a dynamic column family record that stores the orders of a customer. The shard key for that record is the customer *id* and its orders are the columns of the record. The records of the customers static column family, illustrated in Figure 8, are the same for each record: name, email and birth. On the other hand, the records of the orders dynamic column family present different sets of columns since each record holds the orders of a customer as columns. Figure 9 shows that the orders ids are used as column names and the value of those columns are the orders data. For a customer *id* the columns that represent the order are sorted by the column id. In this example, an order *id* is an integer value that represents the order date.

Sadalage & Fowler (2012) and the documentation of Google Cloud Bigtable define dynamic columns as a two-level key-value map. In order to retrieve a single order record from the database, the customer *id* and the order *id* must be supplied. Those two parameters correspond to the shard key of the customer orders record and the name of the column that holds the order

Figure 9 – Orders dynamic column family.



Source: Created by the author

information. In order to retrieve all the order records of a customer, only the customer *id* is supplied, which in the case is the shard key.

Making an analogy with relational databases, the orders dynamic column family corresponds to a table whose records primary keys are composed by the customer *id* and the order id. Since customers and orders are involved in a one-to-many association, the customer *id* would be a foreign key in the orders table. Both fields are necessary to differentiate records, however, only the customer *id* is used as shard key. Consequently, the order records that belong to the same customer are stored in the same shard. The customer *id* acts like a clustering key and the order *id* acts like a sort key for the order records of a customer.

Examples of CF databases are: Cassandra[8], Google Cloud Bigtable[9], and DynamoDB[10].

2.1.5.3.1 Cassandra Query Language (CQL)

Cassandra is one of the databases used to test two patterns in this work, and CQL is the interface used to interact with the database. The analogy made between column-family databases and relational databases is implemented by CQL, which is a SQL-like syntax language implemented in Cassandra. Source code 1 illustrates how the orders table may be defined using CQL. The primary key of the orders table is composed by the customer *id* and the order date (represented as an integer number). The customer *id* is the shard key, consequently all the order records of a customer are stored in the same shard. Additionally, the records that share the same customer *id* are sorted by their creation dates.

Source code 1 – Orders dynamic column family defined by CQL.

[8] http://cassandra.apache.org
[9] https://cloud.google.com/bigtable
[10] https://aws.amazon.com/dynamodb

```
CREATE TABLE orders (
    customer_id uuid,
    order_date bigint,
    product_id uuid,
    quantity int,
    PRIMARY KEY (customer_id, order_date)
) WITH CLUSTERING ORDER BY (order_date ASC);
```

2.1.6 Secondary Indexes in NoSQL Databases

The following three major types of secondary indexes (ELMASRI; NAVATHE, 2010) (CONNOLY; BEGG, 2005) can be sparsely found in NoSQL databases:

- range indexes, implemented with binary trees, sorts records by one or more fields and supports equality and range queries;
- hash indexes, implemented with hashing algorithms, supports only equality queries and are appropriate for high cardinality keys;
- bitmap indexes, implemented with bit vectors, supports only equality queries and are appropriate for low cardinality keys and queries based on multiple indexed fields.

Secondary indexes are not homogeneously available in NoSQL databases as they are in relational databases. The three basic types of secondary indexes previously listed are provided by most relational databases. On the other hand, it is rare to find the three types in NoSQL databases. In fact, the most popular NoSQL databases in the market implement only two or one of them.

However, the most important fact about secondary indexes in NoSQL databases is that in the majority of the NoSQL databases, specially the most popular ones, secondary indexes are local to the nodes where they reside. With local secondary indexes, the only way to query only the node where the record resides is using shard key based queries. Therefore, queries based on indexed fields, that are not the shard key, are submitted to all the nodes in the cluster since is not possible to discover in which node the record resides. However the queries executed in each node are faster due the secondary index influence.

2.1.7 NoSQL Map-Reduce Functions

With a single, though complex, SQL query, relational databases are able to aggregate large amounts of data to produce a meaningful result set. From large amounts of data, a relational database can filter records that meet a set of constraints and reduce those filtered records to a single record or small set of records by applying aggregate functions.

For instance, considering the orders table described in Source code 1 in a relational database, with a single query it is possible to retrieve the quantity of units sold in the last month of a specific product. However, since a large collection (or column family) may be sharded across many independent nodes, it is not possible to submit such query to a distributed NoSQL database.

In order to answer to those type of requests, NoSQL databases implement or provide integration with map-reduce frameworks. Basically, map-reduce is a programming pattern used to process large amounts of data by taking advantage of the parallel processing capacity of distributed systems (DEAN; GHEMAWAT, 2008). The first stage of a map-reduce job filters the aggregates that meet the constraints. Each application of the map function is independent of all others, therefore, they are executed in parallel by all the cluster nodes. In the reduce stage, the aggregate function is applied to the output of the map stage resulting in the desired information. Part of the reduce stage can be executed in parallel by each node. However, there must be a centralized last phase in which a single node is responsible for taking the output of all other nodes and aggregating them in order to obtain the final result.

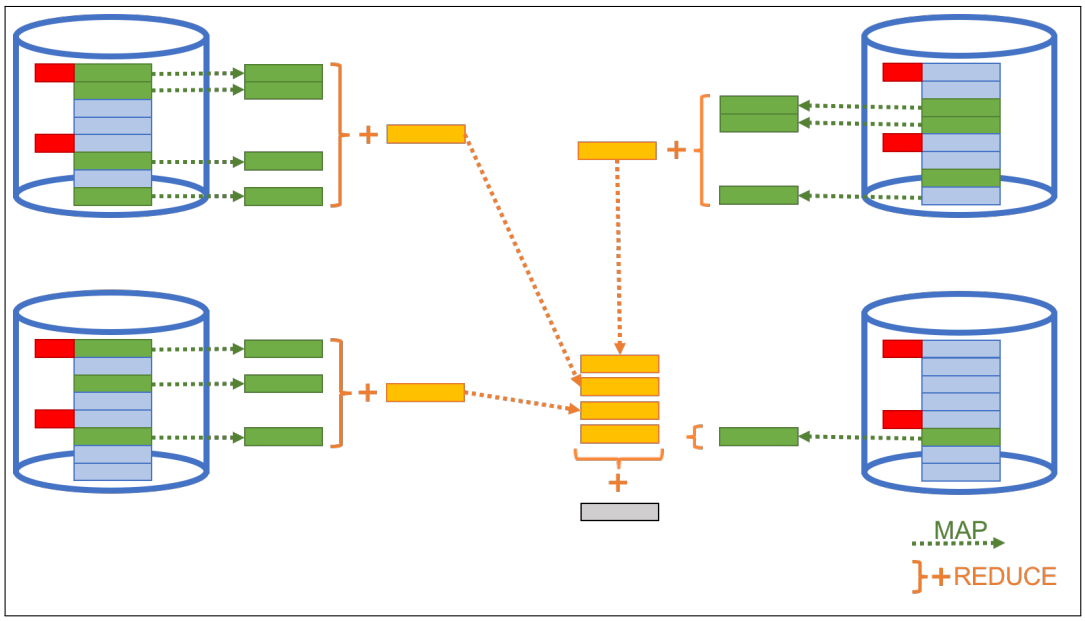
Figure 10 illustrates a map-reduce function applied to a CF database in order to obtain the number of iPhones 7 sold between January 1st and 31th of 2016. The map stage extracts only the records that meet the filtering constraints from the orders aggregates (the first phase in each node). All the nodes executed the map stage in parallel. The reduce stage is divided in two phases, each node aggregates the filtered records in order to obtain the shard partial result. At last, a single node receives the results and executes the final aggregation (dotted-line arrows).

2.1.8 RDBMS versus NoSQL features summary

Table 2 presents a comparison between the main features of aggregate-oriented NoSQL and relational databases.

As enumerated in Table 2, relational databases rely on the relational data model,

Figure 10 – Map-reduce function executed in a .



Source: Created by the author

Table 2 – Relational versus NoSQL databases features comparison.

Feature	**Relational Databases**	**NoSQL Databases**
Data model	Relational model	Domain driven
Transactions	Almost all support ACID	Atomic transactions at the aggregate level
Data types	Strongly typed	Loosely typed
Joins	Yes	Emulated at the application layer
Indexing	Primary, secondary and different storage types	Limited
Design complexity	Persistence layer	Application layer
Data integrity	Responsible is Persistence layer	Shifted at the application layer
Consistency	Strong	Eventual
Query support	Complex and ad-hoc queries	Not suitable for ad-hoc and complex queries
Query language	SQL	Rest, Client libraries, SQL-like DSLs
Query optimization	Responsibility of database	Responsibility is shifted to the application
Complex OLAP fashion queries	SQL statements	Map-Reduce frameworks

Source: Adapted from (LAMLLARI, 2013)

which represents domain entities as rows within relations (tables) that can have relationships with each other. On the other hand, NoSQL databases are domain driven, that means that the entities that must be stored are not translated to another abstract model, like the relational model.

Instead, the business entity is persisted as an aggregate that represents the entity much more directly.

Another important issue enumerated by the table is that relational databases provide a complete and standard set o data types while NoSQL databases do not. NoSQL databases support different sets of data types and also the representations of those data types are not homogeneous among them.

Since NoSQL databases do not support integrity constraints as relational databases do, and provide eventual consistency, those concerns must be handle by the application. Additionally, as with indexes, NoSQL databases do not provide a standard approach for submitting requests to the databases, each NoSQL database implements its own query API.

2.2 DESIGN PATTERNS

As defined by Alexander et al. (1977), "each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice". Despite concerning buildings and towns as firstly introduced by Alexander et al. (1977), the design pattern concept has been applied in many areas since then, including software engineering.

During the many phases of software development process, it is common the rising of recurrent known problems that can be solved by employing strategies well-proven by the industry and software engineering community. Many of those well-proven and widespread solutions have been captured and described as patterns. As stated in (BUSCHMANN et al., 1996), "patterns help you build on the collective experience of skilled software engineers. They capture existing, well-proven experience in software development and help to promote good design practice". Patterns may act as shortcuts to good solutions for less experienced software engineers, since they allow to avoid caveats by presenting solutions based on the knowledge and experience of many more experienced professionals. A pattern captures a solution that is not just an abstract principle or strategy since it provides enough information to guide the implementation of the solution, and allow the identification of the context in which the described solution can be applied.

Since the process of developing and maintaining computer systems involves many areas, there are multiple types of patterns. The following list mentions only a few examples:

- An **architectural pattern** provides a set of predefined subsystems or components, specifies

their responsibilities, and includes rules and guidelines for organizing the relationships between them in order to express the structural organization of a system. The Model-View-Controller (MVC) (BUSCHMANN et al., 1996) pattern is a well-known example.

- A **design pattern** identifies the participating classes and objects, their roles and collaborations, and the distribution of responsibilities in order to solve a general design problem of a subsystem. The Observer (GAMMA et al., 1995) pattern is a widespread example.
- A **Human-Computer Interaction (HCI) pattern** describes a proven solution to a common usability or accessibility problem in a specific context of the Human-Computer Interaction domain. Card navigation and Springboard are examples of navigation patterns for mobile devices (NEIL, 2014).
- An **integration pattern** (a.k.a enterprise integration patterns) describes a high-level architectural solution for allowing different systems to communicate. The Canonical Data Model pattern, described in (HOHPE; WOOLF, 2004), minimizes dependencies when integrating applications that use different data formats.
- A **database pattern** describes a data modelling schema for persisting data in order to solve a storage issue related to space usage, performance or consistency. For a schemaless NoSQL database, a database pattern still describes a schema since the application imposes an implicit schema when storing data. The Incrementing Key pattern, named as Auto Number For Most Tables in (HARATY; STEPHAN, 2013), is a widespread approach for generating surrogate keys.

Those are just a few examples of patterns identified and employed in different areas. Some set of patterns can be very specifics to subareas or frameworks, for instance, patterns for real-time applications and JEE architectural patterns for the Java Enterprise Edition platform (ALUR et al., 2003).

In addition to providing well-proven solutions based on the software community experience for recurrent problems, and promoting large-scale reuse of those solutions, patterns also improve the communication between the members of a team and systems documentation. When a group of professionals share knowledge related to a set of relevant patterns, they can precisely communicate design and implementation ideas with few words.

Nonetheless, patterns can also present a few disadvantages. Although a pattern provides a structured solution for a problem, it can increase the complexity of a system if the same problem may be solved by a simpler, despite non-standard, approach. The same way

patterns improve teams communication and systems documentation, they can have the opposite effect for professionals that do not know the patterns used in a system for whom communication and documentation becomes more difficult to understand.

2.2.1 Patterns Templates

A pattern names, abstracts, and identifies key aspects of a common design structure that make it useful for creating a reusable solution (GAMMA et al., 1995). In order to describe precisely and didactically a pattern regarding the addressed problem, the context in which it should be applied, how it solves the problem, and its consequences, the pattern description should be structured, presenting a set of essential elements. According to Gamma et al. (1995) a pattern description must have four fundamental elements:

- a succinct name that acts like an alias which allows to refer to a problem, its solution and consequences;
- the problem description, in order to identify when to apply the pattern;
- the solution, which describes the strategy and elements employed to solve the problem;
- and the consequences, the results and trade-offs of applying the pattern since they are critical for understanding the costs and benefits of applying the pattern.

Many authors (Alexander et al. (1977), Buschmann et al. (1996), Gamma et al. (1995), Hohpe & Woolf (2004)) published structured templates that include the four before-mentioned elements, for describing patterns. Those templates are didactic, unambiguous, and well organized forms for describing a pattern with precision and sufficient information so it can be correctly applied and implemented. Since there are multiple templates for describing patterns, for some applications, domains or occasions, some templates may be more adequate than others. Certain patterns templates, like the ones presented by Gamma et al. (1995) and Buschmann et al. (1996), have become more well-known by the community. One of the most widespread formats is the Canonical form, which is derived from the Alexadrian form, and contains the following sections:

- **Name**, a meaningful name.
- **Problem**, a statement that succinctly describes the problem.
- **Context**, the scenario and situation in which the problem rises.
- **Forces**, a description of relevant forces and constraints that make the problem difficult to solve.

- **Solution**, the description of the proven solution that solves the problem.
- **Examples**, sample applications that implement the pattern.
- **Resulting Context**, the state of the system after the pattern has been applied, i.e. its consequences
- **Rationale**, explanation of steps or rules of the pattern.
- **Related patterns**, the static and dynamic relationships with other patterns.
- **Known use**, occurrences of the pattern and its application within existing systems.

Describing patterns following a structured form, like the Canonical or other well-structured form, helps to quickly find information about the pattern, like its applicability and consequences, and to compare the pattern to others within a catalog of patterns in order to implement the correct pattern to solve a problem.

2.3 CONCLUSION

This chapter has presented the fundamental concepts that all distributed aggregate-oriented NoSQL databases are based on. The differences between relational databases and NoSQL databases have been presented in order to describe the functionalities not implemented by NoSQL databases and which make them able to provide greater scalability. The types of NoSQL databases have also been explained. At last, a brief review of patterns in the software engineering context have been presented.

The next chapter, Related Work, presents work that related to this work in terms of research subject and tools implementations.

3 RELATED WORK

This chapter discusses the main work related to the research reported in this master thesis. The first section reunites work that concern patterns and modeling techniques applied to aggregate-oriented NoSQL databases, specially those supported by benchmarks executed with YCSB workloads. The second section presents work that, like this research, have developed extensions to the YCSB framework in order to attend to the research necessities.

The related work have been gathered during the development of the research presented in this work by consulting the following online platforms: ACM Digital Library[1], IEEE Xplore[2], Google Scholar[3], ResearchGate[4], and Springer Link[5]. The search string used consisted in a series of combinations derived from the following words: "nosql", "database", "modeling", "techniques", "patterns", "benchmark", "ycsb", and "scalability".

3.1 PATTERNS AND MODELING TECHNIQUES

Haraty & Stephan (2013) present 24 patterns addressed to relational databases. The patterns presented by the authors have been discovered by searching multiple open source repositories with the help of Google Code Search[6]. Files with an ".SQL"extension and containing the words "CREATE TABLE"have been inspected in order to identify common characteristics across the database schemas versioned in the repositories.

Despite concerning database patterns as this work does, the modeling approaches presented in (HARATY; STEPHAN, 2013) are addressed to relational databases, on the other hand, the patterns presented in this work are addressed to NoSQL databases. Haraty & Stephan (2013) have presented the 24 patterns on a very succinct way. Differently from this work, in (HARATY; STEPHAN, 2013) no pattern form have been used to structure the pattern description. Each pattern is briefly described with few lines, without discussing their results, consequences, or presenting an example.

Kaur & Rani (2013) explain basic data modeling techniques and query syntax in the three types of NoSQL databases addressed in this work (key-value, document-oriented, and

[1] http://dl.acm.org/
[2] http://ieeexplore.ieee.org
[3] https://scholar.google.com.br/
[4] https://www.researchgate.net/
[5] http://link.springer.com/
[6] Google Code Search API is no longer available. It was shutdown by Google on January 15 of 2012.

column-family) and graph databases. Similarly to this work, which uses examples to describe the patterns, the modeling techniques described in (KAUR; RANI, 2013) are also explained with the help of an example application, a news web site. However, while the authors present basic techniques, this work addresses scalable techniques, and the examples used to illustrate the patterns are accompanied by workloads used to compare the basic techniques to the scalable ones.

Like in (KAUR; RANI, 2013), Tauro et al. (2013) also presents the different types of NoSQL databases, but with higher level of details, and also discusses the querying and replication models of each one. The authors also compare the types of NoSQL databases regarding their scalability levels. However, the discussion presented in the work is not deep and supported by any tests. On the other hand, this work does compare different databases regarding their scalability levels, and different modeling approaches for the same database type.

Vera et al. (2015) proposes a NoSQL data modeling approach in the form of diagrams, introducing modeling techniques that can be used with DO databases, and present a case study in order to validate the proposed model. Some visual elements are used to describe associations between entities based on basic document nesting techniques. However, since the work in (VERA et al., 2015) is a work based on modeling notation, it does no present how those techniques can be implemented in practice and their consequences.

Silberstein et al. (2010) makes a deep analysis about a selective approach that combines two patterns addressed in Chapter 5 of this work: Fan-out on Read and Fan-out on Write. The authors suggest that the two strategies can be combined in order to take advantage of each other's strengths and minimize their weaknesses. Similar to this work, (SILBERSTEIN et al., 2010) presents custom tests executed in order to prove the efficiency of the suggested approach.

On the other hand, this work focuses on didactically presenting a set of patterns among which the Fan-out on Write approach is included, and compares it to its counterpart implementation, the Fan-out on Read pattern. Therefore, this work complements the research presented in (SILBERSTEIN et al., 2010). Silberstein et al. (2010) does not describe which tools have been used to execute the tests and does not make the tests source code publicly available. Differently, in this research, the tests that compare the before-mentioned patterns are publicly available in order to allow readers to test the trade-offs related to those patterns in their own environment, and with their own workload parameters.

Pirzadeh et al. (2012) present a technique for enabling global range queries in hash-partitioned key-value datastores that do not support range queries. That technique consists on manually implementing a distributed range index based on BLink trees (a distributed variant of B-tree) in top of the key-value store. YCSB workloads have been developed in order to compare the BLink-based index with NoSQL database that support range queries regarding their scalability. The YCSB workloads were executed against Cassandra and HBase, which support range queries, and Valdemort, which does not support range queries. The authors conclude that no approach presents a clear superiority and further research would be interesting in order to better analyse the approaches.

(PIRZADEH et al., 2012) is similar to this work since it compares different modeling approaches regarding their scalability. However, the patterns presented in this work are not focused on providing features not implemented by the database. Instead, it compares basic approaches that rely on features provided by the database to alternative approaches regarding their scalability. In fact, the Enumerable Keys pattern presented in Chapter 5 suggests avoiding range queries supported by range indexes in specific scenarios and suggests the adoption of another scalable approach that is recurrently employed. Additionally, since the approaches suggested in this work are widespread and often adopted in systems that use distributed NoSQL databases, they are presented as patterns. On the other hand, the solution presented in (PIRZADEH et al., 2012) is not an often employed technique since additional research is desirable in order to obtain clearer conclusions.

Shirazi et al. (2012) present a pattern that has the purpose of enabling the migration of data from a column-family NoSQL database to a graph database, and vice-versa. It is related to this work regarding the effort of using a design pattern to reduce the coupling between the stored data and a specific NoSQL product. However, instead of using a formal pattern format like this work does, the authors present a sequence of steps for enabling the data migration. Additionally, the pattern presented in (SHIRAZI et al., 2012) describes the data migration between HBase and Neo4j (and vive-versa), but it is not clear if the sequence of steps can be applied for migrating data between any column-family database to any graph database. On the other hand, this work does not restrict the patterns to any specific database product.

In (STRAUCH et al., 2013), four patterns are presented using the pattern template created by Hohpe & Woolf (2004), which is also used to describe the patterns presented in this work. The four patterns presented in (STRAUCH et al., 2013) are focused in moving the data

layer of applications to the cloud by implementing features often provided by local relational databases and not provided by cloud databases, both NoSQL and relational ones. The work in (STRAUCH et al., 2013) differs from this one because the patterns described in that work are much more generic and abstract, and do not provide enough details to guide an implementation without further research. The examples used in (STRAUCH et al., 2013) to describe each pattern are concise and not supported by tests.

Mior (2014) also concerns about the optimization of NoSQL database modeling. This work uses workloads as a tool for demonstrating the differences between modeling patterns for NoSQL databases regarding their levels of scalability. Differently, Mior (2014) proposes the use of workloads as a phase of an automated process that suggests improvements that may be applied to NoSQL database schemas. Mior (2014) describes a model to calculate the costs of common workloads handled by NoSQL databases. However, since it was presented as a proposal, it does not describe how workload metrics would be generated and processed.

Costa et al. (2015) present the Sharding by Hash Partition pattern for distributed databases as an approach that provides good load balancing. The authors use the same pattern template employed in this work and was a previous step in the research that resulted in this work. Costa et al. (2015) also present the other three sharding approaches (mentioned in Chapter 2), however, it does not use workload tools to compare the pattern with the other sharding approaches.

3.2 YCSB EXTENSIONS

Yahoo! Cloud Serving Benchmark (YCSB) (COOPER et al., 2010) was the tool chosen for implementing and executing the patterns tests described in Chapter 5. However, it demanded the development of additional features in order to attend to the requirements of this research.

Patil et al. (2011) present the development of a tool called YCSB++ that consists of extensions added to YCSB in order to enable the test and debug of advanced features provided by NoSQL databases that are not possible to debug with the original YCSB implementation. The CF NoSQL databases HBase and Acumulo are used in (PATIL et al., 2011) to demonstrate those additional features. YCSB++ provides valuable new features like parallel testing, consistency time evaluation and access control tests among others. However, despite adding valuable features to YCSB, YCSB++ still does not provide some functionalities required by the tests reported in

this work like access to the native datastores client APIs, easy specification of native exceptions, and workload instances with multiple clusters connections.

Xia et al. (2014) present a tool designed to benchmark analytical queries over social media systems called Benchmarking Social Media Analytics (BSMA). BSMA provides real-life datasets extracted from Twitter and implements a data generator capable of generating social networks and synthetic timelines data. Similarly to this work, BSMA provides a set of workloads, however, they are focused in benchmarking style queries. As this work, Xia et al. (2014) reports that YCSB was extended in order create a toolkit for measuring and reporting the performance of systems tested with BSMA workloads. The authors mention a feature called Scalability Over Data Volume as one of the features of the toolkit adapted from YCSB and a prototype system implemented in order to test and demonstrate BSMA. However, the prototype and the tests are superficially described and the tests results are not reported.

Most distributed NoSQL databases do not support transactions. However, some distributed non-relational databases, like Spanner (COOPER, 2013), also classified as NewSQL, do support transactions across cluster nodes. Dey et al. (2014) describe a tool called YCSB+T, an extension of YCSB that wraps database operations within transactions in order to benchmark the transactions overhead in distributed non-relational databases. The authors describe two additional tiers added to the common YCSB database interaction interface that allows measuring the overhead added to individual operations when executed in the context of a transaction, and detecting consistency anomalies during the execution of a workload. The authors also describe a workload created to benchmark distributed databases that implement transactions. Differently, from (DEY et al., 2014), the YCSB extensions implemented in this research do not reuse the databases interaction interface provided by YCSB. Instead, the native API of database client drivers is exposed.

3.3 CONCLUSION

This chapter has presented 10 researches that are related to this work because they are also based on database modeling patterns or strategies. Similar to this one, some of the presented work also employ a benchmark tool in order to test or validate their approaches. The last 3 researches presented in this chapter are relate to this work because they also have provided a technological contribution by implementing extensions to the benchmarking tool employed.

The next chapter, Methodology, describes how the patterns have been identified and

tested, and the extensions developed to the YCSB benchmarking tool in order to implement the tests workloads.

4 METHODOLOGY

This chapter describes the methodology used in the research that resulted in this master thesis. The first section (4.1) presents the sources from where the patterns have been identified and the form used to structure the patterns description. The second section (4.2) describes the tests that compare the patterns with more basic approaches and presents the tools used to implement them. And the last section (4.3) presents the cloud environment in which the NoSQL databases used in the tests have been deployed and where the tests themselves have been executed.

4.1 PATTERNS RESEARCH

In order to gather a set of best practices employed by the industry in building scalable applications with NoSQL databases, multiple sources of information have been used, such as:

- documentation of relevant NoSQL databases like Cassandra, Couchbase, DynamoDB, Elasticsearch, Google Cloud Bigtable and MongoDB;
- articles from the ACM Digital Library, IEEE Xplore Digital Library and Springer Link;
- articles from tech blogs of companies that support scalable systems that rely on distributed NoSQL databases like eBay Tech Blog[1], The Netflix Tech Blog[2], and from trustworthy and well known tech blogs (InfoQ[3], Thoughworks[4], High Scalability[5], High Scalable Blog[6])
- tech reports from companies that use distributed NoSQL databases to support scalable applications;
- and books related to relational databases, software engineering and NoSQL databases;

This work catalogs a subset of those best practices as patterns. The cataloged patterns do not apply for a single NoSQL database product, but rather to a category of NoSQL databases. The first three patterns presented in this work are basic patterns that should be applied to most applications that have a considerable number of concurrent users, and the fourth pattern is related to near-real time event streams, which is a feature currently present in many online applications.

[1] http://www.ebaytechblog.com/
[2] http://techblog.netflix.com/
[3] https://www.infoq.com/
[4] https://www.thoughtworks.com/insights
[5] http://highscalability.com
[6] https://highlyscalable.wordpress.com/

4.1.1 Pattern Template

The pattern template used for describing the NoSQL scalable patterns presented in Chapter 5 is adapted from the form presented in (HOHPE; WOOLF, 2004). The template structure comprises the following sections:

- **Name** - an identifier for the pattern that indicates what the pattern does. Some pattern names have been given based on how the approach is known by the NoSQL community.
- **Context** - details the scalability problem that the pattern solves. The scenario in which the problem occurs is described with the help of an example, which is also used in the next sections.
- **Problem** - expresses the problem tackled by the pattern as a succinct question in order to allow the reader to quickly determine if it can be used to his/her problem.
- **Forces** - this section enumerates the constraints that make the problem not trivial and may consider an alternative solution that seems promising but does not solve the problem properly.
- **Solution** - explains what should be done to solve the scalability problem. The solution is also described using the example previously presented in the context section and contains figures that act like sketches for a better explanation of the solution.
- **Scalability Tests** - this section is not part of the template presented by HOHPE; WOOLF (2004). It describes scalability workload-based tests involving the solution proposed by the pattern and its counterpart less scalable approach previously mentioned in the context section. The results of the tests are presented in charts that compare the approaches scalability levels.
- **Sidebars** - discusses non-essential technical issues related to the pattern or alternative strategies.
- **Consequences** - this section presents the good and bad consequences of the solution, i.e., the trade-offs involving the solution and new challenges that may arise as a result of the solution.
- **Related patterns** - this sections mentions other patterns or approaches employed by the NoSQL community that relate to the pattern being described.

The main differences between the template presented in (HOHPE; WOOLF, 2004) and the template used in this work are the use of the example across the pattern sections, while the other template has an example section, and the scalability tests section, which substitutes the

example section and describes workload-based tests that use the example as scenario.

4.2 SCALABILITY TESTS AND TOOLS

Since there are different strategies of modeling the same problem in NoSQL databases and one of the benefits of this work is to introduce new users of distributed NoSQL databases to modelling strategies focused in scalability and which are generally not presented in NoSQL database books, tests have been executed in order to compare each pattern to a more ordinary, or less complex, approach. The quantified comparison between the approaches may help readers to decide if the downsides introduced by a pattern will be compensated by the benefits for a particular project, and makes easier to notice how much the pattern adoption will contribute to the system scalability in a more tangible way.

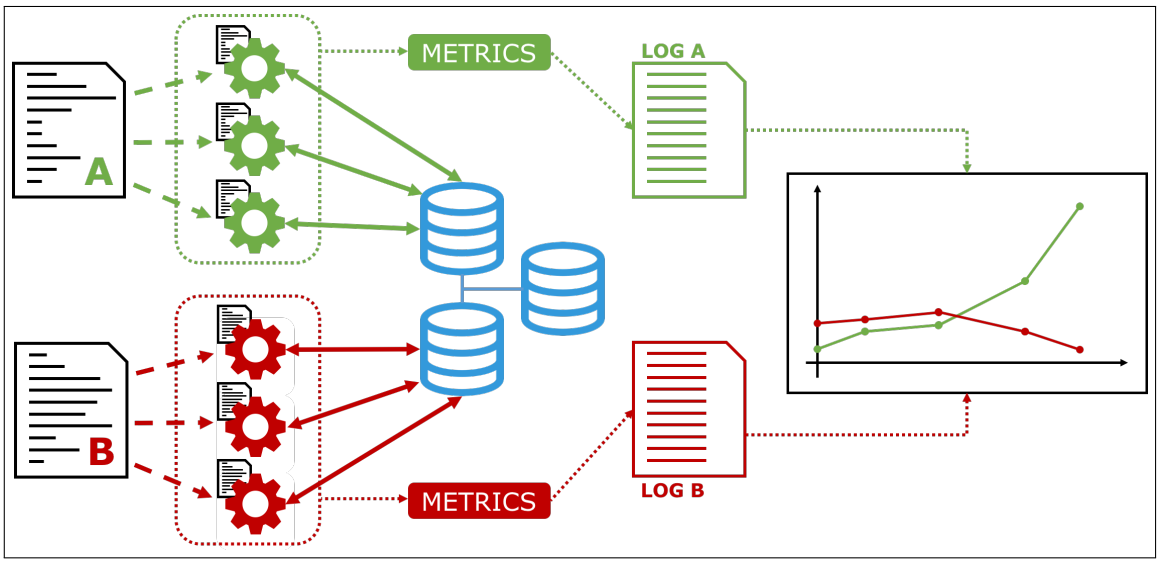
The tests consist of the executions of workloads that simulate many concurrent users submitting requests to the NoSQL database cluster. However, each workload is implemented according to the pattern and basic approach being tested.

In order to simulate an application with several concurrent users, a workload execution is composed by multiple client threads. In a test, each workload is executed several times in order to collect data that indicates how each implementation strategy (the basic one and the pattern) handles an increasing number of clients. That means that each time the same workload is executed, the number of client threads is increased.

Figure 11 describes the tests explanation so far. The figure shows two workloads implementations: workload A implements a scalable pattern, and workload B implements a non-scalable pattern. Multiple threads submit simultaneous requests to the database according to the code implemented in the workload. The metrics obtained during the workload execution are used to plot charts and to compare the behaviour of the workloads, i.e., the scalable and non-scalable patterns. Figure 11 shows a simplification of the tests flow. In fact, a workload is executed several times, and, for each execution, the number of threads is increased. After gathering the log for all the executions of the workloads, the comparison chart is built.

Many NoSQL databases are schemaless and, in that case, the application imposes the data implicit schema. Therefore, for those cases, the executable code of the workload carries all the pattern implementation, i.e., the pattern implementation relies solely on the application code. However, for NoSQL databases that are not schemaless, like Cassandra, when its CQL interface is used, the pattern implementation also depends on the database. In those cases the

Figure 11 – Simplified patterns tests flow.



Source: Created by the author

pattern suggests how the database schema must be modeled. Cassandra maintainers encourage the use of CQL as the standard interface for interacting with the database, and discourage the use of its predecessor protocol, Thrift, which will soon become deprecated.

4.2.1 Tests Metrics

At the end of each workload execution, a set of metrics related to the NoSQL database cluster performance is generated: throughput, average latency and 95th percentile latency. Those are common metrics used to quantify the performance of the database as in many other computational systems.

The Input/Ouput per Second (IOPS) metric is frequently used in many database benchmarking tests. However, the IOPS metric is not appropriate for measuring the performance in the pattern tests presented in this work due to the distributed architecture of the database clusters employed in the tests. The multiple nodes that compose a cluster may present individual high IOPS values but, despite the individual high throughput of each node, the whole distribute database system may present high latency values due to nodes network communication overhead. Therefore, for the tests presented in this work, the IOPS metrics does not provide for the user an appropriate perception of the system performance.

In the database domain, throughput is a measure of how many operations a system can process in a given amount of time. Generally, it is measured in operations per second. In the pattern tests described in this work, a workload executes many transactions against a database,

and the throughput measures how many transactions per second the database cluster processed. In the context of the patterns tests, a transaction refers to a set of subsequent atomic operations. For instance, for a transaction composed by three update requests, the throughput measures how many times in a second the database processed multiple transactions composed by those three update operations.

The average latency for a set of requests submitted to a database corresponds to the average delay between the request submission and the database successful response for that request. Generally, for database systems, the average latency is measured in milliseconds. As for the throughput, in the pattern test, the average latency does not measures the average latency for a set of atomic operations, instead, it measures the average latency for a set of transactions that can be composed by multiple atomic operations.

In the database context, the 95th percentile latency points the value for which 95 percent of requests latency are smaller. That means that 5 percent of the requests submitted to the database presented a latency value greater than the value indicated by the 95th percentile metric. For instance, the average latency for a set of 1 million transactions submitted to the database is 10 milliseconds, which can be a satisfactory value for a particular system. However, the 95th percentile latency is 2 thousand milliseconds. That means that 5 percent of the transactions are taking at least 2 seconds to be processed, and that latency may be prohibitive to many systems. There are other important values of percentile latency like 90 and 99, but this work collects only the 95th percentile latency. The charts that demonstrate the 95th percentile latency for the patterns are shown in Appendix A.

4.2.2 Yahoo! Cloud Serving Benchmark (YCSB)

As mentioned in the previous section, a workload is executed several times, and for each subsequent execution, the number of client threads is increased in order to generate data about how the pattern handles the concurrency growth. In order to implement the multithreaded workloads and measure database metrics, a widespread tool called Yahoo! Cloud Serving Benchmark (YCSB)[7] (COOPER et al., 2010) has been used.

The YCSB project comprises a framework, written in Java, and a common set of workloads for evaluating the performance of different key-value datastores. A common use of the YCSB tool is benchmarking different database systems and using the generated metrics to

[7] https://github.com/brianfrankcooper/YCSB/wiki

compare them. However, in this research, despite using more than one datastore to test some patterns, the YCSB tool was used to compare different implementations for a problem in the same database rather than comparing databases.

A YCSB workload consists of a class that extends the abstract class `com.yahoo.ycsb.Workload`, from the YCSB framework, and implements the operation that must be executed with the datastore, e.g., querying a set of documents based in a search key or updating a record, and a set of parameters that configure the workload execution, e.g., the number of times the transaction must be executed and the number of threads that will execute the workload. During the workload execution, the YCSB client gathers information about the workload execution itself and database performance:

- runtime, measured in milliseconds;
- throughput, measured in operations per second;
- latency, measured in milliseconds;
- 95th percentile latency, measured in milliseconds;
- 99th percentile latency, measured in milliseconds.

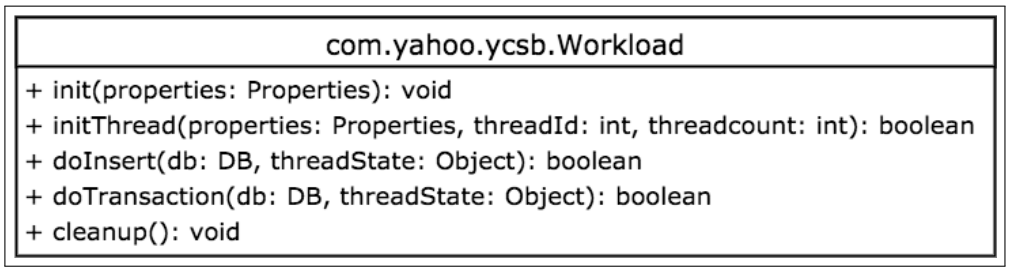
The metrics collected for the test patterns presented in this work do not include runtime and 99th percentile. Since the average latency value is enough for comparing the patterns behaviour, it is not necessary to collect both the 95th and 99th percentile latency. In this work, only the 95th percentile is used.

As described in the previous section, the metrics listed above are not related to the number of atomic operations the database executes. Actually, they are related to how many workload transactions are executed. For instance, a workload transaction consists of retrieving ten records from ten different tables in a database. The throughput is not the average number of read operations executed in a second by the database, but is rather calculated over the number of times the set of ten read operations were executed in a second by the database. The same is valid for the other metrics.

The YCSB client provides two executable phases for a workload: the loading phase and the transactions phase. During the loading phase, it is possible to load the database with data that will be necessary during the transactions phase. Generally, the transactions phase is the test itself. However, it is possible to gather metrics in both phases. The metrics logged by the YCSB client at the end of a workload execution are calculated over the total number of transactions executed during the transactions or loading phase. Figure 12 shows the UML class

diagram of the `com.yahoo.ycsb.Workload` class. The loading phase must be implemented in the `doInsert` method, and the transactions phase must be implemented in the `doTransaction` method.

Figure 12 – UML class diagram of the YCSB workload.



Source: Created by the author

Depending on the phase that must be executed (loading or transactions), the workload parameter that sets the number of times a transaction must be executed actually determines how many times the `doTransaction` method, or the `doInsert` method, must be executed. The YCSB client tries to equally divide the number of transaction executions among the threads. For instance, if the transactions (or loading) phase must be executed 100 times by 10 threads, each thread will execute the `doTransaction` (or `doInsert`) method 10 times.

The YCSB tool already implements communications with multiple databases, like Cassandra, Couchbase, DynamoDB, HBase, Memchached, MongoDB, Redis, among others. The YCSB client provides a common communication interface to all those databases. As depicted by Figure 12, the first parameter of the `doInsert` and `doTransaction` methods is an object of type `com.yahoo.ycsb.DB`. That is an abstract class that declares the five following basic transactions, each one as a method:

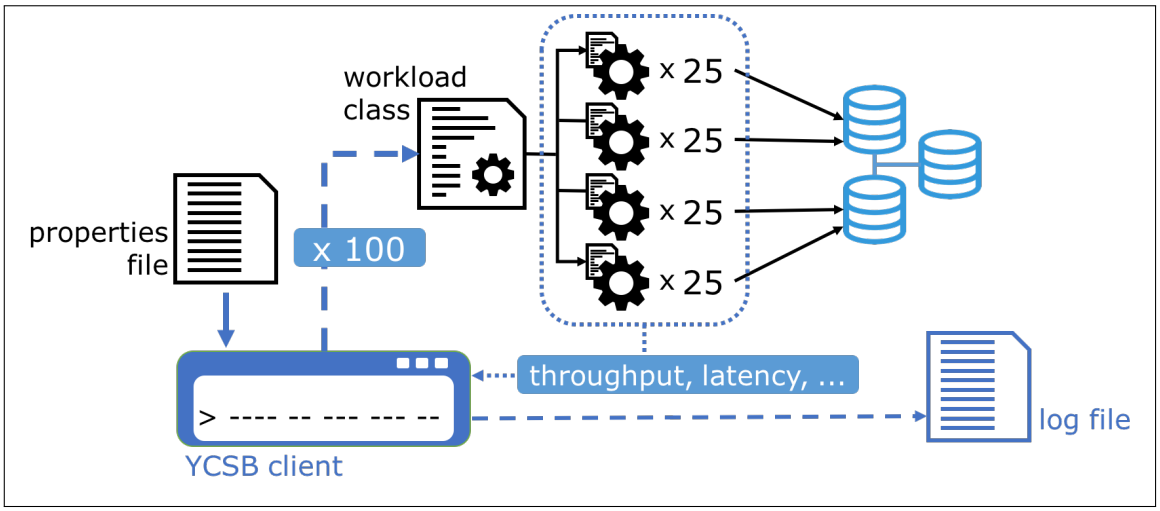
- read a record;
- insert a record;
- update a record;
- delete a record;
- and perform a range scan.

In order to execute a workload, the user must inform through the command line which database is going to be tested. Then, at execution time, the YCSB client loads the appropriate concrete class that implements the `com.yahoo.ycsb.DB` abstract class and implements the specific code to enable communication with the informed database. Therefore, regardless the

database being tested, the `com.yahoo.ycsb.DB` provides a common interface for them.

Figure 13 illustrates the explanation of how the YCSB client works with an example. Through the command line, the user invokes the YCSB client, and informs the workload that must be executed (a subclass of `com.yahoo.ycsb.Workload`) with its parameters. The parameters are listed in a properties configurations file. The properties file informs that the workload must executed 100 times and that the work must be shared by 4 threads. During the whole workload execution, the YCSB client prints a log that, at the end of the workload, will contain the performance metrics.

Figure 13 – An example of a workload executed by the YCSB client.



Source: Created by the author

Unfortunately, the YCSB tool presents some limitations when considering the requirements of the patterns tests described in this work. Consequently, just implementing the workloads was not enough and a few extensions to the tool had to be implemented. Those implementations are described in the next section.

4.2.3 YCSB Extensions

YCSB was one of the projects developed by the Yahoo Labs team. Due to changes in its research teams, including the Yahoo Labs, Yahoo stopped officially supporting the YCSB project. Consequently, many of its libraries that allow the communication with different datastores, called bindings, are outdated, including the binding implementations for Cassandra, Couchbase and MongoDB, which are used for the patterns tests in this work. That was the first issue that motivated the development of some extensions in order to enable the pattern tests.

However, the main problem found with the YCSB client was the fact that it treats all databases as key-value datastores and does not provide access to custom features of the databases. As described in the previous section, the YCSB provides a common interface of communication with all databases (the `com.yahoo.ycsb.DB` abstract class), which provides the five basic transactions mentioned in the previous section: read, insert, update, delete and scan.

The three databases used in the pattern tests (Cassandra, Couchbase and MongoDB) are not key-value datastores only. For those three databases, the content of a stored document or record is not opaque. In addition to providing a key-value interface, like all distributed NoSQL databases do, they provide more specific transactions, which are used by the tests desbribed in Chapter 5, such as:

- Couchbase implements counters (ZABLOCKI, 2015) that can be atomically incremented and have its new value retrieved with a single request;
- Couchbase implements bulk get operations (ZABLOCKI, 2015) that can accept a list of keys as parameter and return all the found documents at once;
- with MongoDB, it is possible to push new values into a list nested in a document with a single request. There is no need to retrieve the document, update the list, and save the document back to the database (MARCHIONI, 2015).

The three abovementioned features are used in tests presented in the next chapter (Chapter 5). However, it is not possible to use them through the common database interface provided by the YCSB client (the `com.yahoo.ycsb.DB` abstract class). In order to access those features it is necessary to interact directly with the client drivers of the databases. Therefore, a parallel hierarchy of classes has been developed in order to facilitate the implementation of YCSB workloads that rely directly on database drivers APIs instead of the common interface provided by the YCSB client. Additionally, a few important features have been added to the developed classes in order to provide functionalities necessaries to the patterns tests.

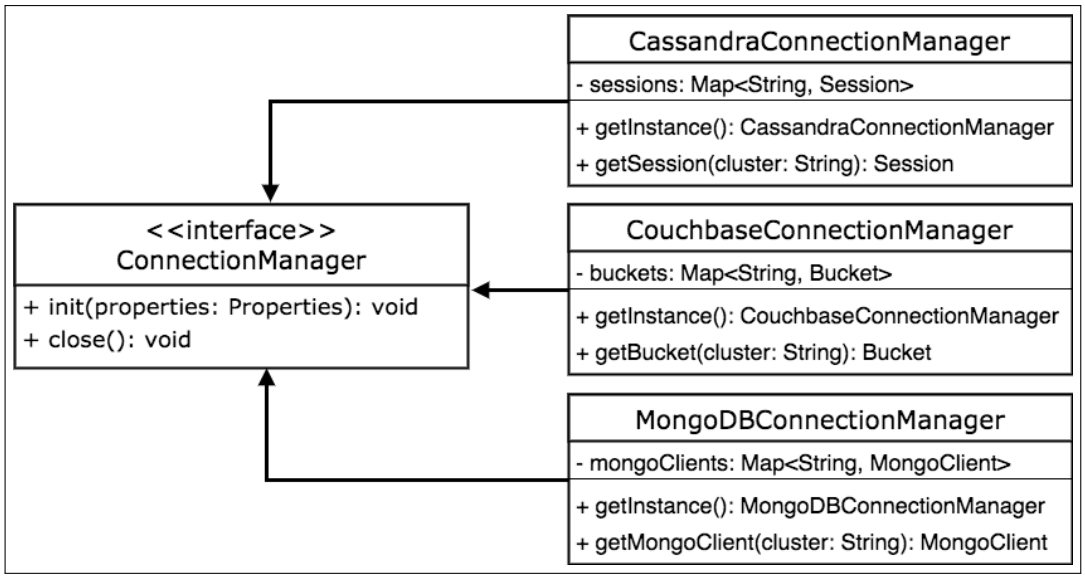
4.2.3.1 Access to Database Drivers APIs and Multiple Clusters Connections

The Java client drivers of each NoSQL database used in the tests provide an object that manages the pool of connections to the database cluster and implement the database transactions: the `Session` object for Cassandra, the `Bucket` object for Couchbase, and the `MongoClient` for MongoDB. For instance, when the Couchbase client driver establishes a connection with a cluster, it returns a `Bucket` object and that object provides the methods

necessary to increment and decrement a counter, among others. Therefore, a Couchbase workload must have access to the `Bucket` object that encapsulates the connection with the cluster and the transactions API.

Since the YCSB client does not provide direct access to those objects, in order to make the whole set of features of a database accessible to the workload classes, a singleton (GAMMA et al., 1995) connection manager class has been developed to each NoSQL database used in the tests. In addition to encapsulate the boilerplate code required to establish a connection, the connection manager classes are able to establish connections with multiple database clusters simultaneously. Figure 14 shows the UML for the connection management classes.

Figure 14 – UML class diagram of the connection manager classes.



Source: Created by the author

The `init()` method shown in Figure 14 receives a `Properties` object containing the information necessary to connect to the clusters. The YCSB client populates the `Properties` object with the properties informed through the command line or properties file. Source code 2 shows an example of a properties file that describes the information necessary to connect to two Couchbase clusters, each one composed by two nodes, simultaneously. The name of the clusters are listed in the `clusters` property. The properties composed by the concatenation of the `nodes` prefix and a cluster name (lines 2 and 3) list the nodes that compose the cluster.

Source code 2 – Properties describing two Couchbase clusters.

```
1 clusters=cluster1,cluster2
2 nodes.cluster1=couchbase1.subnet1,couchbase2.subnet1
```

```
3 nodes.cluster2=couchbase1.subnet2,couchbase2.subnet2
```

As described in Figure 14, each connection manager class has a `Map` attribute that holds the clusters connections as key-value pairs. The key is the name of the cluster and the value is the object that encapsulates the driver API and the connection (the `Session` object for Cassandra, the `Bucket` object for Couchbase, and the `MongoClient` for MongoDB). One Couchbase workload developed for the tests described in Section 5.1 of Chapter 5 uses the multiple clusters connection feature because it simultaneously connects to two database clusters: one in USA and another in Ireland.

4.2.3.2 Handling Retries and Exceptions

When a request submitted to a database cluster fails due to some temporary resource contention condition, e.g., memory threshold exceeded or timeout due to too long request queue, it should be added a delay between the retries. An immediate retry will overload the cluster even more. Therefore, the retries must be performed using a backoff algorithm. Although the YCSB client implements a backoff algorithm, it was designed to be used with the common databases client interface presented previously, which presents the before-mentioned issues: outdated implementation and restriction to only five basic transactions (insert, read, update, delete and scan).

The backoff algorithm must wrap every request submitted to the database cluster. However, every request must be sent using the public interface provided by the client driver API of the database used in the test. As the UML diagram shown in Figure 14 describes, each connection manager class has a `Map` attribute that holds the client driver objects that implement the API for a specific database. Therefore, a point of convergence through which all requests had to pass should be implemented.

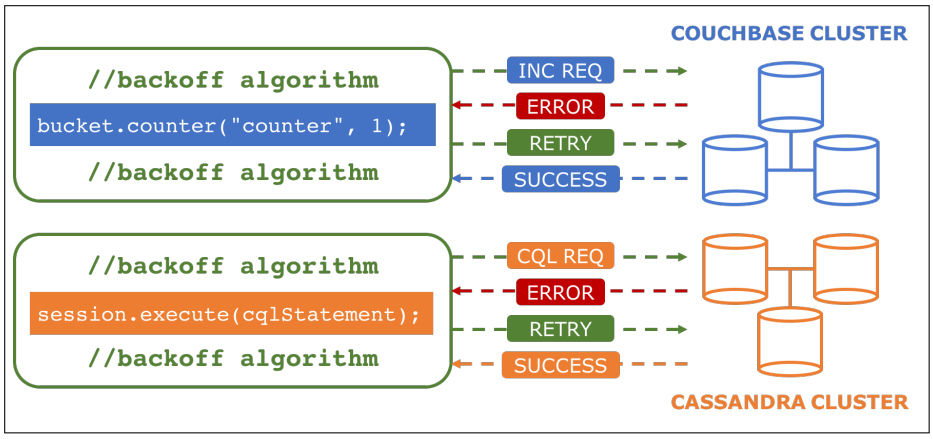
For instance, the `CouchbaseConnectionManager` class holds in its `buckets` attribute a `Bucket` object for each cluster that it is connected. And the `Bucket` class provides the `counter()` method that can be used to increment a counter document as demonstrated in the second line of Source code 3. On the other hand, the `CassandraConnectionManager` class holds in its `sessions` attribute a `Session` object for each cluster which it is connected. The `Session` class provides the `execute()` method that submits a CQL statement to the cluster, as demonstrated in the fourth line of Source code 3.

Source code 3 – Examples of Couchbase increment counter request and Cassandra CQL statement submission.

```
1 //Increments a Couchbase counter by one.
2 bucket.counter("counter::usercounter", 1);
3 //Inserts a user in a Cassandra table.
4 session.execute("INSERT INTO user (id, email, information) VALUES
    (5425, "user1@email.com", "...");
```

The backoff algorithm must encapsulate both transactions, however, they are executed by different objects and take different types as parameters. As described in Figure 15, the backoff implementation must accept a transaction request code and wraps its execution inside the backoff algorithm.

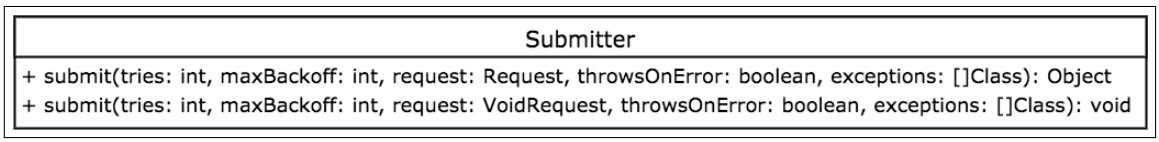
Figure 15 – The backoff algorithm wraps requests submitted by different APIs.



Java 8 lambda expressions have been employed in order to encapsulate at runtime the different requests submitted to the databases clusters in a class that implements the backoff algorithm. As defined by Cameron (2014), "Lambda expressions (commonly know as closures or anonymous methods) allow you to encapsulate a block of code in an anonymous method (i.e. a method without a name or enclosing class), and pass it to another method which will invoke it at the appropriate time". A simple exponential backoff algorithm was implemented in the methods of the new class `Submitter`, described by the UML diagram shown in Figure 16.

The overloaded `submit` method implements the exponential backoff algorithm and accepts the lambda expression (the `Request` or `VoidRequest` interfaces) that submits a request to the database as one of its parameters. In fact, lambda expressions are anonymous classes that implement functional interfaces (CAMERON, 2014), which are interfaces that define a single method that adheres to a naming convention.

Figure 16 – UML class diagram of the class `Submitter`.



Source: Created by the author

Source code 4 shows a snippet of a workload source code that inserts two records in two Cassandra tables using the `Submitter` class (lines 2 and 3). The third argument passed to the `submit` method as a lambda expression is an insert request submitted to the Cassandra cluster using the API of its own client driver. Source code 5 shows another source code snippet from a workload that inserts two documents in a Couchbase bucket also using the `Submitter` class (lines 2 and 3). The first two parameters of the `submit` methods are the number of attempts for a request and the maximum backoff time a thread can sleep before retrying. Both parameters can be specified as properties in a properties file or directly through the command line. The `Submitter` class allows to submit any request to a database cluster regardless the API used to implement the transaction.

Source code 4 – Two CQL statements submitted by the `Submitter` class.

```
1 try {
2     Submitter.submit(tries, maxBackoff, () -> session.execute(
    insertStatements.get("user")));
3     Submitter.submit(tries, maxBackoff, () -> session.execute(
    insertStatements.get("email")));
4     Measurements.getMeasurements().measure("INSERT-USER", (int) (
    System.currentTimeMillis() - start));
5 } catch (SubmitterException e) {
6     Measurements.getMeasurements().measure("INSERT-USER-FAILED", (int
    ) (System.currentTimeMillis() - start));
7     System.err.println(e.getMessage());
8 }
```

Source code 5 – `Submitter` class executes two requests to insert a document in Couchbase.

```
1 try {
2     Submitter.submit(tries, maxBackoff, () -> bucket.upsert(
    userDocument));
3     Submitter.submit(tries, maxBackoff, () -> bucket.upsert(
    emailDocument));
4     Measurements.getMeasurements().measure("INSERT-USER", (int) (
    System.currentTimeMillis() - start));
5 } catch (SubmitterException e) {
6     Measurements.getMeasurements().measure("INSERT-USER-FAILED", (int
    ) (System.currentTimeMillis() - start));
7     System.err.println(e.getMessage());
```

```
8 }
```

Not always a request fails due to a contention condition. Sometimes it fails due to an error in the workload implementation or an unexpected hardware network outage. Sometimes, the developer implements an alternative algorithm to be executed when a transaction fails during a workload execution. In those cases the request should not be submitted again by the backoff algorithm. With the original implementation of the YCSB client it is not possible to inform to the common databases interaction interface (`com.yahoo.ycsb.DB`) which exceptions should interrupt the backoff algorithm and return to the caller for custom handling.

Therefore, that control was implemented in the `Submitter` class. The UML diagram in Figure 16 shows that the `submit` methods of class `Submitter` accept a boolean flag called `throwsOnError` and an array of classes as the fourth and fifth parameters respectively. When the `throwsOnError` flag is set to true, if any of the exceptions passed as classes arguments are triggered, the backoff algorithm is not executed, the `submit` method returns to the caller immediately. That implementation facilitates the debug and control flow of a workload regardless of the database being used as datastore.

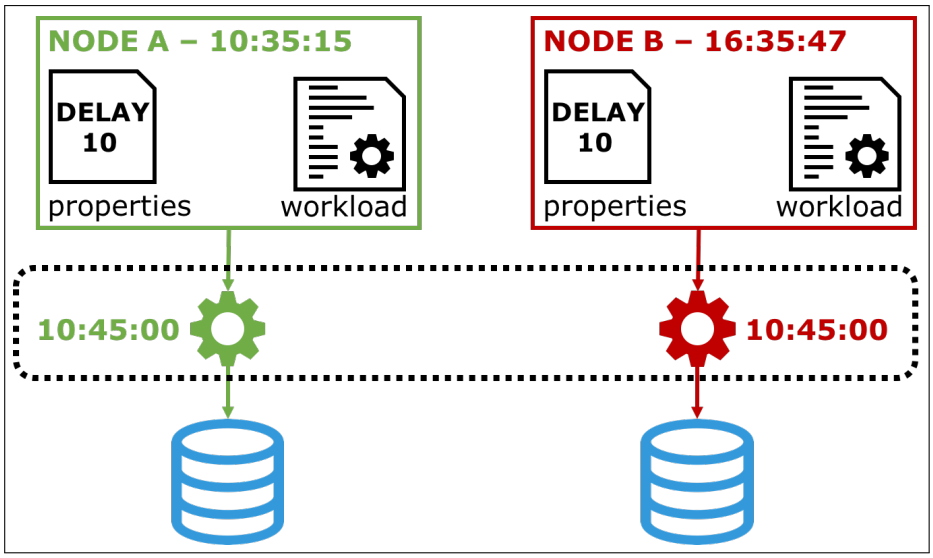
4.2.3.3 Synchronizing Workloads Start

Sometimes a test requires the simultaneous execution of multiple workloads at different client nodes. Consequently the workloads must start sending their requests to the database(s) at the same time. That is another feature that had to be developed in addition to the features provided by the YCSB client. The `PatternWorkload` class has been developed in order to meet that requirement. All new workload classes must extend that class, which in turns, extends the YCSB `Workload` class.

The `PatternWorkload` class implements the workloads synchronization by receiving an initialization parameter that adjusts the workload start time. The example described in Figure 17 helps to understand how it works. The workloads A and B must start at the same time, however they are in different nodes, at different countries. The current time in the country of workload A is 10:35:15, while the current time in the country of workload B is 16:35:47. As delay initialization parameter both workloads receive the value of 10. The `PatternWorkload` class ignores the current seconds value and schedules both workloads to start in 10 minutes. Therefore, workload A will start at 10:45:00 and workload B will start at 16:45:00. Ignoring the

seconds is important because if the user will start both workloads directly from a SSH session, it takes few seconds to change terminal windows and press enter to trigger another workload.

Figure 17 – Workloads in different client nodes start at the same time.



Those are the main features added to the parallel class hierarchy in order to complement the features provided by the YCSB client. The source code for those features are available at https://bitbucket.org/caiohc/.

4.2.4 YCSBtoCSV

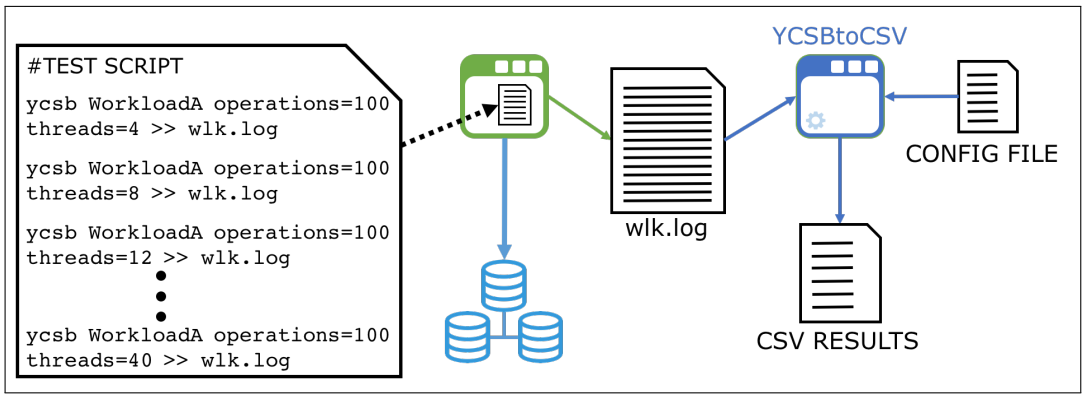
As previously described, in a pattern test, a workload is executed several times, and for each execution, the number of client threads is increased. For example, a workload that implements a pattern must be executed 10 times and for each execution the number of threads is increased by 4, starting from 4 up to 40. Figure 13 may be used as the model that describes the first execution of the described test. For the subsequent executions of the same workload, the number of transactions remains the same, but the number of threads is increased by 4. Each execution generates a log that contains many debug information and the results (runtime, throughput, latency) of the tests at the end.

Since the tests are executed in the Amazon Web Services (AWS) cloud, it makes sense to execute all the tests as quick as possible in order to save money. Therefore, instead of invoking the YCSB client through the command line for each subsequent execution of a test, a script that encapsulate all the YCSB client invocations has been created. That approach also makes the process more productive and less error prone. As illustrated in Figure 18, the script

appends the log of all the executions to a single file. Consequently, the task of moving the results of the tests from that big cluttered log file to a spreadsheet in order to plot the comparison charts is a tedious, slow and error prone process.

In order to facilitate that task and make it more reliable, a utility application called YCSBtoCSV has been developed. As demonstrated in Figure 18, the YCSBtoCSV parses the log of the tests and outputs a file containing only the results of the workload executions. In addition to the test log file, the YCSBtoCSV utility requires a configuration file. The configuration file includes the metrics that must be gathered, the YCSB param that aggregates the metrics (e.g., the threads flag), the seperator character for the resulting CSV file, and others. The YCSBtoCSB utility source code is available at https://bitbucket.org/caiohc/.

Figure 18 – The YCSBtoCSB parses the test log and outputs a CSV file with the results.



4.3 TEST ENVIRONMENT

In order to create the distributed NoSQL database clusters used in the patterns tests, the Amazon Elastic Compute Cloud (EC2)[8], a service that provides resizable computing capacity in the cloud, has been adopted. It is possible to deploy several server instances in the cloud network and have complete control over them. The AWS cloud was adopted because it is widely known in the cloud computing community and provides great documentation.

The clusters nodes have not been shared between the databases, i.e. each EC2 instance had only one database instance deployed. All nodes that composed the databases clusters were cloud shared t2.medium EC2 instances, which have 2 virtual CPUs and 4GB of RAM memory. There was no need for very powerful hardware resource since the objective of the

[8] https://aws.amazon.com/ec2/

tests was to compare different approaches in the same database and hardware, not benchmarking the databases used in the tests.

The pattern tests executed against MongoDB presented in Section 5.2.5 have been executed in a cluster that is two times greater than the other pattern tests presented in this work. The cluster used in that test is composed by 8 nodes while the clusters of the other patterns tests are composed by 4 nodes. Consequently, the 8-nodes cluster is more susceptible to public cloud performance fluctuations. Therefore, in order to reduce the effect of performance fluctuation on the tests executed against that larger cluster, m4.large private instances have been used. Those instances are not shared with other cloud clients. Amazon EC2 does not provide private t2.medium instances, therefore m4.large instances have been used in that case. M4.large instances have 2 virtual CPUs and 8GB or RAM memory.

The operating systems of the EC2 instances was Ubuntu 14.04 LTS and the databases versions were: Cassandra 3.0, Couchbase Community 4.0 and MongoDB 3.2. Version 8 of the Java JRE was necessary in order to execute Cassandra 3.0 and the auxiliary classes implemented to complement the YCSB client. The client node used to execute the tests scripts, including the YCSB client, was also a t2.medium EC2 instance. When an additional client was necessary in order to generate more load to the database, it was also an t2.medium instance.

5 NOSQL SCALABLE PATTERNS

This chapter presents the four scalability patterns for distributed NoSQL databases proposed in this work. The first section (5.1) presents the UUID Key pattern, the second section (5.2) presents the Index Table pattern, the third section (5.3) describes the the Enumerable Keys pattern, and the fourth section (5.4) describes the Fan-out on Write pattern. The last section (5.5) concludes the chapter by presenting a table that summarizes the properties of the each pattern.

5.1 UUID KEY PATTERN

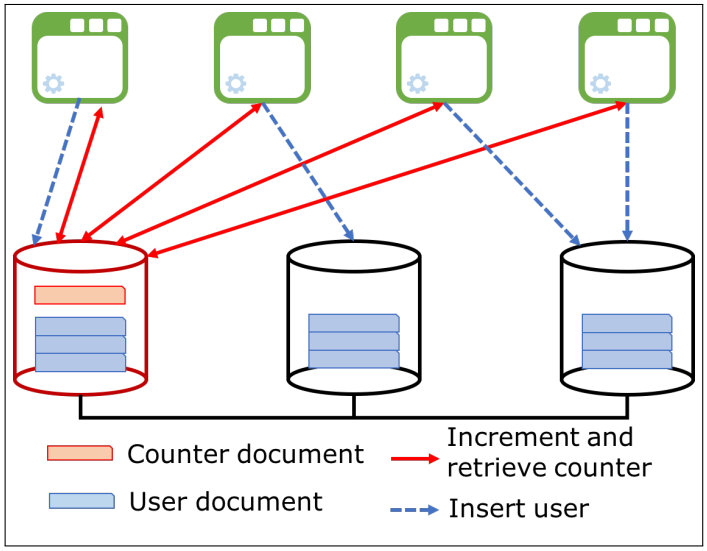
5.1.1 Context

Often, when modeling the domain of an application, it is necessary to create a surrogate key in order to uniquely identify an entity instance in the application context. A very usual pattern is to delegate to the database the responsibility of generating surrogate keys to be used as primary keys. Usually, in relational databases, auto-incrementing fields or sequences are used to generate those keys. That pattern is known as Incrementing Key (IK). The IK pattern can also be used with NoSQL databases, since many of them implement counters whose generated values can be used as primary keys. However, despite the simplicity of the IK pattern, that approach is not appropriate for wide-area distributed NoSQL databases because it may generate availability and consistency issues.

When the IK pattern is used with NoSQL databases sharded collections (COSTA et al., 2015), the requests to store records are distributed across the cluster, while the requests to generate surrogate keys are targeted to a single node. Figure 19 illustrates that scenario. In order to retrieve a surrogate key from the counter, client applications submit an increment and get request (the contiguous red lines) to the node in which the counter is stored. Once the applications got the surrogate keys, the requests to store new records (the dashed blue lines), with the surrogate keys as the records primary keys, are distributed among the nodes.

If the counter is replicated in order to increase system reliability, the increment and get requests are submitted to the counter master copy only. Since the requests for incrementing and retrieving the counter value are targeted to a single node, a Single Point of Failure (SPOF) is created. If the node in which the master counter record resides fails, it will not be possible to generate new surrogate keys until a failover operation completes and another node becomes the

Figure 19 – Sharded collection with primary keys generated by centralized counter.



Source: Created by the author

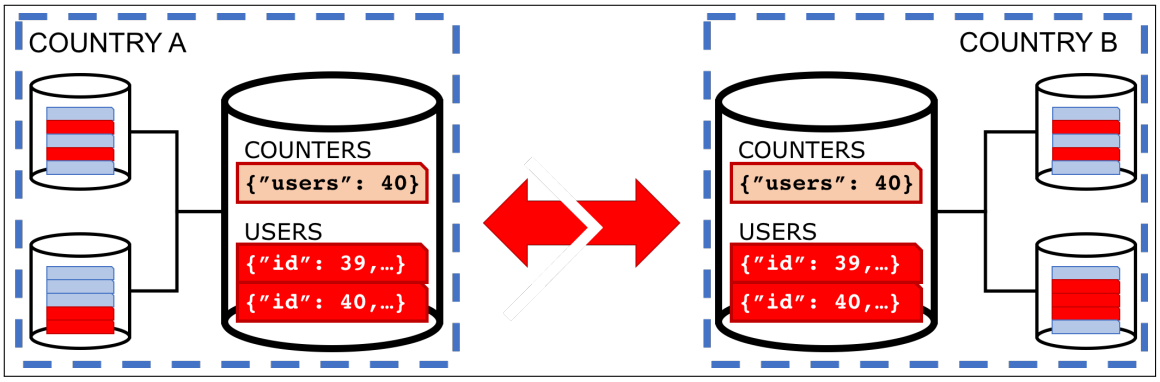
master.

The described issues do not necessarily become problems when using the IK pattern with clusters deployed in low latency networks. However, with wide-area distributed clusters, the network high latency may cause problems. Consider a simple example in order to illustrate that scenario. An application and its NoSQL database are deployed in datacenters of two countries (A and B). An instance of the application and a local NoSQL cluster are deployed in each datacenter. The database clusters are synchronized with bidirectional replication. The application uses the IK pattern in order to generate primary key values for the user records.

In order to keep accepting new users registrations in both countries during a network partition, each database cluster stores a master copy of the counter. The distributed counters are constantly synchronized by the bidirectional replication. However, despite the possibility of keeping generating surrogate keys during the network partition, the cluster cannot synchronize data, generating inconsistencies. The example shown in Figure 20 illustrates that situation. Since the counters could not synchronize during the network partition, records were stored with duplicated primary key values. Figures 20 shows that after increment requests, the values 39 and 40 were used in both clusters as primary keys because the counters could not synchronized their values.

Regardless network partitions, the application may face consistency problems due to the high latency of wide-area networks. Still considering the previously described scenario, clusters A and B are separated by a continental distance. If the application is facing a period of

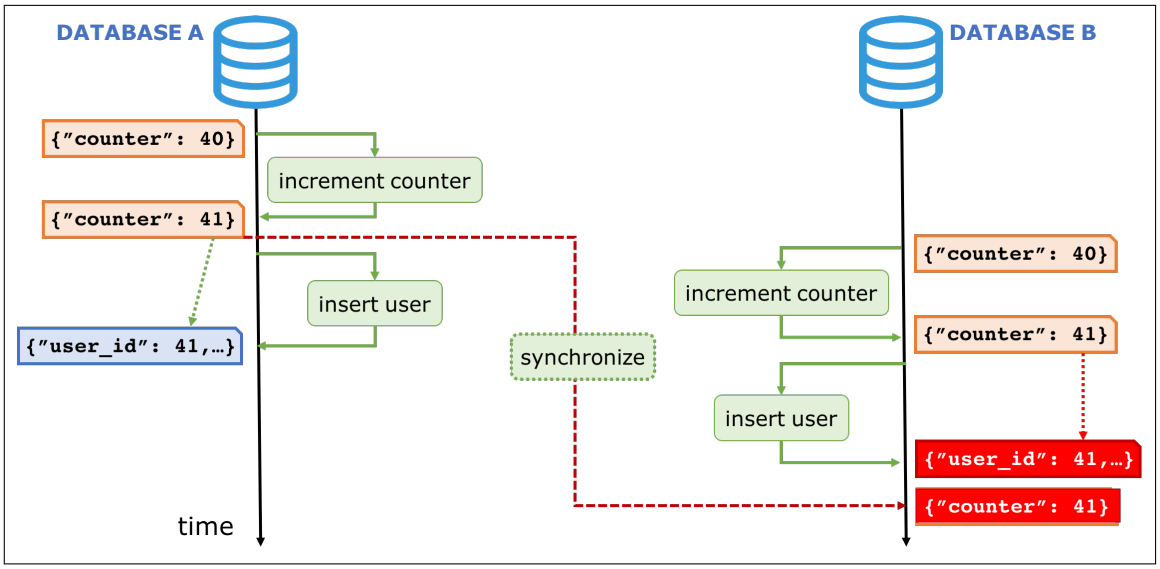
Figure 20 – Inconsistencies generated by increments of replicated counters during network partition.



Source: Created by the author

higher demand, it may not be possible to synchronize the counter values with the necessary speed to keep consistency. Figure 21 illustrates that situation. The counter in cluster A is incremented and its value is used as primary key of a new user record. A few milliseconds later, the same happens in cluster B. Due to the high network latency between the two clusters, the counters were not synchronized in time. Consequently, cluster B used a stale counter value as primary key of a new user record.

Figure 21 – Great latency between clusters A and B causes the use of stale counter values.

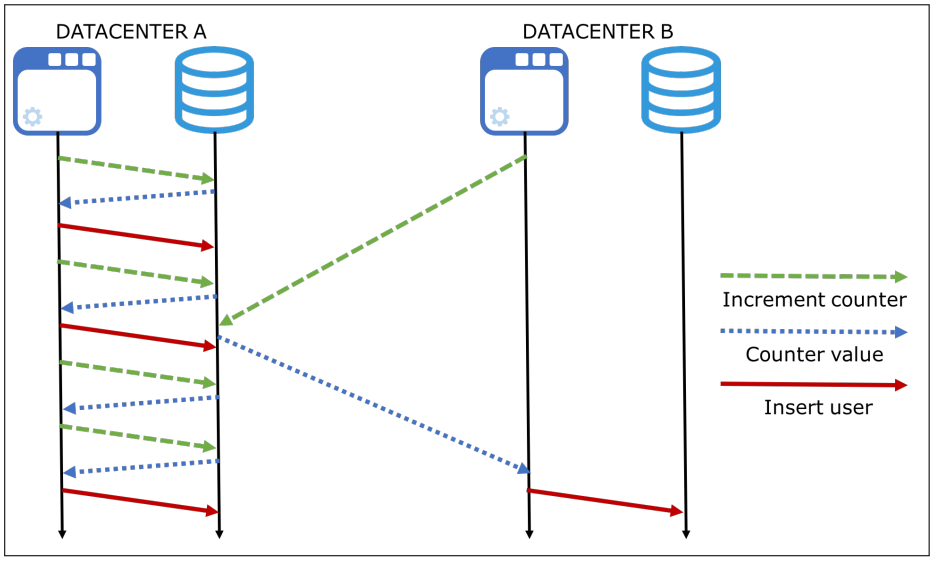


Source: Created by the author

In order to avoid consistency problems, the counter must be centralized in one cluster. Suppose cluster A holds the master counter record. The client users in country B will experiment the high network latency. As illustrated in Figure 22, before every request to store a new user

record in cluster B, the application must request a new value from the master counter record stored in cluster A. That request takes too long, which may be unbearable for the business.

Figure 22 – Application in DC B experiments great latency when incrementing counter in DC A.



Source: Created by the author

The issues described previously demonstrate that the widespread Incrementing Key pattern, or any approach based on a centralized surrogate key generator, is not appropriate for distributed NoSQL databases. Another approach is necessary in order to provide higher consistency and availability levels.

5.1.2 Problem

How to generate surrogate keys for wide-area distributed NoSQL databases without compromising consistency and availability?

5.1.3 Forces

- System availability is impaired when a centralized counter is used as surrogate key generator, since during network partitions, only clients who reach the master node will be able to submit increment requests.
- Distributed counters can be used to generate key values during network partitions. However, consistency is impaired because, even though the count is updated after the network restoration, the records stored during the partition will present duplicate primary key values.

- In wide-area distributed databases, it is difficult to avoid the occurrence of duplicated primary key values, since the distributed counters take too long to synchronize due the high latency of the network.
- When a central coordinator is responsible for generating surrogate keys, as stated by the CAP theorem (BREWER, 2012), it is no possible to maintain availability and consistency during a network partition.

An alternative approach that consists of an extension of the Incrementing Key pattern can be implemented in order to increase the database scalability. An independent counter can be stored in each node of the cluster in order to avoid the centralization of surrogate key requests in a single node. In order to avoid surrogate key duplicated values, the current value of each counter must be concatenated with an additional identifier, such as the node address, generating a unique value.

Despite improving the scalability of the system by distributing the counters across the cluster, the responsibility of equally distributing the surrogate key requests is shifted to the application layer. The application, or an additional tier, must implement a request distribution algorithm in order to choose to which node to submit a surrogate key request. Similar approaches based on the distribution of counters across the cluster, such as setting an offset for each counter, will also require a distribution algorithm.

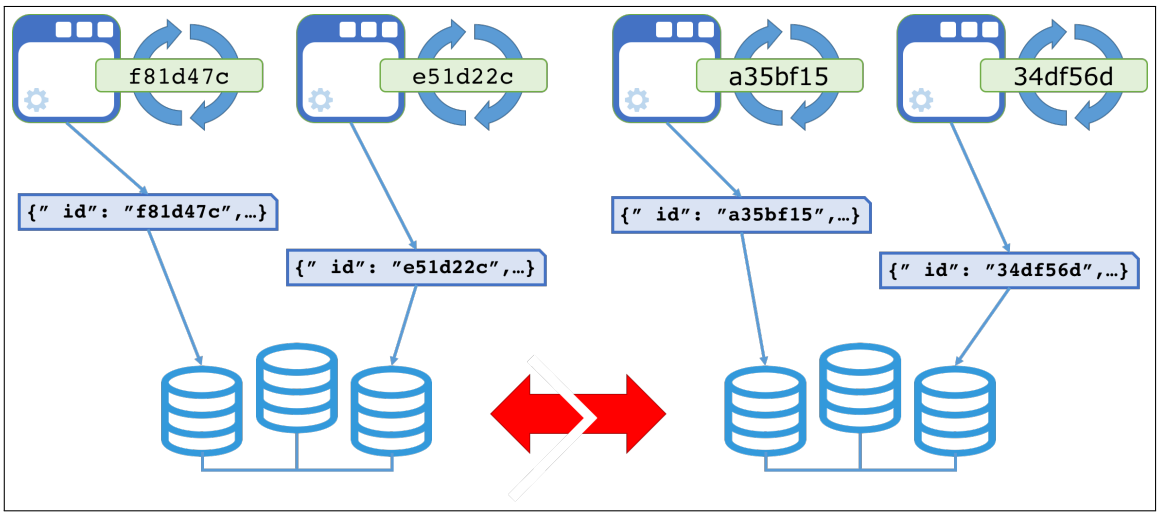
5.1.4 Solution

In order to generate surrogate keys for entities stored in wide-area distributed NoSQL databases and also deliver satisfactory levels of consistency and availability, even during network partitions, Universally Unique Identifiers (UUID) (LEACH et al., 2015) should be used. An UUID, also known as Globally Unique Identifier (GUID), represents a 128-bit value that can be used to uniquely identify objects. It is appropriate for distributed systems since it does not need central coordination in order to be generated.

Figure 23 demonstrates the UUID Key pattern. Each time a client application needs a surrogate key in order to store a new record, the application itself generates a new UUID value that is used as the record primary key. Cluster nodes do not need to communicate since they do not participate in the generation of surrogate keys. Consequently, network partitions do not affect system availability or consistency regarded to the insertion of new records.

Additionally, if the clients generate the surrogate keys, they are not submitted to the

Figure 23 – Client applications generate UUIDs to be used as primary key for new records.



Source: Created by the author

high network latency values imposed by the need of centralizing the generation of surrogate keys by distant nodes, thus also contributing to the availability of the system.

For human-readable purposes, UUIDs are frequently displayed in a canonical format that consists in hexadecimal digits with inserted hyphen characters. For example, f81d4fae-7dec-11d0-a765-00a0c91e6bf6 is an UUID represented in a canonical format.

Although there is a very small possibility of generating duplicate values, that is very unlikely to happen since UUIDs are composed of 128 bits, which makes possible to generate 2^{128} values. RFC 4122 (LEACH et al., 2015) recommends efficient algorithms for generating UUID values and most mainstream programming languages implement those algorithms and offer them through APIs making their use simple.

5.1.5 Pattern Tests

In order to support the previous recommendations, this section presents an example that simulates a scenario in which the UUID Key pattern should be adopted instead of the IK pattern and quantitatively compares both patterns. Based on the example used in the Context section (5.1.1), the tests simulate the registration of new users in a web application directed to the north american and european public. In order to provide a satisfactory experience to the users, there are two instances of the application: the first is deployed in a datacenter located in the United States, and the second is deployed in a datacenter located in Ireland. The NoSQL database in which data is stored is composed by two clusters: one located in the USA datacenter,

and other located in the Ireland datacenter. Both clusters accept read and write requests and are configured with bidirectional replication.

The new user registration process executed by the client application is represented by workloads that implement the IK and UUID Key patterns. Couchbase is the database used for the tests, since it provides bidirectional replication between clusters through its Cross Datacenter Replication (XDCR) feature. Since the objective is to compare the behaviour of the UUID Key and IK patterns, it is enough to create each database cluster with only one node. The USA datacenter is represented by an AWS EC2 instance in the Oregon region, while the Ireland datacenter is represented by an AWS EC2 instance in the Ireland region.

5.1.5.1 Consistency Test

In the first test, the two clusters are not able to synchronize their data due to a network partition between the American and the Irish datacenters. Regardless the network partition, the instances of the application in both datacenters keep accepting new users registrations. In each datacenter, the application accepts the registrations of 10000 new users. At the end, each cluster holds 10000 new documents that should sum 20000 documents after the reestablishment of communication between the two datacenters.

Regarded to the IK pattern, in order accept new users registrations during the network partition, each database cluster holds a master copy of the counter document. That is how the workload that implements the IK pattern was implemented. In each datacenter, an IK workload inserted 10000 new users documents in its local database and the key of each user document was obtained from the local counter document. The execution of the two workloads was simultaneous. After the execution of the workloads, the communication between the datacenters was restored and the clusters were able to synchronize their data.

The workloads that implement the UUID Key pattern were executed in the same scenario in which the IK workloads were. Each UUID Key pattern workload inserted 10000 new user documents in its local database, simultaneously. After the execution of the workloads, the communication between the datacenters was restored and the database clusters synchronized.

During the network partition, each IK workload was able to insert the 10000 new user documents in the local database cluster. However, after the reestablishment of the communication between the datacenters and synchronization of the clusters, only 10000 user documents left in the database, instead of 20000. During the network partition all the values generated by the

counters were duplicated, since they could not be synchronized. Consequently, all the user documents were stored with duplicated values for their primary keys and Couchbase handled the duplication by choosing a winner document and discarding the other one.

The result was different for the UUID Key workloads. After the reestablishment of the communication and synchronization of the database clusters, the user collection totaled 20000 documents. Since the UUID value is generated by the client application itself and without any database coordination, the network partition did not harmed the database consistency.

5.1.5.2 Availability Test

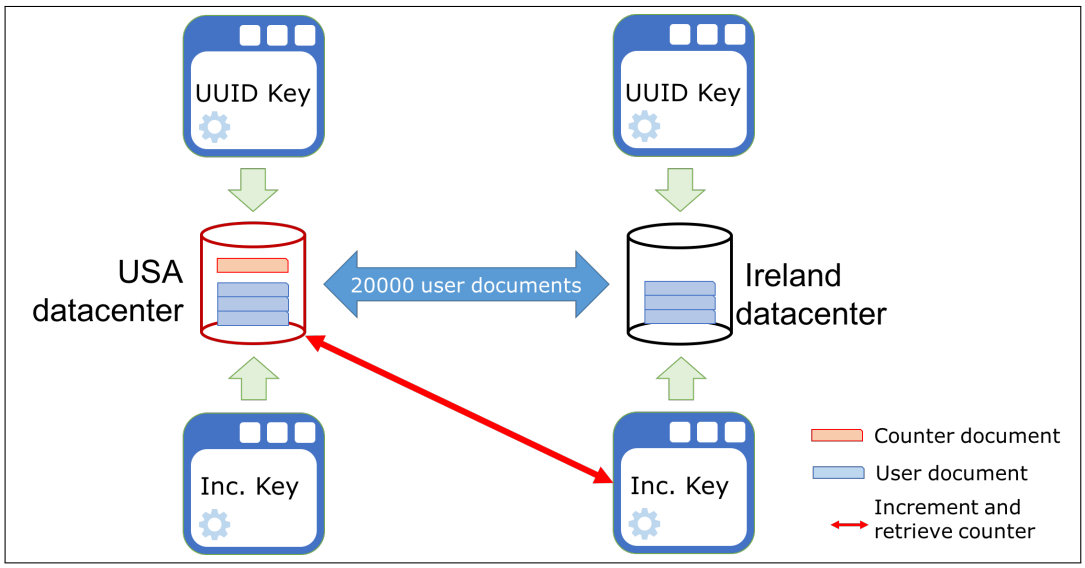
The second test simulates a great number of new users, in USA and Ireland, trying to register simultaneously. This time, there is no network partition and the database clusters are kept in sync all the time. In each datacenter the application receives 10000 user registration requests. Therefore, at the end of the test, each database cluster must hold 20000 user documents, since they are synchronized. Differently from the first test, only the counter document stored in the cluster located in the USA datacenter can be used with the IK pattern, as illustrated in Figure 24. That restriction is adopted in order to avoid inconsistencies in data.

For each workload (IK and UUID Key), two instances were executed at the same time, one in USA and other in Ireland, as shown in Figure 24. The IK workload executed in the Ireland datacenter had to request surrogate keys to the counter document stored in the USA datacenter. The workload instances were executed without any throughput restrictions. That simulates the client applications submitting as many requests as they can, until it reaches 10000 user registrations.

The next figures show the results for the second test. Figure 25 compares the throughput for both patterns. The UUID Key pattern throughput in both datacenters are about 40 percent greater than the IK pattern throughput in the USA datacenter. That difference is acceptable, since the IK pattern has to submit an increment and get request to the database before saving a user record. However, the IK pattern throughput in the Ireland datacenter is more than 10 times less than the UUID Key throughput due to the latency between the USA and Ireland datacenters.

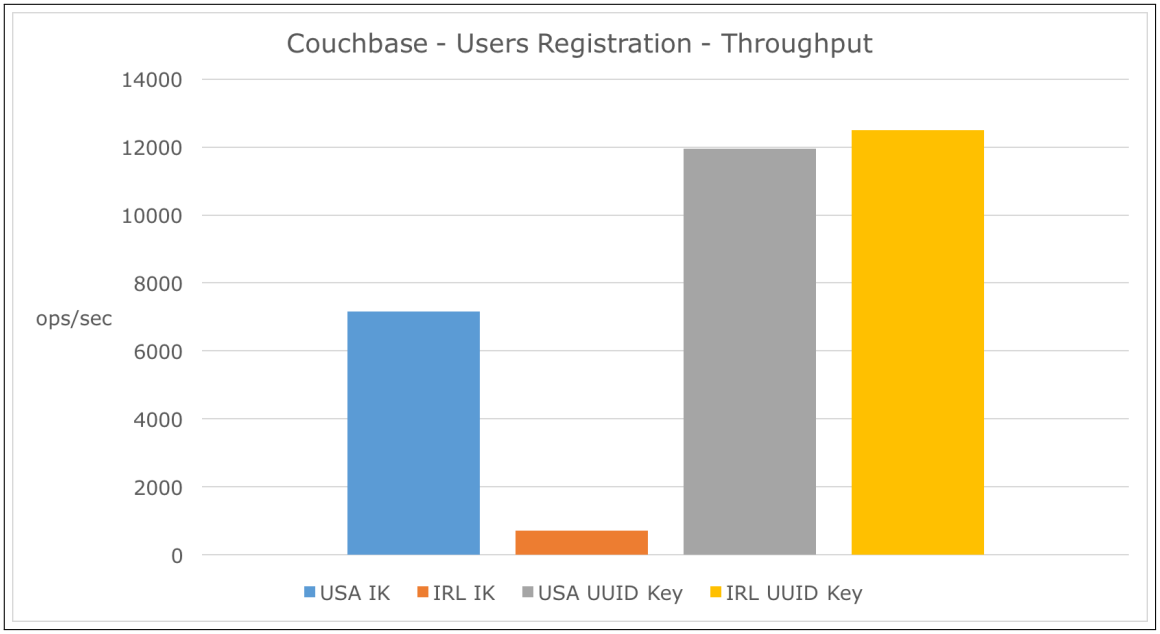
Figure 26 compares the latency for the IK and UUID Key patterns. The IK workload instance executed in Ireland presents a prohibitive average latency value: almost 140 milliseconds. On the other hand, both instances of the UUID Key workload present satisfactory values for the

Figure 24 – Two instances execute simultaneously: one in USA and another in Ireland.



Source: Created by the author

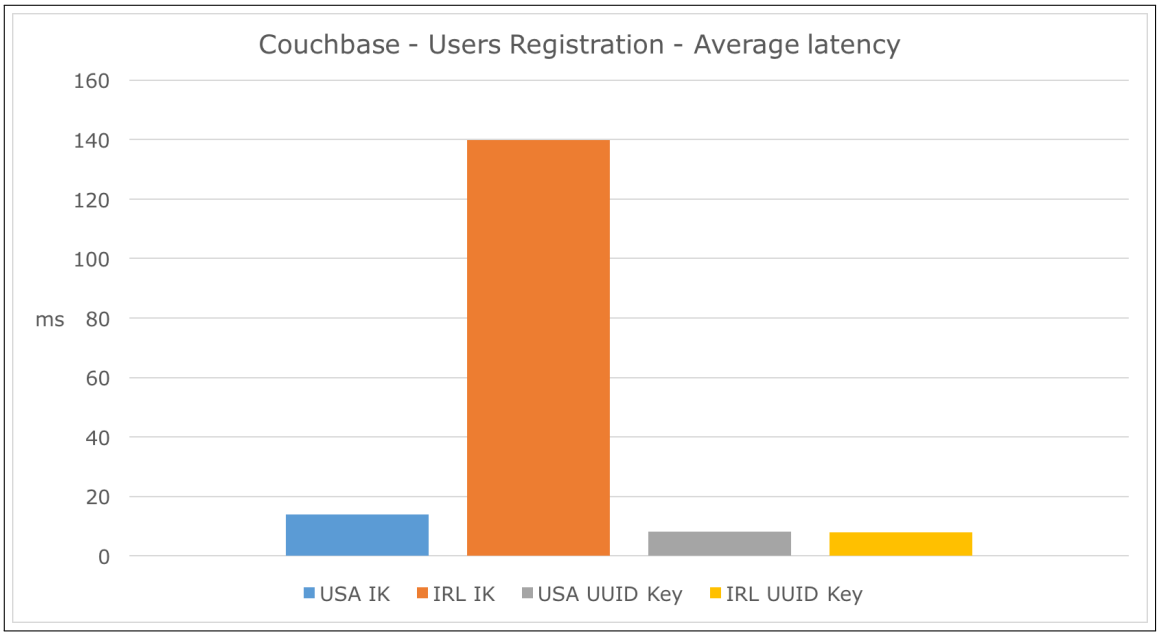
Figure 25 – Comparison between the Incrementing Key and UUID Key workloads throughput.



Source: Created by the author

average latency: less than 10 milliseconds, which is more than 14 times less than the Ireland datacenter IK workload. The IK workload instance executed in the Irish datacenter was penalized by the great latency imposed by the distance between the datacenters.

Figure 26 – Comparison between the Incrementing Key and UUID Key workloads average latency.



Source: Created by the author

5.1.6 Sidebars

In order to use the IK pattern, the stored data must be transparent to the database, so it can atomically increment the counter attribute inside a record, or document. In pure key-value datastores, the persisted data is opaque to the database, which means that as the database does not understand the content of the record, so it is not possible to increment the counter in a single atomic update.

The Cassandra database has a very efficient protocol for synchronizing the value of distributed counter records (SHARMA, 2014). Cassandra distributed counters can be incremented even during network partitions. After the network restoration, all the counters will be synchronized with the correct total value of increment requests submitted to all nodes during the network partition. However, they cannot be used as surrogate key generators, since Cassandra does not provide an atomic operation for incrementing and retrieving a counter value. Additionally, as it takes time to synchronize the counter records across all nodes, stale values could be used as record keys. That is similar to the use of stale values resulting from the big latency between distant servers when applying the IK pattern (Figure 21).

5.1.7 Consequences

The UUID Key pattern delivers better scalability than approaches that rely on central coordinators since it decentralizes the generation of surrogate keys by moving that responsibility to the database clients. Therefore, the UUID pattern takes better advantage of cluster expansions and avoids the creation of SPOFs.

The UUID Key pattern improves database availability, since it is not necessary to stop generating surrogate keys during network partitions, nor request surrogate key values to a potentially distant node, in order to keep database consistency.

With the UUID Key pattern, the responsibility of generating surrogate primary keys is shifted to the application layer. Therefore, the UUID Key pattern is more appropriate for pure key-value NoSQL datastores than the Incrementing key pattern since the last one requires that the content of the aggregate be transparent.

UUIDs are composed by 128 bits, consequently, more disk space will be required than by the IK pattern. However, nowadays, disk space is cheap and loosing clients interest is much more expensive for most business. On the other hand, the range of surrogate key values that can be generated using UUIDs, 2^{128}, is much larger than the range that can be generated from an integer counter, usually about 2^{64}. That is an important matter when applications have to handle great volumes of data.

If the secrecy of the data is important, the UUID Key pattern helps to preserve data integrity since it does not give any information about the number of records in the database as the IK pattern might. When the UUID Key pattern is adopted in order to generate record primary keys, it is not possible to submit range queries based on those keys, nor sort the result of queries by the primary key. The generation of UUIDs is random and does not follow any order. On the other hand, the IK pattern automatically allows sorting by insertion-order and range queries.

The UUIDs are not human friendly, users of an application that adopts the UUID Key pattern cannot use records primary keys as codes. On the other hand, that is possibile when the IK pattern is used.

5.1.8 Related Patterns

The UUID Key pattern is related to the Incrementing Key pattern as both are used for the same purpose, and generally both are considered by developers when modeling an application

domain. When the database is not geographically dispersed, and all the nodes are connected by a low latency network, both patterns can be adopted.

5.2 INDEX TABLE PATTERN

5.2.1 Context

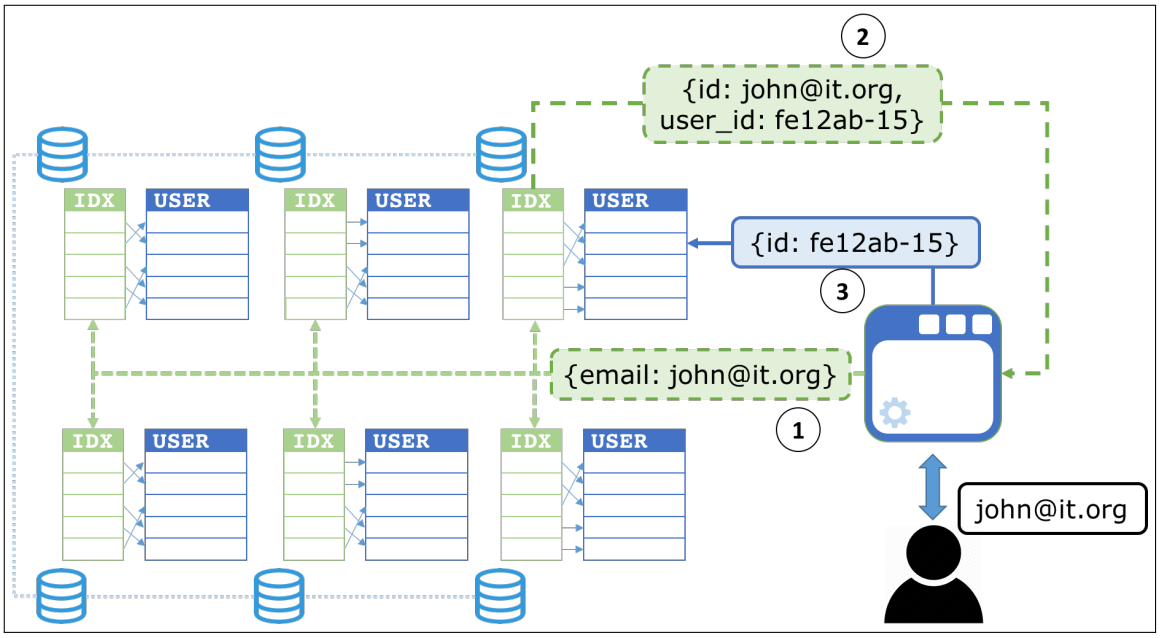
Many applications implement equality queries that search a record using an attribute different from its primary key. For example, a user record can be queried using its email attribute as search key instead of its primary key value. Usually, those kind of queries are implemented with secondary indexes, specially with relational databases. However, the use of secondary indexes in NoSQL databases is not as straightforward and standardized as in relational databases.

In order to illustrate scenarios in which secondary index based queries are used, let us suppose a web application that adopts the UUID Key pattern in order to identify its users records. The UUID Key pattern provides the benefits previously described in the Consequences section of the UUID Key pattern section (5.1.7), but is not practical for the users to supply the UUID of their records in order to authenticate in the application, since UUIDs are not human-friendly. Therefore, the application allows its users to authenticate with their emails, which are stored as an attribute of the user record.

As illustrated in Figure 27, in order to locate the user record without executing a full table scan, a secondary index is created on the email attribute. Since the secondary index is local to each shard, the index based query (1) is scattered across the whole cluster. Once the record address is fetched (2), a request is targeted to the node in which the record is stored (3). For each additional attribute of the user record that can be used by the users to authenticate, an additional secondary index must be created.

In the described scenario, when a user asks for authentication, the database processes an equality query. The user email attribute is a high cardinality key, since it is unique in the collection, or it hardly repeats. Additionally, email values tend to be quite different, they do not differentiate from each other monotonically. Based on those characteristics, the appropriate type of secondary index for that kind of scenario is a hash index. However, as previously mentioned, secondary indexes are not homogeneously available in NoSQL databases as they are in relational databases. Some NoSQL databases implement only range secondary indexes, which are not the appropriate type of index for queries based on high cardinality attributes.

Figure 27 – Secondary index on email attribute allows user to authenticate without using its UUID.



Source: Created by the author

An issue even more relevant than the fact that the appropriate index type is not available is the fact that most NoSQL databases implement only local secondary indexes, which are not scalable. Considering the scenario described above, even when a NoSQL database provides hash secondary indexes, the system will not scale if the index is local. If the cluster has only a few nodes, a local secondary hash index may perform well, but if the cluster grows, scalability problems begin to arise.

For NoSQL databases that do not provide global secondary indexes nor secondary hash indexes, implementing local range indexes to support equality queries are not the appropriate solution. Often, NoSQL databases are adopted due to their capacity of easily scale in order to handle big volumes of data and thousands of concurrent users. Even if a NoSQL database provides secondary hash indexes, local indexes will perform badly in those kind of environments. Another approach must be adopted in order to leverage database scalability.

5.2.2 Problem

How to enable sharded NoSQL database clusters that do not provide global secondary hash indexes to respond to equality queries based on high cardinality attributes that are not primary keys, without compromising scalability?

5.2.3 Forces

- Most NoSQL databases do not provide global secondary indexes, therefore, secondary indexes are local to each shard. Consequently, equality queries whose search key is not a primary key are scattered among all the nodes, impairing database scalability.
- Some NoSQL databases do not provide secondary hash indexes, only range indexes. Consequently, when secondary indexes are used to perform queries based on high cardinality attributes, it becomes more difficult to achieve satisfactory performance.
- Not sharding the secondary index would avoid scattering the requests across all the nodes in the cluster. However, centralizing index data in a single node would create a SPOF and would not scale automatically. Therefore, that approach would require manual management in order to properly shard index data, which may not worth it for simple lookup queries.

5.2.4 Solution

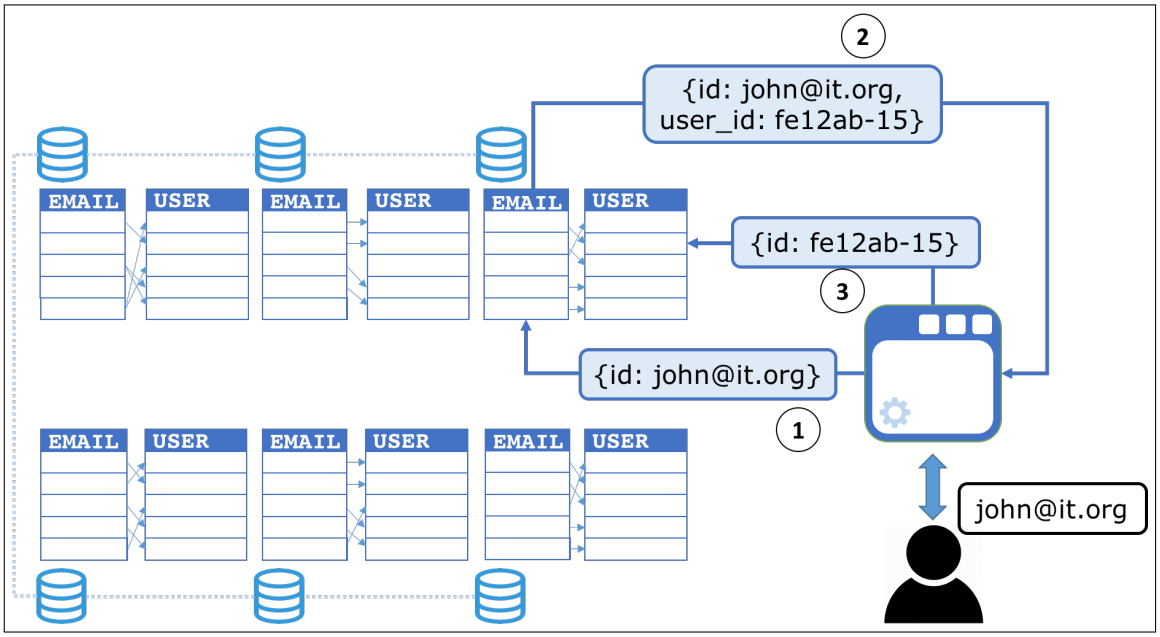
Almost every NoSQL database that supports data sharding by hash partitioning provides a key-value interface. Key-value based requests are fast and scalable since only the shard that stores the searched record is hit. On the other hand, queries based on local secondary indexes must hit all the shards in order to find a record. Therefore, in order to quickly find a record based on an attribute rather than its shard key and leverage database scalability, the search key attribute must be stored as an independent record, which means it must be denormalized.

The solution consists on the creation of an additional lookup record for each attribute that can be used as a search key. The attribute that acts like the search key is the primary key of the lookup record, and the second attribute of the lookup record holds the primary key of the primary record, from which the lookup record was derived. In order to find the primary record based in a search key rather than the record's primary key, the lookup record is fetched with a simple key-value request. Then, a second request is submitted in order to get the primary record. As that approach simulates a global secondary hash index, the pattern is known as Index Table.

Figure 28 illustrates the use of the Index Table pattern in the scenario described in the Context section (5.2.1). The user provides his/her email to the web application and a key-value request that uses the user email as search key is sent to the database (1). Since the user email is the primary key of the lookup record, only the shard that stores the searched record is hit. Once the lookup record is retrieved by the application (2), its second attribute, the user UUID, is used

to get the user record (3) in order to execute the authentication process. For each attribute of the primary record that must be used as search key, an additional lookup record must be created.

Figure 28 – Lookup record is used in order to find the user record.



Source: Created by the author

5.2.5 Pattern Tests

The example described in the previous sections is used again in this section as the scenario for the tests that compare local secondary indexes with the Index Table pattern. As previously described, the example application adopts the UUID pattern in order to identify its users records. Since UUIDs are not human-friendly, the users are allowed to authenticate providing their emails.

The following tests are composed of two phases: the first phase simulates users registrations, and the second simulates users authentications. The objective of the Index Table pattern is to leverage database read scalability, and that is verified in the second phase of the tests. However, it is important to verify how the Index Table pattern can impact database write scalability, and that is the objective of the first phase of the tests.

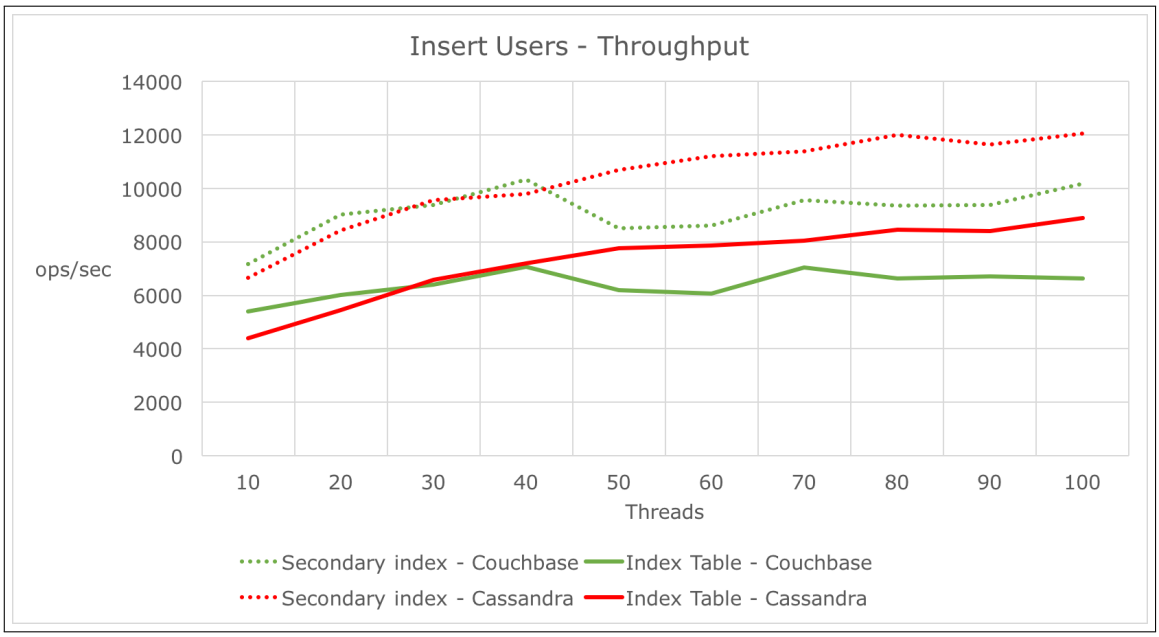
5.2.5.1 Local Range Index and Index Table Pattern

The objective of the first test is to compare the scalability between the secondary index approach and the Index Table pattern in databases that do not provide hash indexes. Therefore, Couchbase and Cassandra are used in this test because they provide only range indexes. Two workloads have been implemented for this test: the Secondary Index workload, and the Index Table workload. The workloads will be executed in both databases, and each database cluster is composed of 4 nodes.

In the first phase of the first test, each time an workload is executed it inserts 100000 user records in the database. Each workload executes 10 times, and for each execution the number of client threads is incremented by 10. The first execution starts with 10 threads and the last finishes with 100 threads. The database is cleaned between the executions of a workload.

Figure 29 shows that when inserting user records, for both databases, the throughput of the secondary index based approach is higher than the throughput of the Index Table pattern. The difference is not substantial, 4000 user registrations per second at most, and that result is expected since the Index Table pattern workload performs two requests in order to register one user.

Figure 29 – Secondary Index and Index Table pattern insertion throughputs.

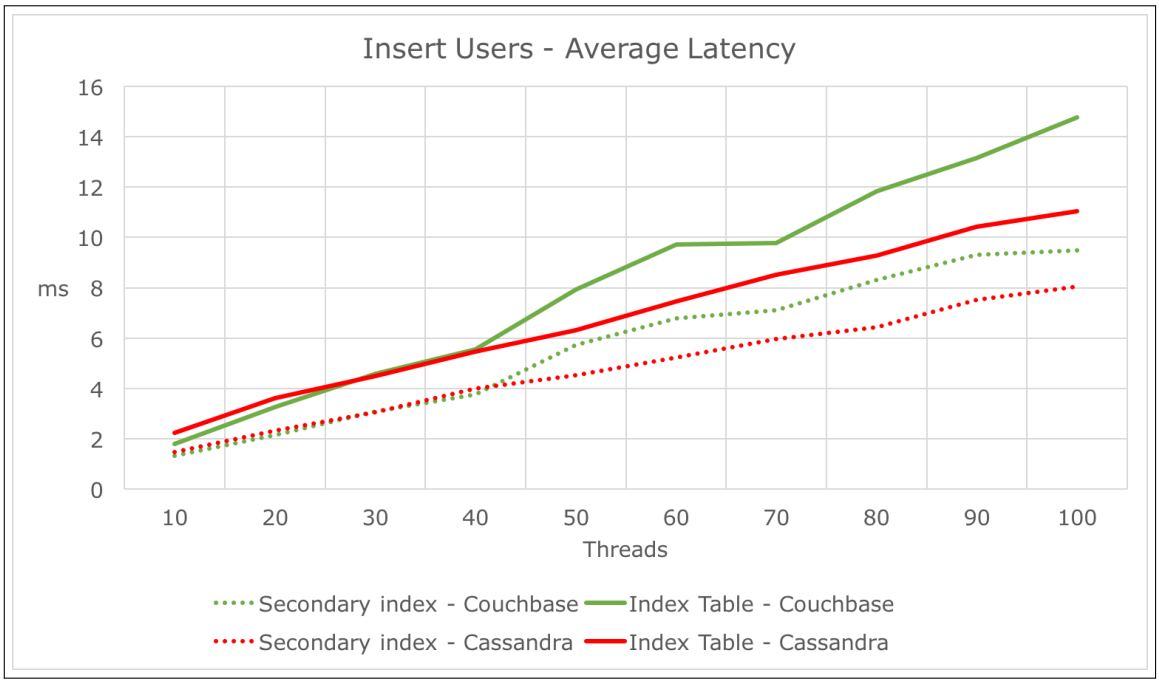


Source: Created by the author

Figure 30 also demonstrates expected results. The average latency, for both databases,

is lower for the secondary index based workload. With Couchbase, at 100 threads the average latency for the Index Table pattern is almost 50 percent greater than the average latency for the Secondary Index approach. For Cassandra the chart demonstrates a similar behaviour. The superiority presented by the secondary index based workload does not mean that it is better than the Index Table pattern. Figures 29 and 30 demonstrate that for writing data, the Index Table provides an acceptable lower scalability. The results demonstrate that the extra overhead added by the additional operation performed by the Index Table pattern workload does not compromise database performance.

Figure 30 – Secondary Index and Index Table pattern insertion average latencies.



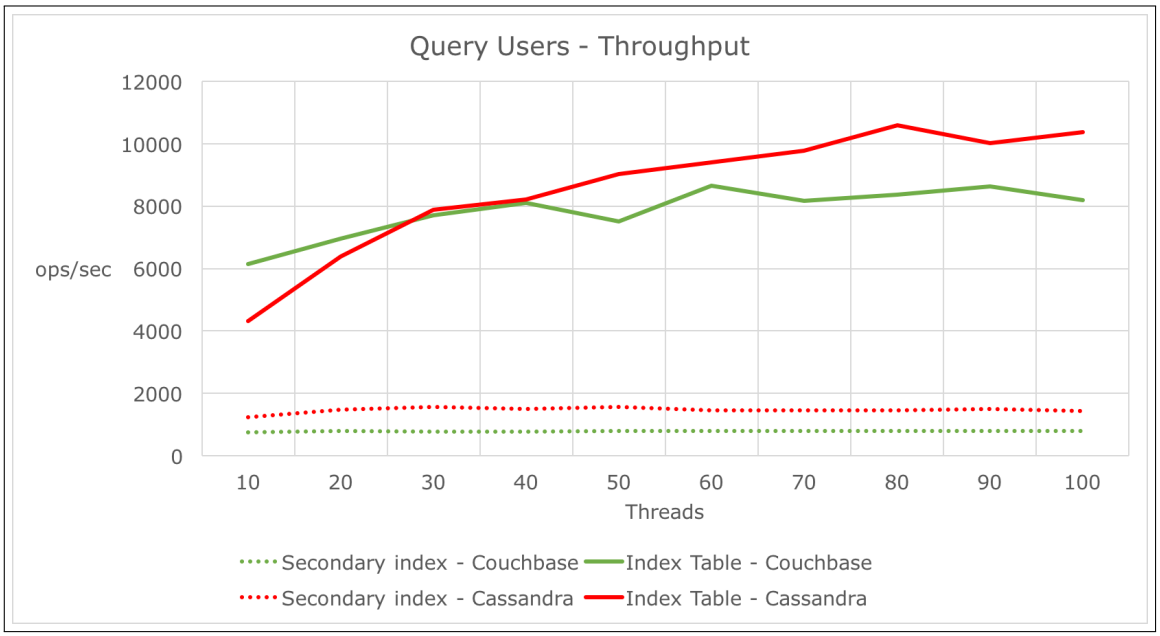
Source: Created by the author

As in the first phase, in the second phase of the first test, each workload also executes 10 times, incrementing the number of threads by 10 after each execution. The first execution starts with 10 threads, and the last finishes with 100 threads. However, this time each workload execution submits 100000 read request to the database, simulating users authentication.

Figure 31 demonstrates that the throughput provided by the Index Table pattern is substantially better than the throughput provided by a local secondary index when querying the user records. For both databases, the throughput of the Index Table pattern workload increased when the number of client threads increased. For Cassandra, the throughput increased more than 100 percent from 10 to 100 threads. And for Couchbase, the throughput increased about

33 percent. On the other hand, for the secondary index based workload, the throughput did not change, remaining considerably inferior to the Index Table pattern throughput.

Figure 31 – Secondary Index and Index Table pattern throughputs for user queries.



Source: Created by the author

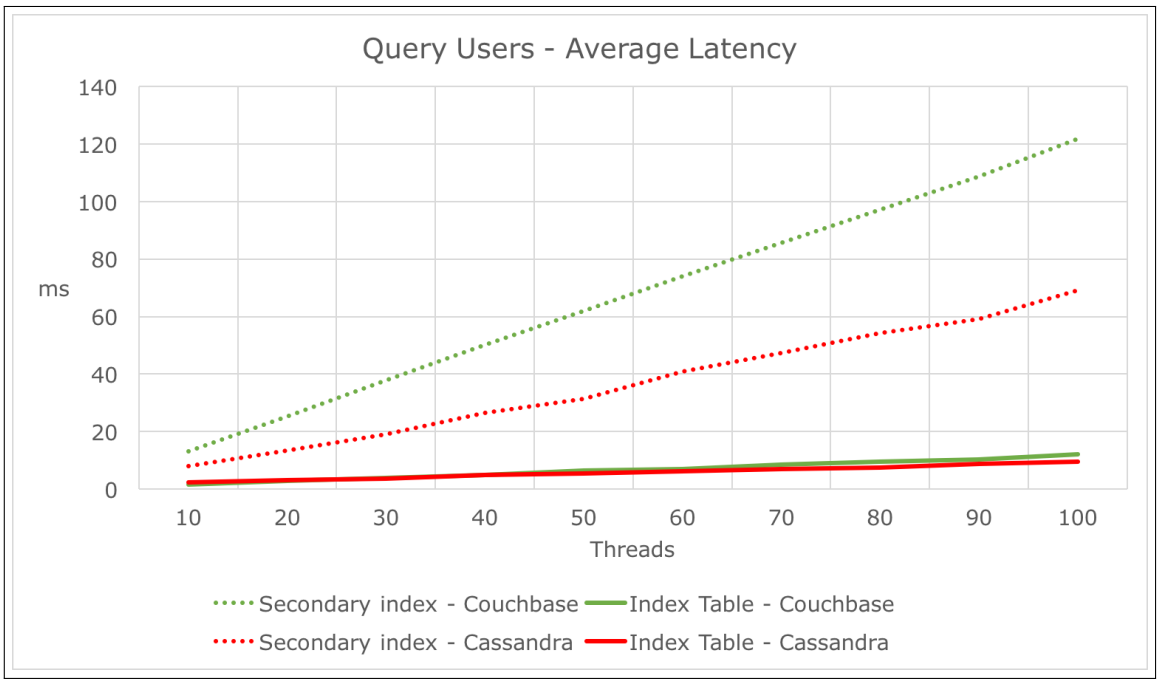
Figure 32 shows that with the increase in the number of client threads, the latency presented by the secondary index approach increased much more sharply than the latency presented by the Index Table pattern. With Cassandra, the Secondary Index workload average latency becomes about 7 times bigger considering the threads range. With Couchbase, the result was worse, the average latency becomes about 10 times bigger. That demonstrates that the Index Table pattern scales better than local range secondary indexes regarding data reading.

5.2.5.2 Local Hash Index and Index Table Pattern

MongoDB is used for the second test because it provides a hash secondary index, which is the appropriate type of index for equality queries based on high cardinality attributes. Still, it is not a global secondary index solution, which means that it is not scalable. Therefore, the second test compares the behaviour of the local secondary hash index approach and the Index Table pattern when the number of nodes in the cluster is increased, in order to verify how they take advantage of the greater capacity of the database.

The second test is similar to the first one, except that this time, instead of executing the workloads in two different databases, they are executed in two MongoDB clusters with

Figure 32 – Secondary Index and Index Table pattern average latencies for user queries.



Source: Created by the author

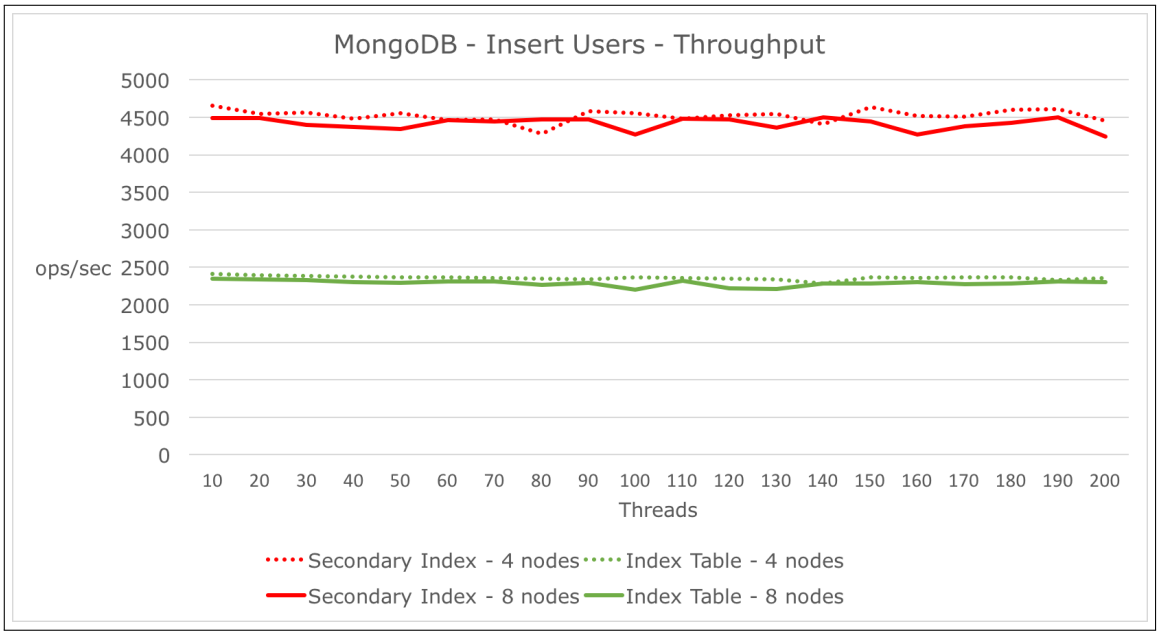
different number of nodes. Each phase is executed intially in a cluster with 4 nodes, then, in a cluster with 8 nodes. The increase in the number of nodes will demonstrate how the workloads take advantage of the cluster expansion. As explained in section 4.3, in order to reduce the effect of cloud performance fluctuation in the following tests, private EC2 instances were used to compose both MongoDB clusters used in the next tests.

The first phase consists on the registration of 100000 users. For each workload execution the number of threads is increased by 10, starting from 10 up to 200. Figure 33 demonstrates that when inserting new user records, due to the additional operation of inserting the lookup document, the throughput of the Index Table workload is 50 percent smaller than the throughput presented by the secondary index workload. However, for both approaches, the database throughput has remained stable across all the client threads range.

Figure 34 shows a similar behaviour. Across all the client threads range, the Index Table workload has presented an average latency value 100 percent greater than the secondary index workload. That behaviour is expected since the Index Table workload executes one additional operation for each user insertion. For both workloads the average latency has presented a linear growth.

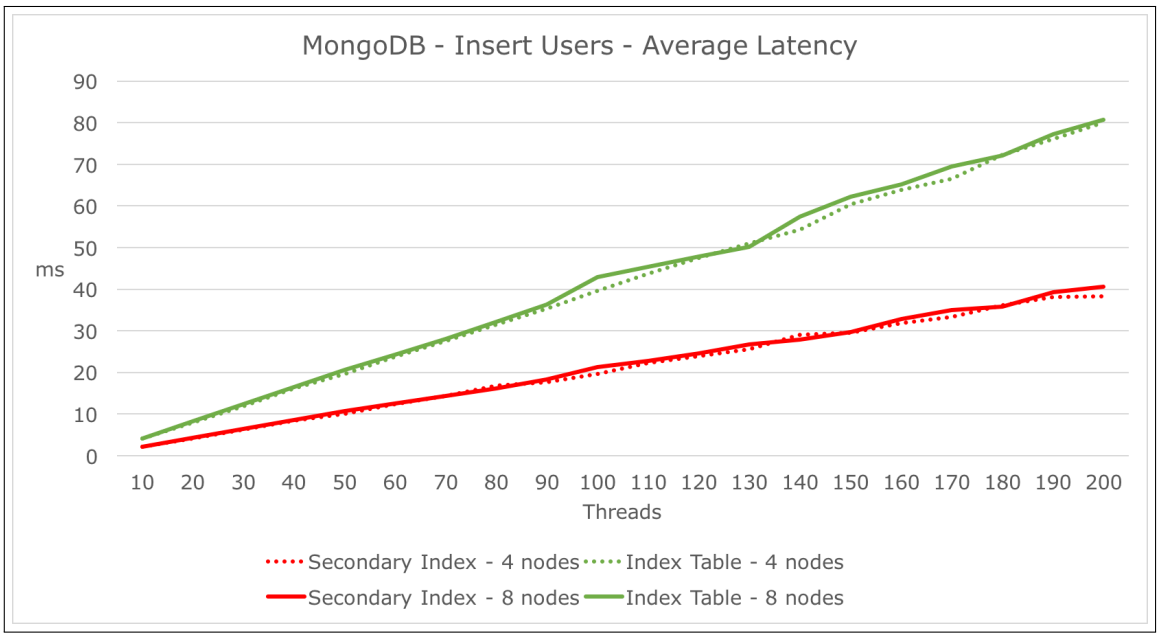
The second phase of the second test consists on the authentication of users, when 100000 user query requests are submitted to the database. For each workload execution the

Figure 33 – Secondary Index and Index Table throughputs with 4 and 8 nodes for user insertions in MongoDB.



Source: Created by the author

Figure 34 – Secondary Index and Index Table average latencies with 4 and 8 nodes for user insertions in MongoDB.

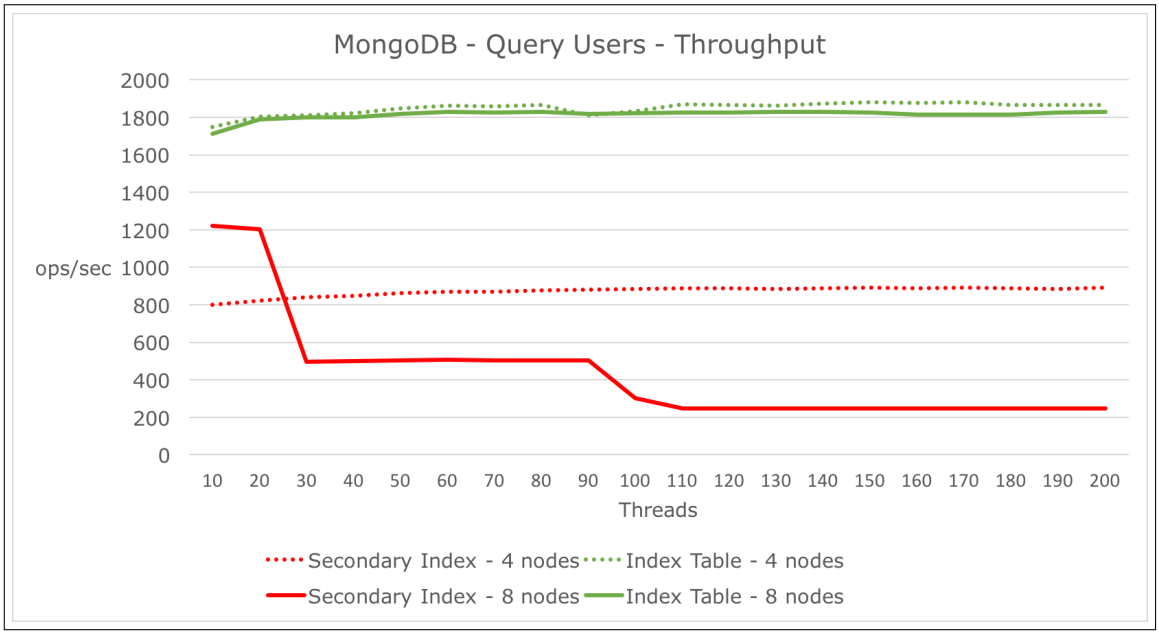


Source: Created by the author

number of threads is also increased by 10, starting from 10 up to 200. Regarded the query of user records, the secondary index and the Index Table pattern approaches present different behaviours. Figure 35 shows that, for the Index Table pattern, the cluster throughput has remained stable

after the addition of the four nodes. While the database throughput for the secondary hash index has reduced aggressively. For the 8-nodes cluster, starting from 120 threads, the throughput of the Index Table pattern is almost 9 times greater than the throughput of the secondary hash index approach.

Figure 35 – Secondary Index and Index Table throughputs with 4 and 8 nodes for user queries in MongoDB.



Source: Created by the author

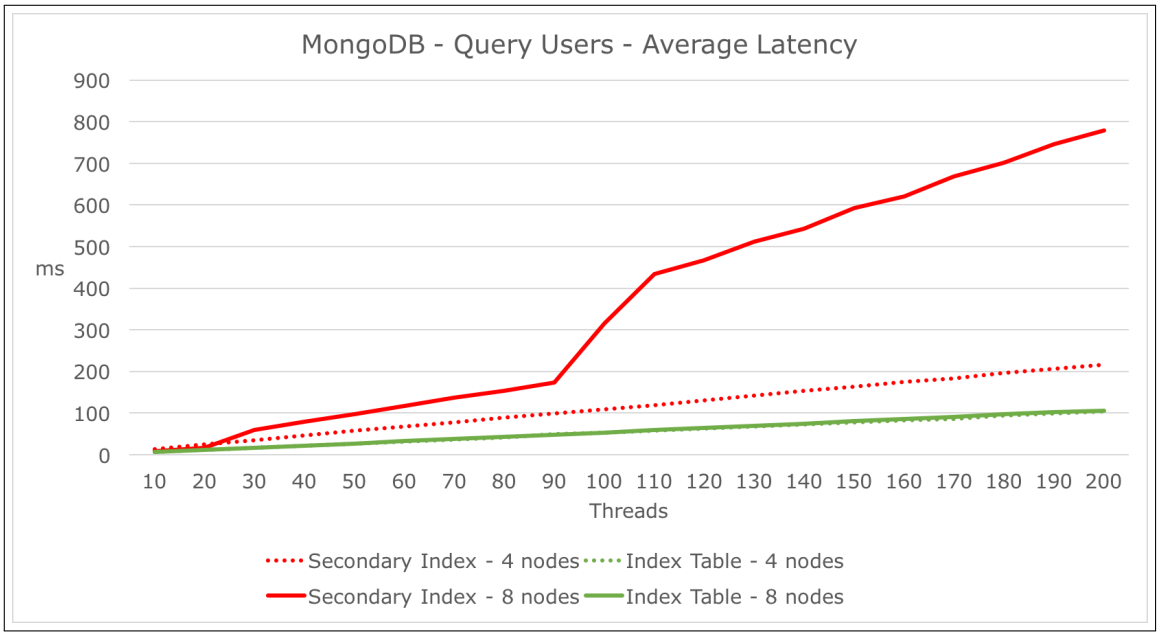
Figure 36 shows that when using the secondary hash index, the average latency increased sharply when the cluster size increased. On the other hand, the Index Table pattern average latency has remained stable. For the 8 nodes cluster, at 200 threads, the average latency of the secondary index approach is almost 9 times greater than the average latency of the Index Table pattern.

5.2.6 Sidebars

Couchbase provides range (B-tree) indexes as Global Secondary Indexes (GSI). As the name suggests, in Couchbase, secondary indexes can be global indexes in order to avoid the network communication overhead generated by local secondary indexes. An exclusive node can be configured to execute the index service. As consequence, all queries based on GSIs are targeted to that single node generating a bottleneck and a SPOF.

In order to scale out the index service, replicas must be deployed. However, unlike

Figure 36 – Secondary Index and Index Table average lantencies with 4 and 8 nodes for user queries in MongoDB.



Source: Created by the author

the storage engine, GSIs do not provide automatic built-in replicas, demanding more knowledge and manual management. The sharding of secondary indexes is possible, but it also demands manual management. The adoption of that solution increases the necessity of a specialist, while the NoSQL movement suggests less dependency on Database Administrators (DBA) and makes the system more dependent on specific features of the datastore.

5.2.7 Consequences

The Index Table pattern improves the system performance for databases that do not provide secondary hash indexes since it reuses the hash partition feature of the database. The set of lookup records related to an attribute of a collection works as a manual global hash index, which are appropriate for distributed databases and high cardinality attributes.

The Index Table pattern improves the scalability of the system for NoSQL databases that do not provide global secondary hash indexes. As demonstrated by the test executed with MongoDB, even using a hash secondary index, the scatter-gather operation executed by the database in order to find a record based on a local index does not provide scalability since it increases latency and decreases throughput as the number of shards of the cluster grows.

Some people may think that the lookup documents require additional disk and

memory space that would not be required when using secondary indexes, but indexes also require additional storage space. Nonetheless, nowadays, disk space is affordable.

The Index Table pattern transfer to the application the responsibility of keeping the consistency between the primary record and its lookup documents. If an attribute of a primary record is updated and there is a lookup document that uses the value of the updated attribute as primary key, the lookup record must be deleted and a new one must be created, since the primary key of a record cannot be changed. On the other hand, if a secondary index is used, the database automatically update the index.

Generally, a secondary index does not have to be an unique index too. If the indexed attribute does not need to be unique in the collection, a secondary index can be used in order to optimize queries that will fetch more than one record that satisfy the search criteria. The Index Table pattern does not allow more than one lookup record with the same value for the shard key since the shard key is also the record's primary key.

5.2.8 Related Patterns

The Index Table pattern present a connection with the Enumerable Keys pattern (5.3) pattern since it reuses the hash partitioning feature of sharded NoSQL databases in order to optimize queries and leverage database scalability.

5.3 ENUMERABLE KEYS PATTERN

5.3.1 Context

CF NoSQL databases, such as Cassandra and DynamoDB, allow a primary key to be composed by two attributes: the partition key, or shard key, and the sort key. The partition key acts like a clustering field (ELMASRI; NAVATHE, 2010) allowing the records that share the same value for the partition key to be stored in the same shard. Records with the same shard key value are sorted by the sort key.

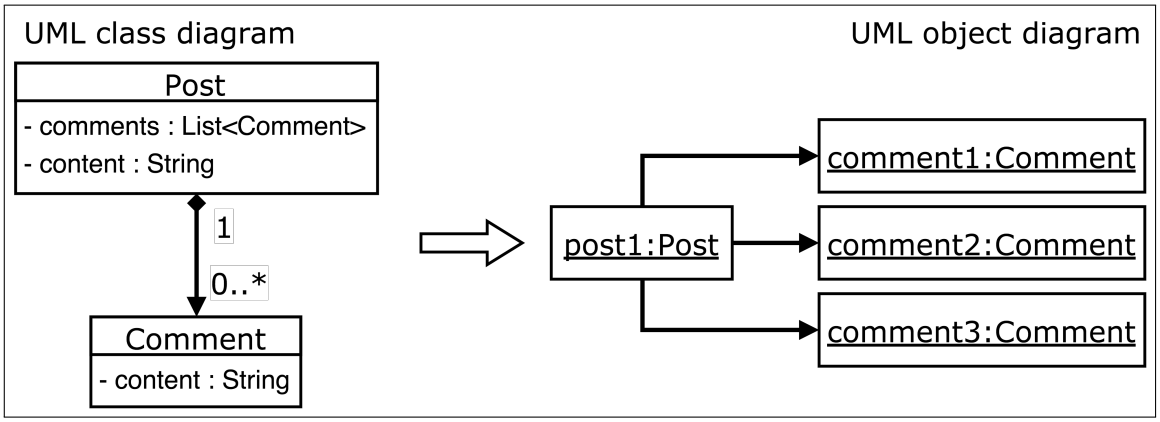
Many NoSQL databases, mostly KV and DO, do not implement compound primary keys that allow clustering and sorting records, at least when using hashed sharding[1] (COSTA et al., 2015). The standard approach to model related documents in DO databases is using

[1] MongoDB implements the clustering and sorting of records with compound primary keys when the Ranged Sharding is used.

nested documents. Related documents can be stored in a list, which in turn is an attribute of a document that contains the attributes whose documents inside the list share the same values. In some DO databases, such as Couchbase and MongoDB, the documents inside the list are sorted by the insertion order. Although very simple and intuitive, that approach is not scalable. The example presented in Section 2.1.5.2 of Chapter 2 demonstrates that pattern employed in order to model an one-to-many association. For the remainder of this text, that pattern will be referred as List-Based Association (LBA).

In order to illustrate the issues discussed in this section, an example of one-to-many association is used. Let us suppose a web blog application. For each post of a user, there can be an arbitrary number of comments posted by other users. Therefore, there is an one-to-many association between a post and its comments (Figure 37). As expected, when the comments of a post are displayed, they are paginated in descending chronological order, and each page displays ten comments.

Figure 37 – One-to-many association between posts and their comments.



Source: Created by the author

Usually, in DO databases, one-to-many associations are represented with the LBA pattern. Therefore the LBA pattern is used to model the association between posts and comments in the web blog example. The comments of a post are stored as documents in a list attribute of the document that represents the post. The database keeps the comment documents in the list sorted by the insertion order.

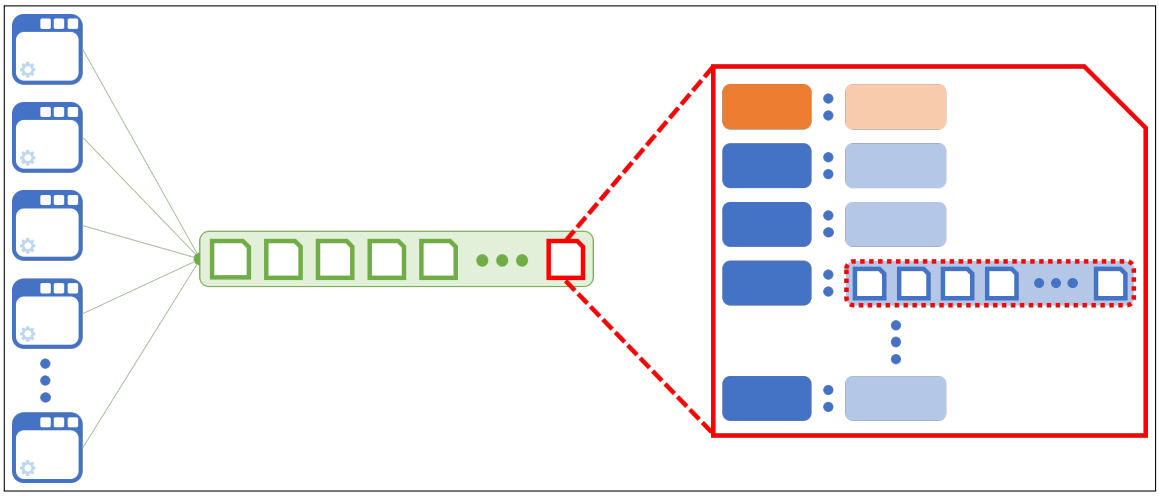
The lack of relationships among records is one of the fundamental concepts that makes possible the great scalability of NoSQL databases. In order to avoid queries that spread across the nodes of a cluster, related data should be stored as a single aggregate. That approach leverages database scalability regarding read requests. Therefore, the LBA pattern provides

read scalability and is the first alternative considered when modelling associations, specially by newcomers in NoSQL databases. Additionally, the LBA pattern presents great compatibility with the OO paradigm. However, despite the before mentioned benefits, the LBA pattern does not provide good scalability regarding write requests.

With the LBA pattern, all the documents at the many side of the association are nested inside the document at the one side of the association. Considering the web blog example, all the comments related to a post are nested inside the post document. When a user submits a new comment, the application pushes a new comment document into the comment list of the post document. Since operations over an aggregate are atomic, the post comment must be locked. If a post becomes extremely popular, a concurrency condition may arise. If hundreds, or even thousands of users, try to simultaneously submit a comment, a great queue of requests will be created. Each request will have to wait its turn to lock the post document and push a new comment document into the list of comment documents, or update a comment document inside the list.

Figure 38 illustrates that concurrency condition. Several web clients (application icons at the left side) simultaneously submit comments related to a popular post (large document outlined in red). The comment documents are stored in a list attribute of the post document (document list outlined in dotted red). A great queue of write requests (green documents queue at the center) is created, since each request has to lock the post document in order to update the comments list.

Figure 38 – The LBA pattern may generate big queues due to concurrency conditions.



Source: Created by the author

The concurrency condition generated by the LBA pattern does not happen with most CF databases due to their compound primary key feature. The records at the many side of the association are stored as independent rows in a table, as illustrated in Figure 9, of Chapter 2.

At last, but not least, one more issue must be considered: every database imposes a limit for the size of a record. Consequently, if a document has nested documents inside a list attribute, that list cannot grow indefinitely. Considering the web blog example, it means that the product of the average size of the comment documents of a post and the quantity of comments of that post must not exceed the maximum document size of the database.

For the remainder of this section, the documents at the many side of an one-to-many association will be referred as many-side documents, and the document at the one side will be referred as one-side document.

5.3.2 Problem

How to model one-to-many associations in hash sharded document-oriented NoSQL databases that do not implement clustering shard keys, without compromising write scalability?

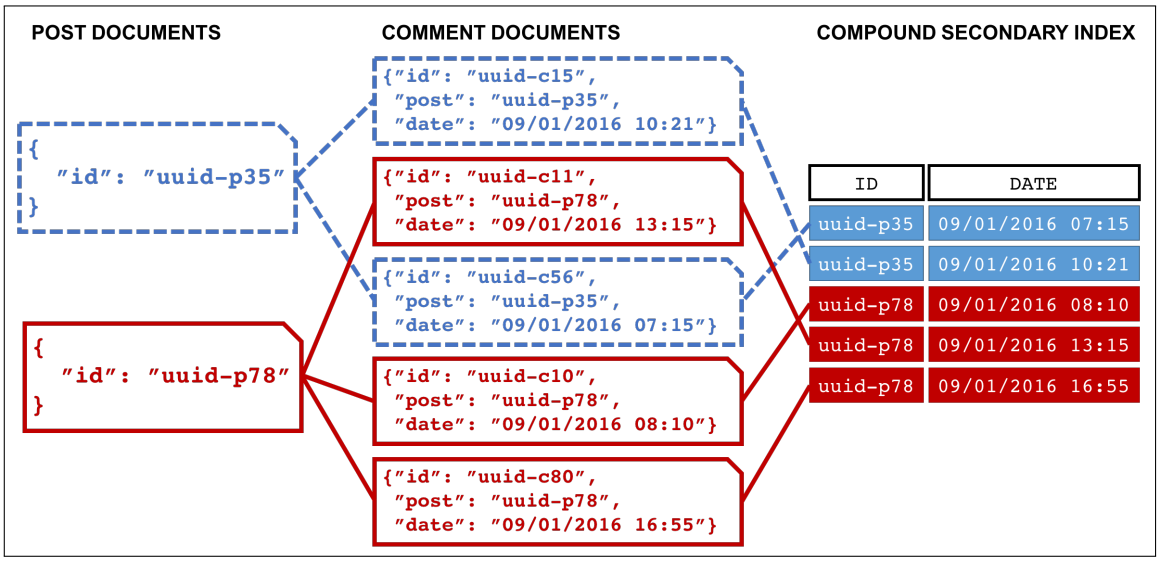
5.3.3 Forces

- Despite the simplicity and compatibility of the LBA approach with the OO paradigm, nesting associated documents in a list attribute of another document may cause performance and scalability degradation in concurrent environments.
- All databases have a limit for the record size. This is no different for NoSQL databases, despite their capacity to deal with great volumes of data. Consequently, if there is a limit for the document size, there is a practical limit for the size of a list nested inside a document.
- One-to-many associations can be represented in DO databases with a normalized approach by storing the many-side documents in their own collection and referencing the one-side documents by their ids. However, as demonstrated in Section 5.2, which describes the Index Table pattern, that approach provides low scalability because it must rely on a secondary index.

Using the web blog example, Figure 39 demonstrates the solution mentioned in the last item of the list above. A composed secondary index is associated to the collection that stores the comments and indexes two attributes: the key of the post a comment belongs to, and the

comment creation date. As can be seen at the right-hand side of the figure, the index records are chronologically sorted for a shared post UUID. Despite the low scalability, with those two attributes indexed it is possible to retrieve the comments that belong to a post, in chronological order.

Figure 39 – Posts and comments one-to-many association based on a secondary index.



Source: Created by the author

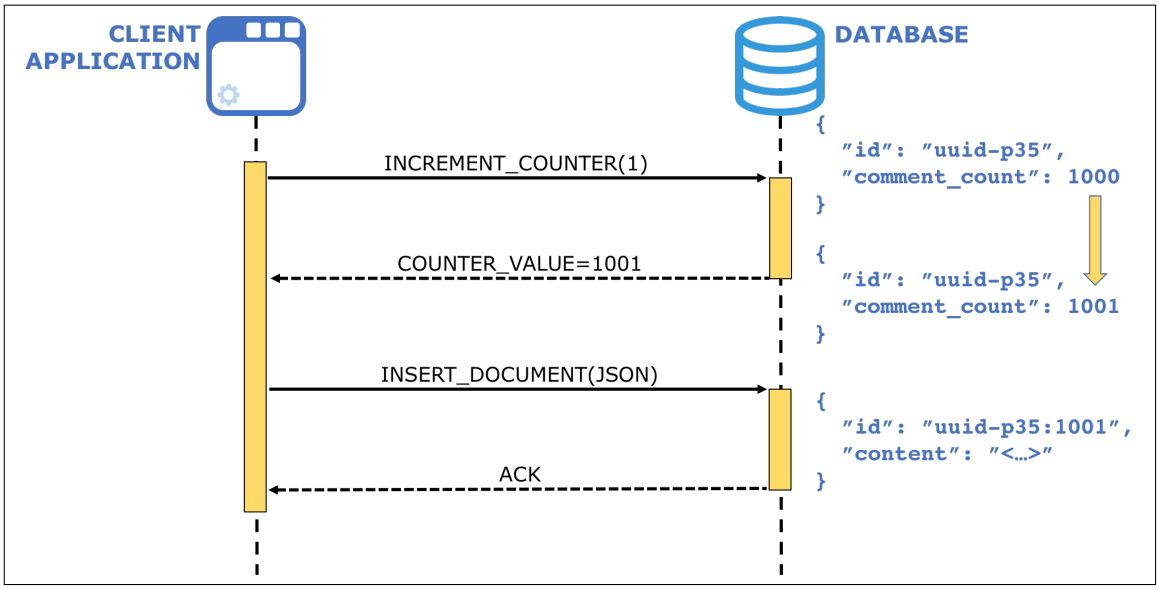
5.3.4 Solution

The pattern presented in this section, called Enumerable Keys, avoids the use of secondary indexes. The documents representing the entities involved in the association are stored in two different collections. A counter attribute must be added to the document representing the strong entity of the relationship, i.e., the one-side document. Each time a document representing a weak entity is added to the many side of the association, the counter attribute in the one-side document it belongs to must be incremented by one. The primary key value of each many-side document is composed by the primary key value of the one-side document they belong to, concatenated with the current value of the counter attribute of the one-side document. The counter attribute of the one-side document must be incremented before the storage of the many-side document, in order to compose the many-side document's key.

Figure 40 demonstrates the persistence of many-side documents for this pattern using the web blog example. Firstly, the client application requests the increment of the counter attribute in the post document (`comment_count`). The counter is incremented from 1000 to

1001 and its updated value is returned to the client application. Then, the client application submits a write request for storing a comment document that has as its primary key the post document primary key (`uuid-p35`) concatenated with the current value of the post document counter attribute (1001) .

Figure 40 – The primary key value is composed by the one-side document's id and the counter current value.



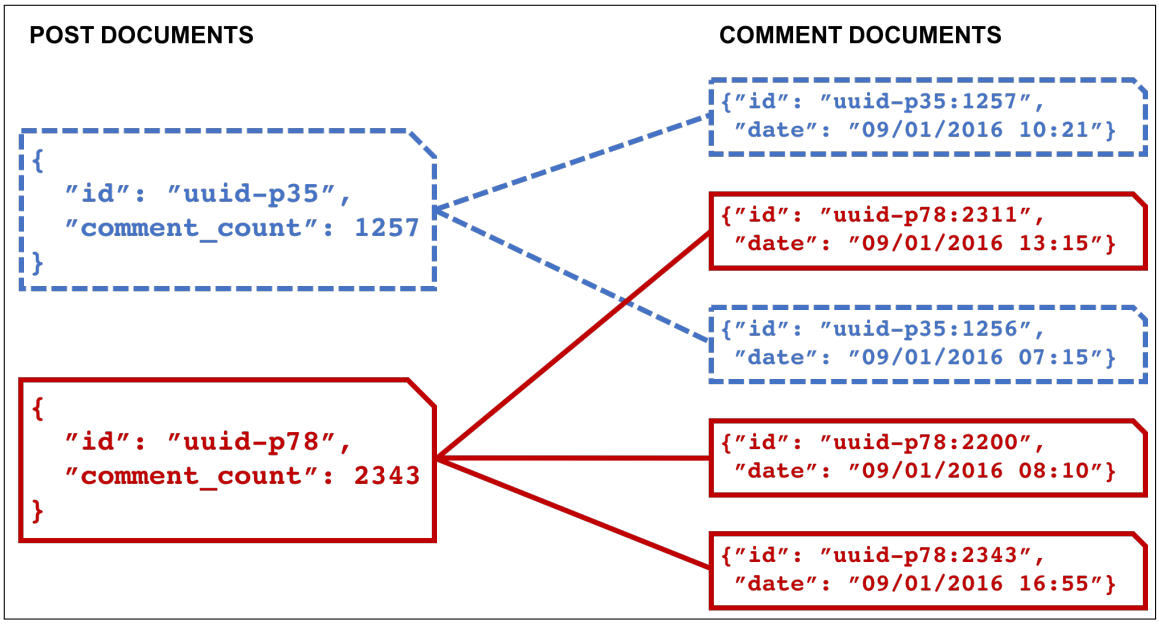
Source: Created by the author

Figure 41 shows how the post and comments shown in Figure 39 would be arranged in the database using the Enumerable Keys pattern. Each comment document has embedded in its primary key the primary key of the post document it belongs to.

It is possible to retrieve a contiguous set of many-side documents from the database because the position of each document, in the sequence of related documents, is embedded in the primary key. If the counter's current value is known, it is possible to traverse previous documents. It is also possible to traverse preceding and succeeding documents for any given many-side document primary key.

This pattern is indicated for associations with many documents. Generally, applications do not load with a single request all the documents at the many side of an one-to-many association with many documents involved. Usually, applications display the many-side documents through pagination. Each request retrieves a contiguous set of many-side documents from the database. For instance, the web blog application could display ten comments per page for an arbitrary post. To accomplish this effect without range secondary indexes, a bulk GET

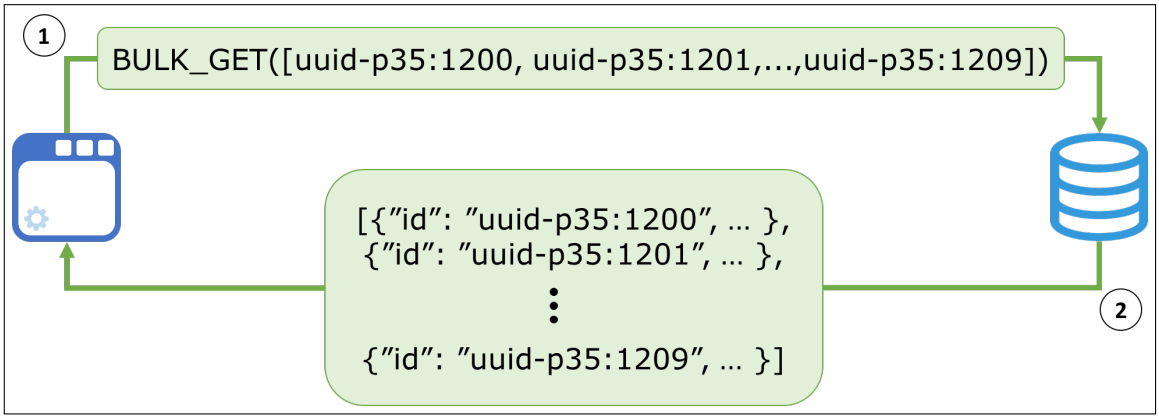
Figure 41 – The many-side document's id refers to the one-side document it belongs to and indicates its position in the list.



Source: Created by the author

operation must be used to retrieve the documents. Figure 42 illustrates a bulk get request. The client application submits a bulk read request that has as parameter a list that contains the id's of the desired documents. Then, the database returns the list of requested documents.

Figure 42 – A bulk GET is used to retrieve a contiguous set of many-side documents.



Source: Created by the author

5.3.5 Pattern Tests

In this section, the web blog example used in the previous sections is used as the scenario for the tests that compare the LBA and Enumerable Keys patterns. Couchbase and

MongoDB are used in the following tests because both are document-oriented databases and the LBA pattern is often used with those databases. Additionally, they do not provide clustering indexes.

For both databases (Couchbase and MongoDB), the following tests are executed in clusters composed of four nodes. The following tests compare the LBA and Enumerable Keys pattern about three concerns:

- write concurrency, in order to analyse how both patterns handle simultaneous write requests;
- data volume, in order to analyse how the data volume of one-to-many associations already stored in database influences the patterns;
- and data retrieval, in order to verify how both patterns behave about reading records of an one-to-many association.

5.3.5.1 Write Concurrency Tests

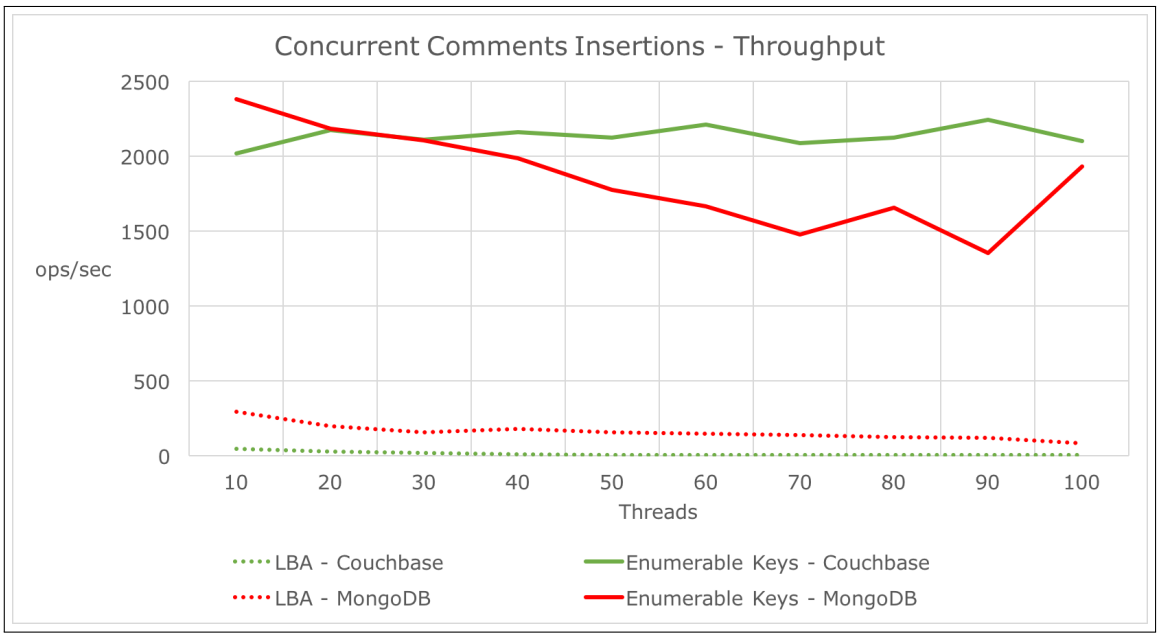
The objective of this test is to compare both patterns, LBA and Enumerable Keys, regarding their write scalability. It simulates many users simultaneously submitting comments related to a very popular post. For the two workloads implemented in this tests, one for each pattern, the client threads represent the users of the blog website submitting the comments.

Each workload executes 10 times and, for each execution, the number of threads is increased by 10, starting with 10 up to 100 threads. The increasing number of threads simulates the growth of the number of parallel users commenting the post. In each workload execution, 3000 requests to save comments are submitted to the database. The database is cleaned between the workload executions.

Figures 43 compares the throughput achieved by the LBA and Enumerable Keys patterns. For both databases, the Enumerable Keys workloads performed substantially better than the LBA workloads. Considering the most concurrent moment of the test, the execution with 100 threads, the Enumerable Keys pattern present a throughput more than 20 times greater that the LBA pattern. The Enumerable Keys workload executed against MongoDB has been affected by the cloud performance fluctuation, mostly between 50 and 90 threads. However, it is still noticeable the superiority of the Enumerable Keys pattern.

Figure 44 shows the comparison between the average latency of the LBA and Enumerable Keys patterns workloads. The LBA workloads present high latency values and it

Figure 43 – LBA and Enumerable Keys patterns throughput for concurrent comments insertions.



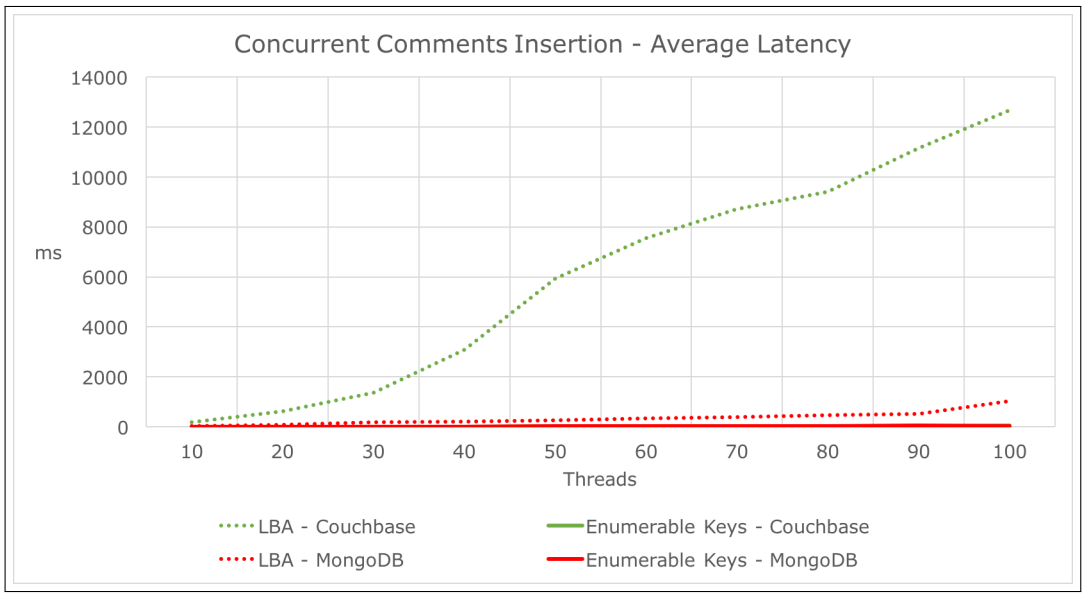
Source: Created by the author

becomes worse each time the number of parallel requests increases. On the other hand, the Enumerable Keys workloads maintain low values for the latency even with the increase of the number of client threads. With MongoDB, at 100 threads, the average latency for the LBA pattern is more than 1000 milliseconds and less than 50 milliseconds for the Enumerable Keys pattern. For Couchabase, the difference is much bigger. The Enumerable Keys pattern presented values similar to MongoDB, however, average latency for the LBA patter reached more than 12000 milliseconds. The average latency of the LBA approach has presented a much more agressive growth for Couchbase because the update has to be made at client side. On the other hand, MongoDB provides an append instruction that allows to update the list by submiting the additional value to the database, without retrieving the document.

5.3.5.2 Data Volume Test

The objective of the next test is to observe how the LBA and Enumerable Keys patterns perform with an increasing volume of data. As in the previous concurrency test, the next test executes each workload 10 times. But this time, a single thread requests the insertion of 1000 comments associated to the main post in each execution, and the database is not cleaned between the workload executions. After the first execution, each subsequent workload will perform with more 1000 comments associated to the post.

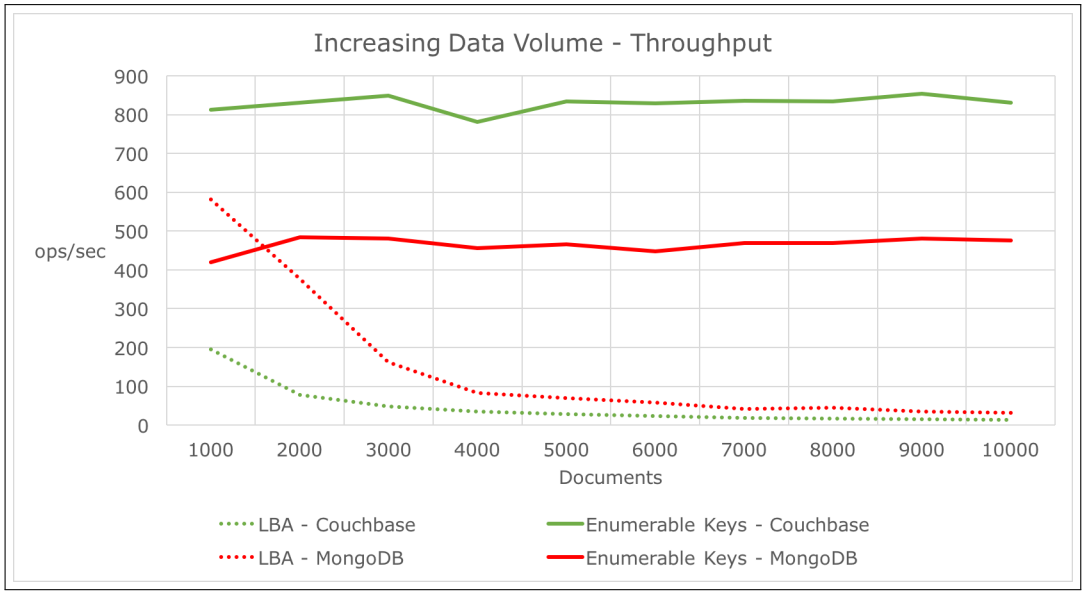
Figure 44 – LBA and Enumerable Keys patterns latency for concurrent comments insertions.



Source: Created by the author

Figure 45 compares the throughput for the LBA and Enumarable Keys patterns as data size increases. With the LBA pattern, the throughput gets worse as the number of comments stored in the list increases. For the Enumerable Keys pattern, the databases present a pretty stable throughput regardless the size of the comments collection. From 7000 records onwards, the Enumerable Keys workload presents a throughput about 60 times greater than the LBA workload in Couchbase, and about 15 times greater with MongoDB.

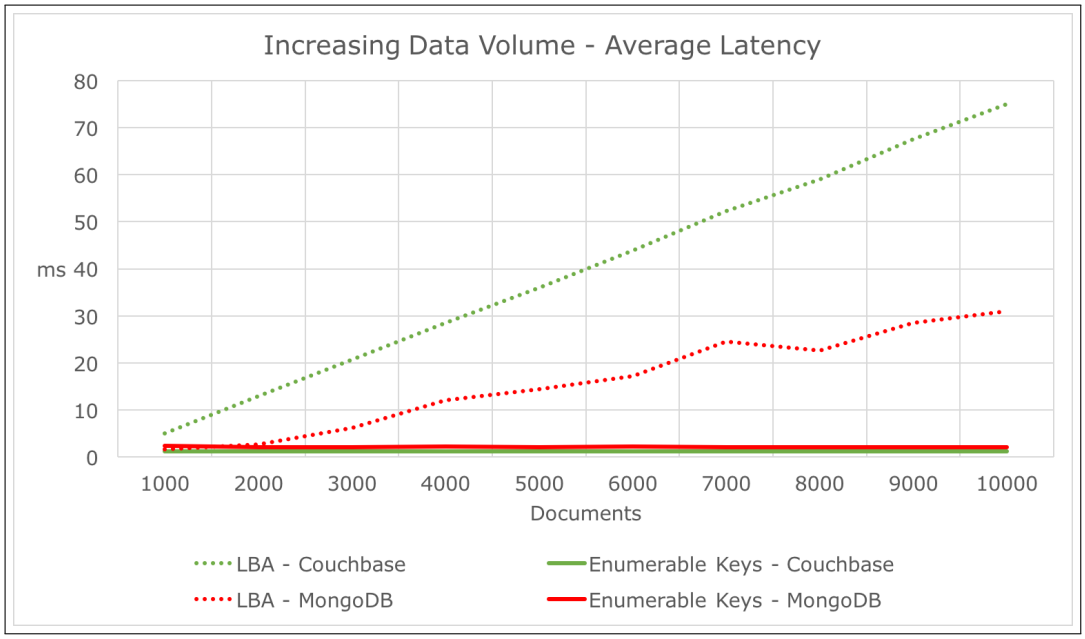
Figure 45 – LBA and Enumerable Keys patterns throughput with increasing volume of data.



Source: Created by the author

Figure 46 compares the average latency of the LBA and Enumerable Keys patterns for an increasing volume of data. For the LBA pattern, the latency increases according to the number of comments stored in the list, while for the Enumerable Keys pattern it remains stable even with the comments collection growth. The average latency is about 15 times bigger for the LBA pattern compared to the Enumerable Keys pattern in MongoDB, and about 70 times bigger in Couchbase.

Figure 46 – LBA and Enumerable Keys patterns average latency with increasing volume of data.



Source: Created by the author

5.3.5.3 Data Retrieval Tests

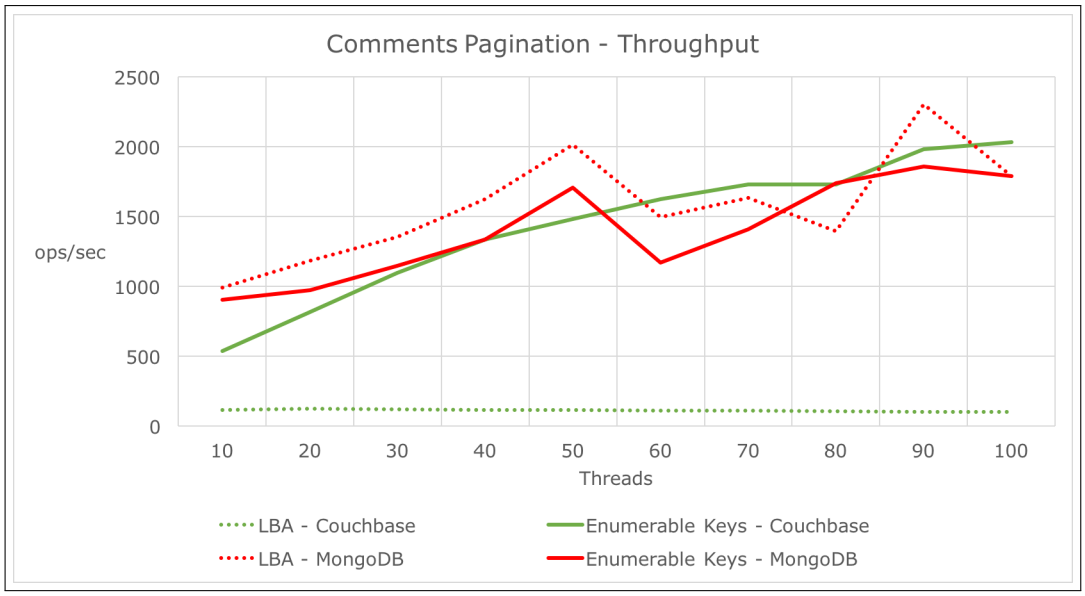
The last test concerns the behaviour of the Enumerable Keys pattern when retrieving information from the database. Although the purpose of the Enumerable Keys pattern is to leverage database write scalability, it is important to analyse how it affects the database read scalability. Generally, if a post has many comments, they are paginated so the user can see a manageable small set at a time. Considering that scenario, the workloads consist in the pagination of the comments associated with a post. Once more, the increasing number of threads simulates the growth in the number of simultaneous users.

After each workload execution, the number of threads is incremented by 10, starting with 10 and finishing with 100 threads. In every execution, each thread paginates the comments 30 times. A pagination consists in the retrieval of 10 comments from the database. The

subsequent paginations of a thread will iterate the comments following their order, but the first page of each thread is sorted at its initialization.

Figure 47 compares the throughput of the LBA and Enumerable Keys patterns regard the comments retrieval. Despite the Enumerable Keys workloads submits 10 requests to the database in order to retrieve the comment documents, the throughput is, at least, as good as the throughput of the LBA pattern workloads, which perform just one request to retrieve the 10 documents necessary to display a comments page. Despite the cloud performance fluctuation has affected the workloads executed against MongoDB, it is possible to realize the similar behaviour of both patterns, LBA and Enumerable Keys.

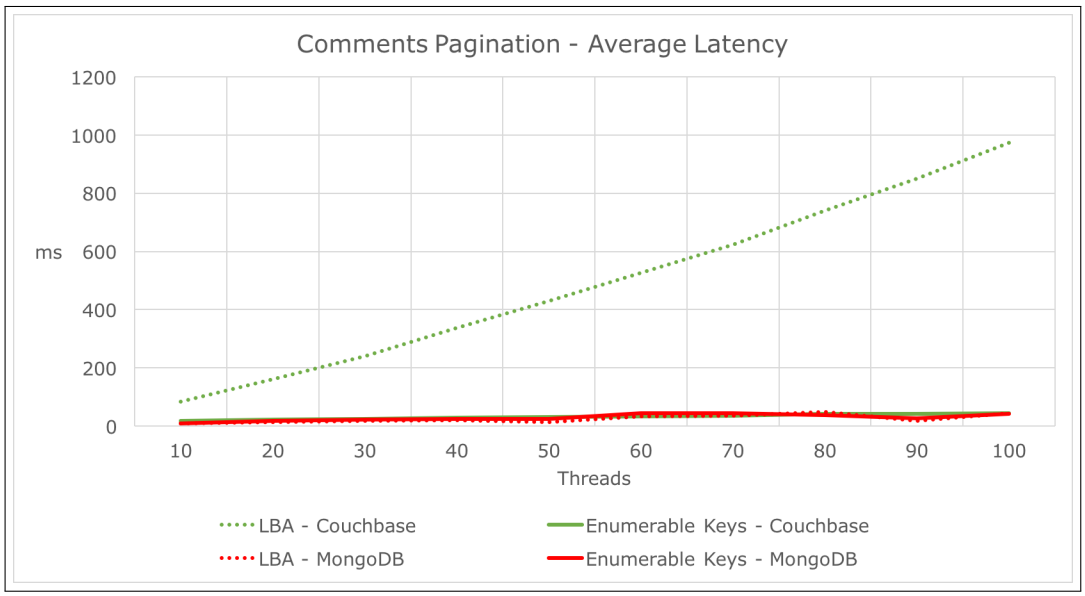
Figure 47 – LBA and Enumerable Keys patterns throughput for comments pagination.



Source: Created by the author

Figure 48 compares the average latency of the LBA and Enumerable Keys patterns regard the retrieval of comments. The figure shows that even for no-locking read operations the Enumerable Keys pattern provides satisfactory latency. However, for Couchbase the Enumerable Keys pattern performs substantially better. The LBA workload averaage latency reaches almost 1000 milliseconds at 100 threads, while for the Enumerable Keys pattern it remains about 50 milliseconds, 20 times less.

Figure 48 – LBA and Enumerable Keys patterns average latency for comments pagination.



Source: Created by the author

5.3.6 Sidebars

Since Couchbase and MongoDB are DO NoSQL databases, both support nested documents and nested lists of documents. However, there is a difference regarding how they write data to nested lists. In Couchbase, in order to add documents to a list nested in another document, it is necessary to request the main document to the database and to update the document in the application layer. Then, the application submits the updated document to the database. That two phases update process may result in optimistic/pessimistic locking failures ((ELMASRI; NAVATHE, 2010)) in concurrent environments.

On the other hand, MongoDB supports a push operation. In order to add a document to a nested list, the document that must be added is submitted to the database and it is just pushed into the attribute list of the container document. The operation is executed entirely in the database layer. That is the reason why MongoDB presents better performance than Couchbase when the LBA approach is used, as can be seen in Figures 43 and 44.

5.3.7 Consequences

The Enumerable Keys pattern leverages the scalability of the cluster since it is based on the key-value interface provided by NoSQL databases. The increase in the number of clients and in the quantity of stored data does not affect the scalability and performance of the system as

it affects when using a more basic approach like the LBA pattern. Regarded to read operations, the Enumerable Keys pattern presents equal or better performance than the standard LBA pattern approach, as can be seen in figures that report the results of the data retrieval tests.

However, the Enumerable Keys pattern is more complex than the LBA pattern. The Enumerable Keys pattern does not seem to be a natural approach as the LBA pattern does. The client application code becomes more complex because the application will be responsible for converting the normalized representation of the Enumerable Keys pattern into the list based approach used in the memory of OO applications (Figure 37).

The client application has to deal with some issues related to counter based identifiers of the Enumerable Keys pattern. Naturally, documents may be deleted from the database, and this can create missing values within a contiguous range of many-side documents identifiers. For example, in Figure 42, if the document that has the key value `uuid-p35:1202` was previously deleted, the return of the bulk get request would contain only nine documents, but ten documents were expected. In those cases, the application will have to identify that the expected number of documents was not obtained and an additional request must be submitted.

The Enumerable Keys pattern is not appropriate when the database is geographically distributed because it relies on a centralized counter. As already demonstrated in Section 5.1, the adoption of a centralized counter reduces database availability as distant clients would experiment great latency.

Due to the increase of complexity, it does not worth to use the Enumerable Keys pattern in traditional environments, where the number of simultaneous clients is not high. The compatibility between the OO paradigm and DO NoSQL databases can bring more agility to the development process. However, for handling big volumes of data in highly concurrent environment, the scalability and performance achieved with the Enumerable Keys pattern compensates the additional complexity.

5.3.8 Related Patterns

The Enumerable Keys pattern is related to the Index Table pattern because both patterns rely on the simpler key-value interface of NoSQL databases. In order to achieve better scalability, they rely on a more primitive, but faster feature, instead of using more convenient features provided by the database. The Enumerable Keys pattern is also related to the LBA pattern as both are used for the same purpose despite their different scalability capacities.

5.4 FAN-OUT ON WRITE PATTERN

5.4.1 Context

Nowadays, near real-time event streams are key features of many online applications. Many web applications allow the creation of custom feeds by selecting the event streams a consumer wants or must follow. Consumers may monitor and analyse data aggregated from multiple information producers, which can be other users or applications. Social web applications like Facebook, Instagram and Twitter allow users to "follow"their friends status, photos and posts. My Yahoo and iGoogle content aggregators allow users to customize feeds by aggregating multiple RSS sources. Dig and Reddit provide feeds based on topics like movies and sports, while news sites like CNN.com allow the monitoring of fine grained topics. When there is no need for the database to be distributed, those type of applications can be implemented with the standard normalized approach, which does not rely on denormalization. That is the scenario where a relational database can be used. However, when a NoSQL distributed database is adopted in order to improve system scalability, the normalized approach is not appropriate.

In order to illustrate the issues discussed in this section, let us consider a web application that implements a news feed feature. A user (consumer) of the example application can have many friends (producers), and the application displays the recent activities of his/her friends in his/her news feed home, sorted by reverse chronological order, from the most recent one to the oldest. As any standard social network, the application also allows the user to visualize his/her own posted activities in his/her timeline, also sorted by reverse chronological order.

Since there is an one-to-many association between a user and his/her posts, which can be many, Cassandra is a suitable datastore for this scenario. As described in chapter 2, CF databases allow to easily implement scalable one-to-many associations because their primary keys can be compound by a shard key that acts like a clustering field (ELMASRI; NAVATHE, 2010) and a sort key.

Source code 6 shows how the association between a user and his/her posts can be implemented in Cassandra 3.0 using CQL. The source code shows the table (colmun-family) that stores the users activities posted in the web application. The `content` attribute holds the activity itself, the `created_at` attribute stores when the activity was posted, and the `user_id` attribute refers to the user who the activity record belongs to, i.e., the producer of the information.

Source code 6 – Table stores user activities and the id of the user an activity record belongs to.

```
CREATE TABLE activity (
    user_id uuid,
    created_at bigint,
    content blob,
    PRIMARY KEY (user_id, created_at)
)
```

Source code 7 shows the CQL for the table that associates a user and his/her friends, who are the users followed by him/her.

Source code 7 – Table stores the friendship relationship between users.

```
CREATE TABLE friend (
    user_id uuid,
    friend_id uuid,
    PRIMARY KEY (user_id, friend_id)
)
```

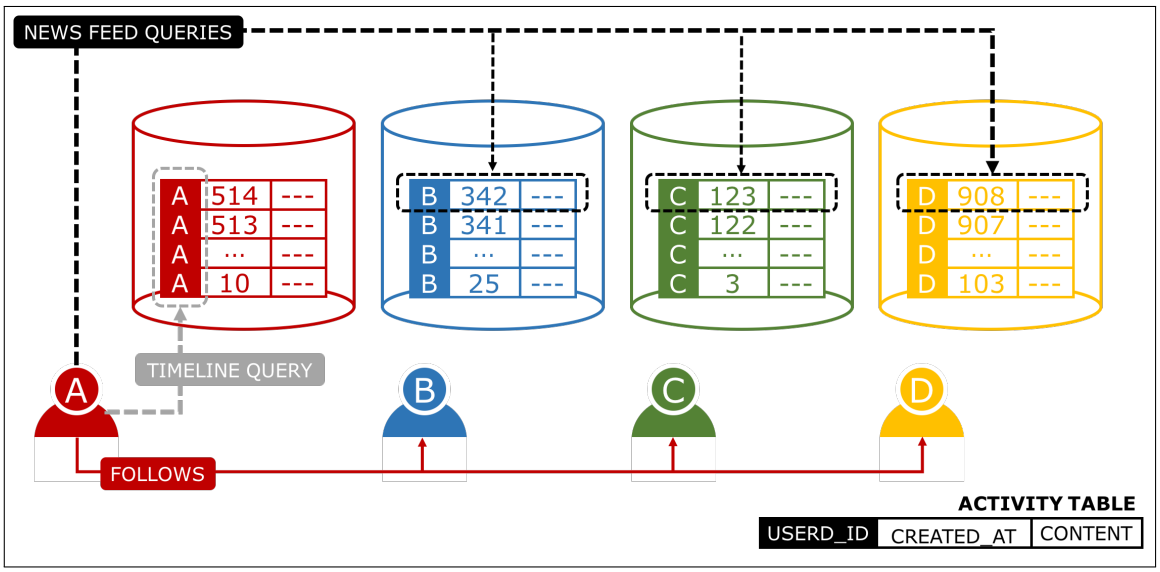
With those two tables a beginner user of NoSQL databases may think that it would be enough for implementing the news feed feature. When a user requests his/her timeline, the page that displays his/her own posts, the data is retrieved with a single query since all the user posts are stored in the same shard because they share the same shard key value, as shown in Figure 49. All the activity records of user A are stored in the same shard (the most left one), since they have the same shard key value: the user id A. However, that solution does not provide read scalability.

As illustrated in Figure 49, the posts of different users have different values as shard keys, consequently they are stored in different shards. Therefore, in order to get the most recent posts of the friends of a user, multiple queries must be submitted, and each query must use the id of a friend user as search key. Figure 49 shows that user A follows users B, C and D. Thus, in order to assemble the first news feed page for user A, three queries must be submitted to the cluster, one for each friend of user A. That pattern is known as Fan-out on Read (FOR) because the data generated by multiple producers is aggregate from multiple shards at reading time.

The FOR pattern may work for non-distributed databases[2], since the queries would

[2] In a post from Highscalability.com (http://highscalability.com/blog/2013/10/28/design-decisions-for-scaling-your-high-traffic-feeds.html), the founder of Fashiolista.com (http://www.fashiolista.com) reports that their

Figure 49 – The posts of a user are kept in the same shard.



Source: Created by the author

hit a single node. However, when a distributed database is used, scattering multiple queries across the cluster does not favor scalability and resembles the use of secondary indexes with distributed NoSQL databases.

5.4.2 Problem

How to implement near real-time event streams aggregation features in NoSQL distributed databases in order to provide read scalability for applications that must handle a heavy read load for those features?

5.4.3 Forces

- Sharding records by the producers ids provides read scalability when the producer reads its own data or when a consumer reads data from a single producer. On the other hand, that approach compromises scalability when the monitored data must be aggregated from different shards.
- When using the Fan-out on Read pattern, since the records are independently fetched from different nodes, it is not possible to retrieve from the database all the set of records already sorted by some criteria, the producers records are sorted only in the shard they reside. Consequently, the application must sort the data after fetching all the necessary records.

feed feature was first deployed on a PostgreSQL database and didn't presented scalability issues during certain period.

- CQL has an `IN` clause that works like the `IN` clause of SQL. That clause accepts multiple keys as input parameter and allows to retrieve from the cluster nodes all the records designate by those keys with a single request. However, that approach does not solve the problem since the node who accepts the query triggers multiple requests across the cluster. That same node gathers the results and return them to the application.

5.4.4 Solution

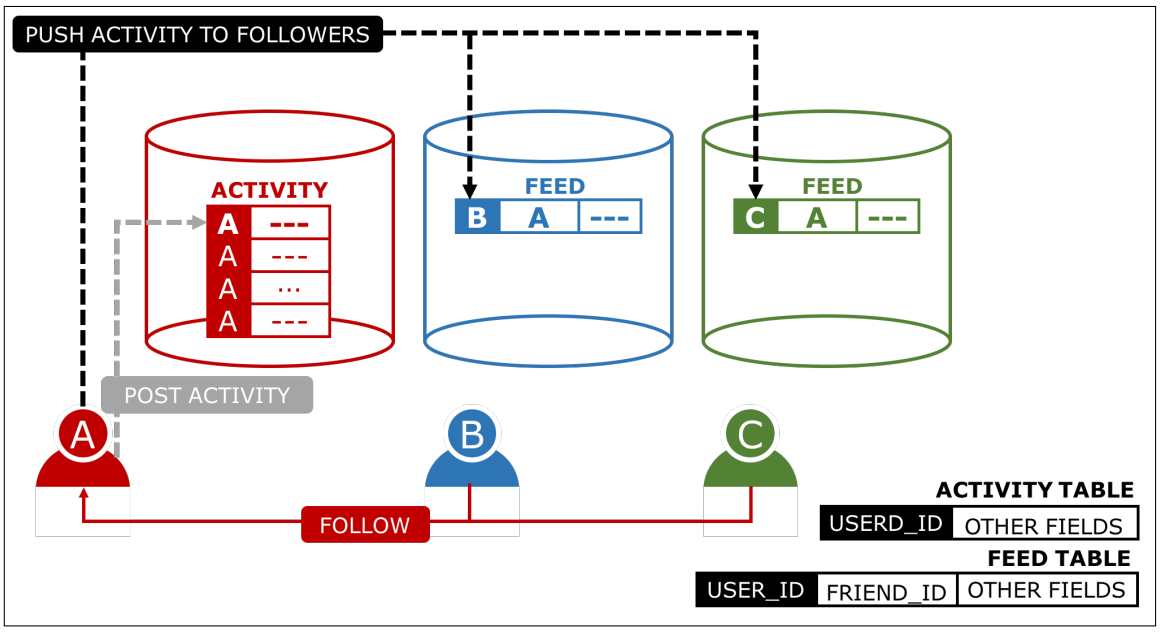
The solution is based on data denormalization and redundancy. The consumers event streams must be materialized at writing time by storing one copy of each producer event by consumer, using the consumers ids as shard key. There is, instead of having the consumer aggregating the records that compose its event stream by fanning-out read request to all nodes of the cluster, at writing time, the application stores multiple copies of each record generated by the producer, one for each consumer, using the id of the consumer as shard key. Consequently, when the consumer requests its event stream, the records are already aggregated in a single shard and sorted. That approach is known as Fan-out on Write (FOW).

Considering the example application described in the Context section, instead of having a user pulling the activities posted by his/her friends at reading time, activities posted by a user are pushed to all his/her followers at the activity writing time. Pushing activity records posted by a user to all his/her followers in advance makes possible to retrieve a user news feed page with a single query and reduced latency.

Let us suppose users B and C follow user A. As depicted by Figure 50, when user A posts an activity, the application stores the correspondent record in the activities table using the user id as shard key and the activity date as sort key (not displayed in Figure 50). Additionally, the application stores in the feeds table an additional record to each follower of the user A. As shown in Figure 50, the activity posted by the user A generates two feed records, one for each follower of the user A: user B and user C. For the feed records, the id of the follower user is used as shard key and the activity date is used as sort key (not displayed in Figure 50). The id of the user who has generated the activity (user A) is stored as an attribute of the feed record in order to identify the author of the activity.

Figure 51 describes the second moment of this explanation. User B is followed by users A and C. Following the same flow described above, an activity generated by user B is stored in the activities table and generates two additional records, stored in the feeds table, one

Figure 50 – Application pushes user A activity record to other users.



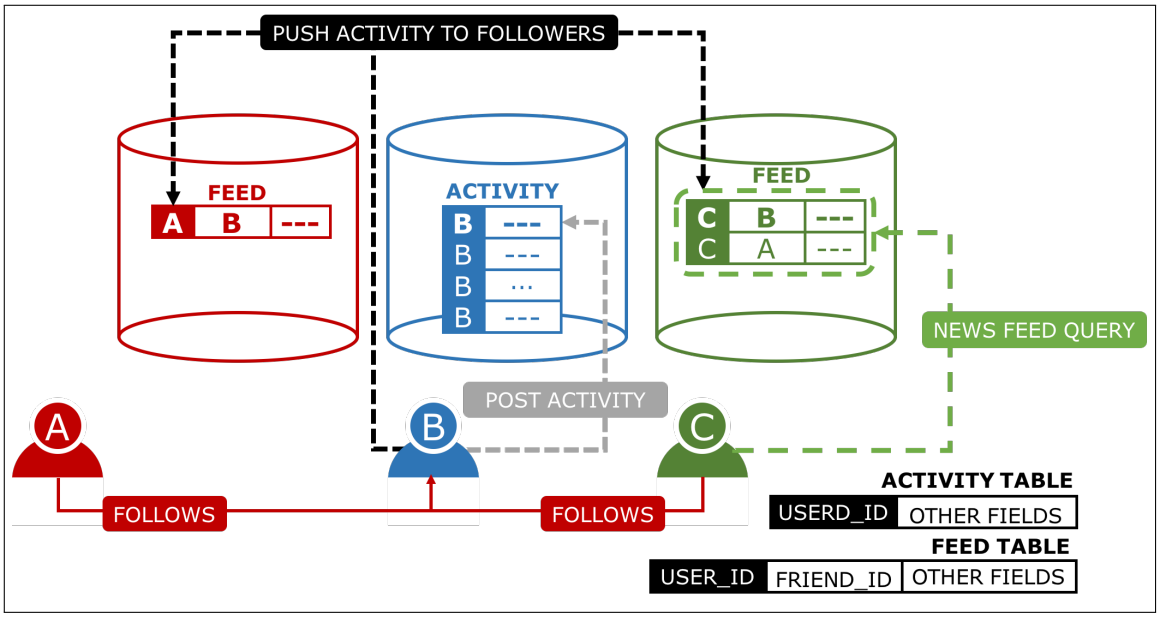
Source: Created by the author

for each follower (users A and C). It is possible to notice that if user C accesses his/her news feed in the application, the activities of the users he/she follows (users A and B) are stored in the same shard of the feeds table, which means that a single node must be queried in order to retrieve the records that compose his/her timeline. In addition, as the activity date is used as sort key, the activity records are chronologically sorted no matter which user has posted that activity.

5.4.5 Scalability Tests

The web application described in the previous sections is used in this section in order to compare the Fan-out on Read and Fan-out on Write patterns. As in the previous example, Cassandra is the database used in the tests since its shard key acts like a clustering key and it allows the records in the same shard to be sorted by the sort key. That capability makes Cassandra taking better advantage of the FOW pattern. The database cluster used in the tests is composed of 4 nodes. For the following tests the simulated application has 1000 users, and each user follows another 100 users. That means that the table that stores the users friendship relationships has 100000 records.

Figure 51 – Activity records of users followed by user C are stored in the same shard and chronologically sorted.



Source: Created by the author

5.4.5.1 Save Activities Test

The first test compares the FOR and FOW patterns regarding the storage of the activities posted by the users. That comparison is important since there is a trade-off between read scalability and write scalability when both approaches are considered.

In this test each approach is represented by a workload. The workloads simulate multiple concurrent users posting new activities in the web application. The FOR-based workload saves a single record in the activities table for each user new activity. On the other hand, the FOW workload stores an additional record in the feeds table for each follower of a user.

In this test, each user posts 80 activities. Since the simulated application has 1000 users, each workload stores 80000 new activities in the activities table. For each activity stored in that table, the workload that implements the FOW pattern stores additional 100 activities in the feeds table, since each user is followed by another 100 users.

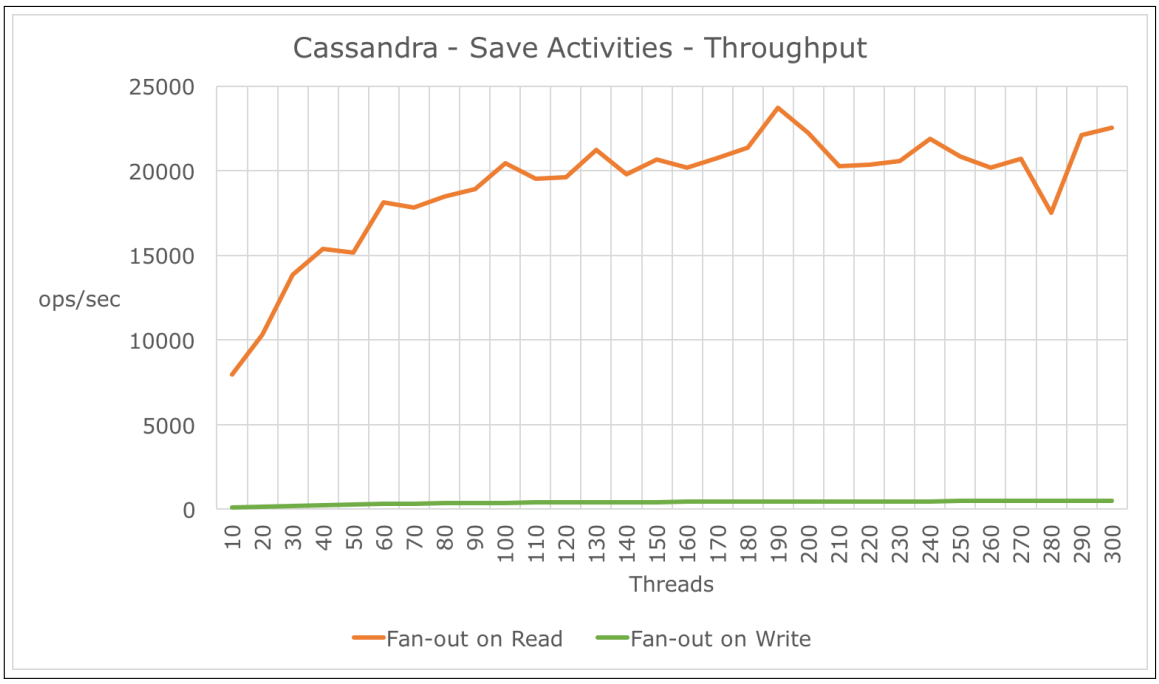
In this test, each workload executes 30 times and the number of simultaneous client threads is increased by 10 for each execution, starting from 10 up to 300. The simultaneous threads simulate multiple concurrent users posting their activities in the application. However, a single thread does not correspond to a single user. A thread does not bind to a user and stores all his/her activities. Therefore, the increasing number of threads between the executions of a

workload simulates an increasing portion of the 1000 users concurrently posting their activities.

It works like a pool of threads in which each thread randomly chooses a user and saves his/her current activity. That process executes until all 80000 activities are stored. For the FOW workload, each thread has an additional step, after storing the current user activity, the same thread stores the additional 100 records in the feeds table, one for each follower of the current user. Each subsequent activity posted by a user has its time increased by one second over the previous activity. That approach avoids the behaviour of storing an unreal amount activities with the same value for the date field and simulates natural time elapsing.

Figure 52 compares the throughput between the FOR and FOW patterns. For the FOR pattern, an operation consists of storing a single activity record, while for the FOW pattern an operation consists of storing the user activity record and all the additional 100 feed records for the followers of the current user. Consequently, the FOR pattern handles more efficiently the increase in the number of simultaneous requests. Its throughput increased with a much greater rate than the throughput of the FOW pattern. At 300 threads, the throughput of the FOR pattern is more than 20 times greater than its FOW pattern counterpart

Figure 52 – FOR and FOW patterns throughput for concurrent activities insertions.

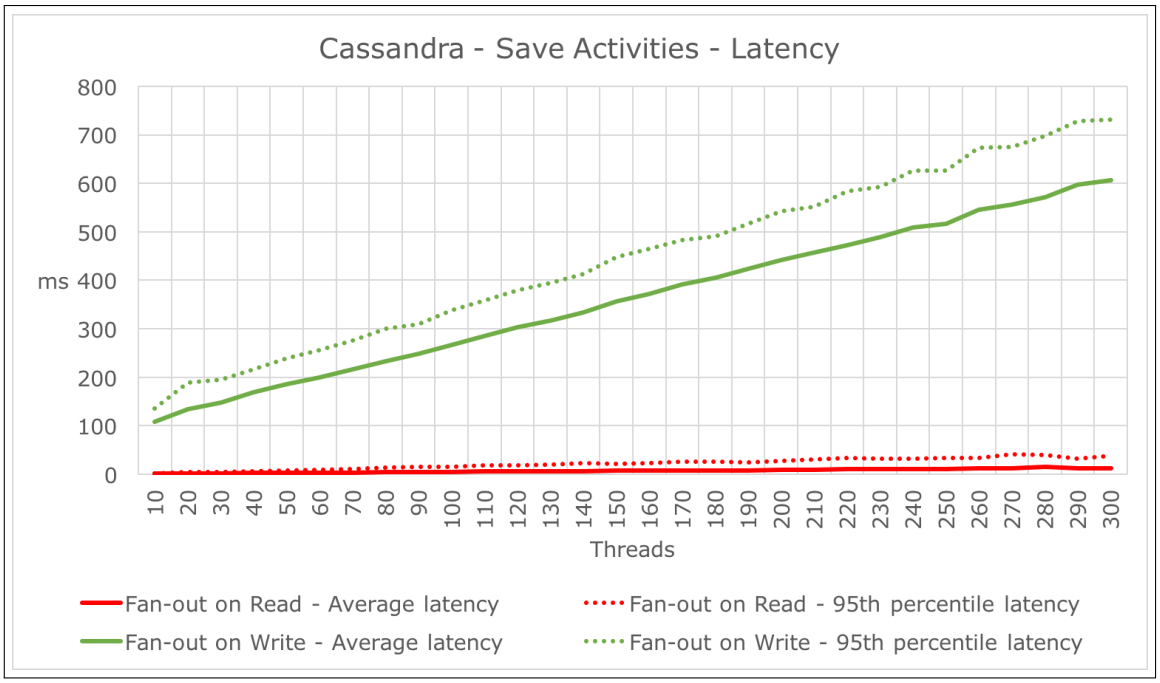


Source: Created by the author

Figure 53 shows that the latency of the FOW pattern grows with a much bigger rate than the latency of the FOR pattern when the number of concurrent requests to store user

activities increases. With only 10 threads concurrently posting user activities, the latency of the FOW pattern is about 10 times bigger than the latency of the FOR pattern. When the number of threads reaches 300, the latency of the FOW pattern becomes about 60 times bigger than the latency of the FOR pattern.

Figure 53 – FOR and FOW patterns latency for concurrent activities insertions.



Source: Created by the author

In order to improve the performance and scalability of the FOW pattern, a second pool of threads could be used to store the followers feed records in parallel. But that approach has not been not chosen in order to better illustrate the additional load that the FOW pattern has to handle to push the activities of a user to all its followers.

5.4.5.2 Paginate Feed Test

The second test compares the FOR pattern and the FOW pattern regarding the retrieval of the users news feeds from the database. Each pattern has been implemented as a workload, and each workload simulates a user paginating his/her news feed. As in the previous test, there is a pool of threads that simulates an increasing number of simultaneous application users paginating their news feeds. Each workload is executed 30 times and the number of threads in the pool increases by 10 after each workload execution, starting from 10 up to 300. Each news feed page displays 10 activities, and each workload paginates the entire news feed for all users.

That means that 800000 pagination operations are executed in each workload execution.

When a thread randomly chooses a user in order to paginate his/her news feed, as in the previous test, it does not paginate the user news feed until the last page. The thread assembles a single news feed page and then chooses another user. When a thread chooses a user that has already been chosen in the past, the pagination of the user news feed is resumed, the workload knows the last page assembled for that user.

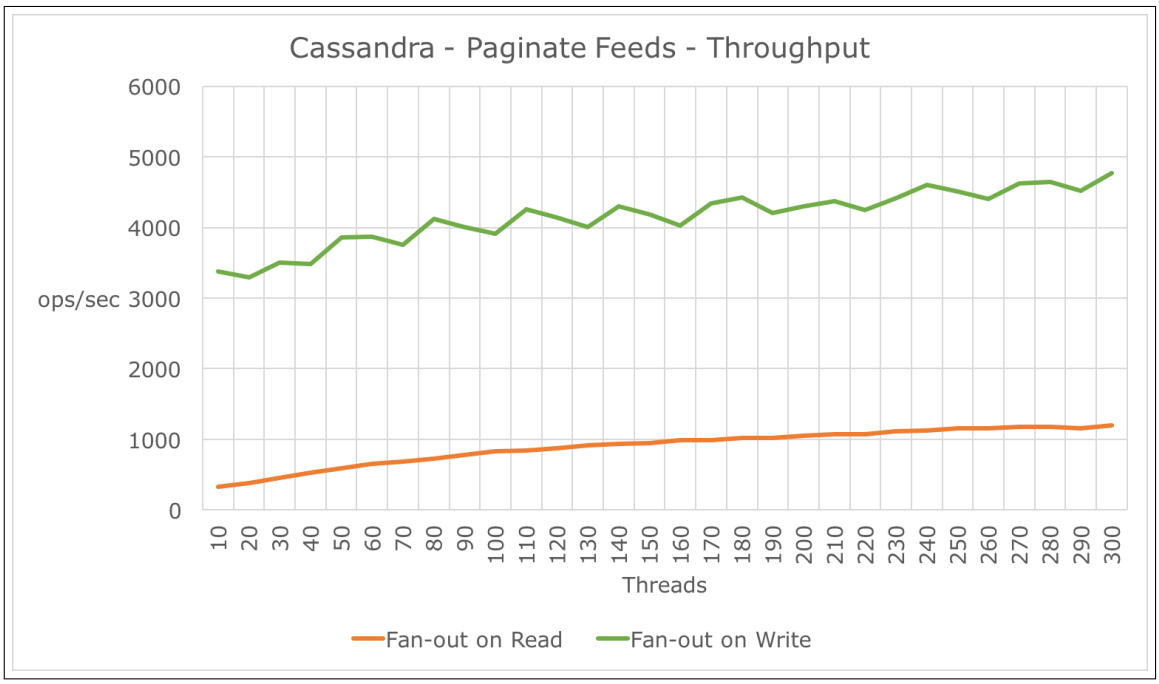
As illustrated in Figure 51, when the FOW pattern is used, it is possible to get the 10 most recent activities of a user friends with a single request since all those records are kept in the same shard and they are sorted in reverse chronological order. When the same user is chosen by another thread, the thread resumes the users news feed pagination by requesting the 10 second most recent activities, and so on.

On the other hand, the FOR pattern workload has to submit 10 queries to the database in order to assemble a user news feed page. Since each user follows 100 friends, a thread executing the FOR workload code submits 10 queries to the database requesting the most recent activity of the first 10 friends of the current user. When that same user is chosen again, the thread will request the most recent activity of the second set of 10 friends of that user, and that process goes on until the most recent activity of all the 100 friends are paginated. Then, the iteration through user friends are restarted, but this time the second most recent activity of each user will be requested from the database.

Figure 54 compares the throughput between the FOR pattern and the FOW pattern for the news feed pagination. That chart shows that for assembling a user news feed page, the FOW pattern provides a throughput about 4 to 6 times greater that the FOR pattern considering the interval between 10 and 300 threads. The throughput for both patterns increases as the number of client threads grows. However the FOW pattern presents a greater growing rate since it grows from less than 3500 operations (pages) per second to almost 5000, while the FOR pattern throughput grows from less than 500 operations (pages) per second to few more than 1000.

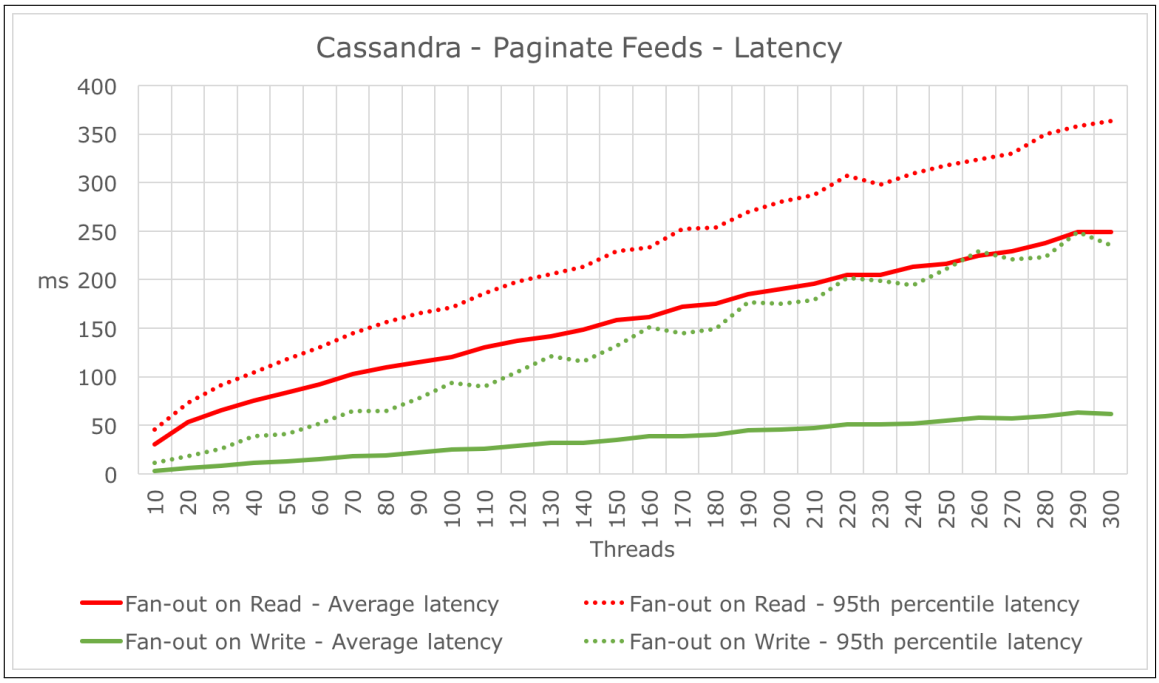
Figure 55 compares the latency between the FOR pattern and the FOW pattern regarding the news feed pagination. The FOW pattern presents a more stable growing of the latency than the FOR pattern. At 300 threads, the average latency of the FOR pattern workload is 5 times bigger than the average latency of the FOW pattern workload.

Figure 54 – FOR and FOW patterns latency for concurrent news feed pagination.



Source: Created by the author

Figure 55 – FOR and FOW patterns latency for concurrent news feed pagination.



Source: Created by the author

5.4.6 Sidebars

The Fan-out on Read and Fan-out on Write patterns are more appropriate for NoSQL databases that provide primary keys that can be composed by a shard key and a sort key, since

the shard key acts like a clustering key. Thus, CF databases like Cassandra, DynamoDB and Google Cloud Bigtable are examples of those databases. All the records that store the same shard key value are kept in the same node of the cluster, and they can be sorted by the sort key, which generally is a date column for time series data. Those type of databases can handle very well event streams and content aggregation systems since a very large number of records are associated to an entity, that means a lot of one-to-many associations that are easily implemented with clustering keys.

DO databases like Couchbase and MongoDB are not an appropriate choice for those type of systems when they are very popular or demanded, since representing one-to-many associations in those databases is more complex, as demonstrated by the Enumerable Keys pattern (section 5.3). It is possible to combine the FOW and Enumerable Keys patterns. However, that approach is complex and may not worth it.

5.4.7 Consequences

In order to improve read scalability, the Fan-out on Write pattern impairs write scalability. For applications in which consumers of content aggregations are much more active than content producers, i.e., read-oriented applications, the data redundancy and additional processing required by the FOW pattern pays off. Additionally, when the aggregated records must be sorted, the FOW pattern avoids sorting at reading time at application side.

The FOW pattern requires a lot of additional space to store many copies of producers events, one for each consumer (follower). That additional space can be reduced by storing just a reference that points to the producers event records, instead of storing the content of the event itself. The FOW pattern also causes a lot of network communication overhead since each copy of a producer event may be stored in a different shard. Since the approach suggested by the FOW pattern heavily relies on writing, it presents better performance on write-optimized databases, like Cassandra.

The low scalability and performance presented by the FOW pattern at the pushing phase can be minimized by using background threads to write in parallel the producers records copies to the consumers shards. That increases its complexity but can greatly improve its performance and scalability. However, let us suppose a news feed application that has users with a huge number of followers, for instance, Donald Trump and Barack Obama. Pushing their activities to all their followers is a challenge even with a huge pool of threads. Silberstein et al.

(2010) suggest a selective approach that basically consists of disabling the FOW process for very active producers who have many consumers and falling back to the FOR strategy.

Another improvement that can increase the write performance of the FOW pattern is to use different priorities to the push threads. Threads writing (pushing) content generated by very active producers with many consumers should have bigger priority than threads writing content generated by producers who have few consumers.

5.4.8 Related Patterns

The Fan-out on Write and Fan-out on Read patterns are related since both approaches can be used for the same purpose, the choice depends on the application nature. Additionally, as suggested by Silberstein et al. (2010), a combination of both approaches can be used. The FOW patterns also relates to the Enumerable Keys pattern (section 5.3) since both can be combined in order to implement near real-time event streams applications in document-oriented databases.

5.5 CONCLUSION

In order to provide a fast consulting resource, Table 3 summarizes the properties of the each pattern by enumerating: the pattern applicability, its benefits, its drawbacks, and candidate databases.

Table 3 – Summary of pattern properties.

Pattern	Applicability	Benefits	Drawbacks	Databases
UUID Key	With wide-area distributed clusters, when it is necessary to generate primary keys without database coordination and with pure key-value databases.	Leverages database scalability as it decentralizes the generation of primary key values. Increases database availability and consistency since it is not affected by network partition neither network high latency. Provides a wide range of distinct values for records primary keys.	Requires more disk space. The generated primary key values are not human-friendly. It does not support primary key based range queries, nor sorting.	Distributed databases: Cassandra, Couchbase, DynamoDB, Google Cloud Bigtable, MongoDB, Memcached, Redis and more...
Index Table	In order to submit equality queries for NoSQL databases that do not provide global secondary hash indexes.	Leverages database scalability since it avoids scatter-gather queries across the cluster nodes. It can play the role of a unique index.	It is not possible to create lookup records for attributes with duplicated values in a collection. The application must guarantee the consistency between primary and lookup records.	Cassandra, Couchbase, Google Cloud Bigtable, MongoDB.
Enumerable Keys	With NoSQL databases that do not provide compound primary keys with clustering key (KV and DO databases), in order to implement one-to-many associations when the order of the records in the many side matters.	Leverages database write scalability since it allows indepent sharding of associated collections. Imposes an order to a set of records enabling the traversing of the records.	Creates an impedance mismatch since the database schema does not directly correspond to applications data structures. Application has to handle missing records when traversing records on the many side of the association. Creates a potential bottleneck due to the dependency of a pseudo-centralized counter.	Couchbase, MongoDB, Elasticsearch.
Fan-out on Write (FOW)	For implementing near real-time event streams features for read-oriented applications with CF databases.	Leverages database read scalability since it stores all data related to a consumer feed in a single shard and keeps data already sorted in the desired order.	Based on great data redundancy and causes a lot of network overhead since it must store an additional record to every consumer (follower) of a producer.	Cassandra, DynamoDB, Google Cloud Bigtable.

Source: Created by the author

6 CONCLUSION

6.1 PATTERNS CONTRIBUTIONS

This master thesis presented four patterns that aim to leverage the scalability of aggregate-oriented NoSQL databases. Differently from traditional pattern forms, the patterns presented in this work are supported by benchmark fashion tests that compare the pattern with a commonly employed, but simpler, implementation that does not perform as well as the pattern solution. That approach has provided a less abstract analyses of the improvements and drawbacks of the solution recommend by the pattern. The proposed patterns allow a more concrete perception regarding the scalability improvement of the application.

The patterns presented in this master thesis reinforce some principles that must guide architects and developers when modeling the data layer of OLTP applications that rely on aggregate-oriented NoSQL databases and require greater scalability levels, which are:

- Avoid modeling schemas that concentrate requests in a single node since it may generate SPOFs and concurrency conditions. The UUID Key and Enumerable Keys patterns avoid the concentration of request in a single data shard.
- Accept denormalization and data redundancy since those concepts provide alternative paths for retrieving the necessary information. The Index Table and Fan-out on Write patterns heavily rely on data redundancy and denormalization in order to provide alternative data paths that contribute for leveraging application scalability.
- Model database schema[1] addressing queries based on the key-value interface provided by the database since it is one of the foundations behind the great scalability capacity of NoSQL databases. The Index Table, Enumerable Keys and Fan-out on Write patterns avoid the use of local secondary indexes by relying on key-value scalable operations.

Generally, professionals that are starting to learn and use NoSQL databases do not apply the before-mentioned principles the way they should since they require experience. The proposed patterns describe solutions that embed those concepts helping inexperienced users to build scalable NoSQL database based applications.

The test presented for each pattern help to avoid problems that can reduce the system scalability. For instance, they reinforced the best practice of avoiding the use of secondary indexes provided by NoSQL databases with features that require greater scalability levels when

[1] The implicit database schema impose by the application.

those indexes are local since they scatter multiple requests across the cluster impairing database scalability.

Another caveat avoided by the patterns, specifically the Enumerable Keys pattern, is the overuse of the aggregate concept in document-oriented NoSQL databases. Aggregating the data required by some feature as a single unit of information is a fundamental scalability concept for NoSQL databases and is an attractive feature of document-oriented databases, however, the misuse of that feature may generate scalability issues.

As the before-mentioned principles are not correctly applied by inexperienced users of NoSQL databases, the incorrect employment of features may also generate scalability issues. Therefore, the presented patterns also help NoSQL novice professionals to avoid those caveats.

The use of patterns may provide more independence regard proprietary features of a specific NoSQL database product. Relational databases provide standard features that facilitate the exchange of database products. However, change a NoSQL database product may not be as easy. In order to implement the features necessaries to fulfil application requirements, it is common to use proprietary features implemented only by the adopted database. The adoption of patterns for NoSQL databases can facilitate the transition between databases. For example, in order to improve the scalability of secondary indexes in Couchbase, indexes data can be sharded independently of business data, and even replicated. However, the management of that solution is complex, while employing the Index Table pattern to improve non-range queries is simpler and scalable.

The development of the workloads executed in the pattern tests and their results confirmed that column-family databases present greater scalability than document-oriented databases when storing data that represent associations between business domain entities. The Enumerable Keys pattern is a scalable approach for storing one-to-many associations in document-oriented databases, however, due to their capacity of storing multiple records that have the same partition key in the same shard, column-family databases provide better scalability for those cases. That is the reason why column-family databases are indicated for time series data, social networks, and content management applications, which present multiple associations between business entities.

6.2 TECHNOLOGICAL CONTRIBUTIONS

In addition to providing a less-abstract and practical aspect to the patterns, the workloads developed for each pattern test are an important technological contribution since they are publicly available, thus allowing interested readers to download and customized the workloads in order to execute them in their own environments. The workloads can be modified by supplying different input parameters or changing their source codes.

The YCSB framework has been chosen in order to implement the pattern tests because is one of the most used tools for benchmarking NoSQL databases and cloud datastores. Additional functionalities based on the YCSB framework have been developed in order to attend some requirements of the patterns tests. Those additional features are also contributions of this work as they are public available for use and modifications. The developed extensions are listed below:

- Submissions of requests independent of the database client driver API, including the backoff handling;
- Synchronization of the start of parallel workloads;
- Simplification of the handling of product specific database exceptions;
- Additional approach that enable a workload to connect to multiple database clusters;

Although all the extensions developed are important technological contributions, the first extension of the list above is the most important one as it departures from the default approach provided by NoSQL client to interact with the datastore. Instead of interacting with all databases through a common interface that restricted the set of available operations, the workload classes hierarchy have been extended in order to allow the direct use of the client driver APIs of the databases. The source code that implements that approach for Cassandra, Couchbase and MongoDB are publicly available for use and modifications. In order to interact with an additional database using the same approach, the user must provide the client library for the target database and extend key classes according to the provided documentation.

The YCSBtoCSV utility tool is also a contribution of this works since it can increase the productivity and make less error prone the task of extracting the test results metrics from big log files generated from multiple contiguous executions of the YCSB client.

6.3 FUTURE WORK

Naturally, the first future work is to document additional well-proven modeling approaches that leverage the scalability of systems that rely on aggregate-oriented NoSQL databases, in order to create a system of patterns (BUSCHMANN et al., 1996) that address scalability. The additional patterns should be described using the template presented in this work, including the scalability tests. A research could be executed in order to verify the extent of adoption of the selective approach presented in (SILBERSTEIN et al., 2010) and consider if it should be include in the proposed system of scalable patterns.

Another relevant NoSQL databases could be tested regarding the approaches presented by the introduced patterns. The additional workloads developed should be shared in the public repository in order to increase the number of NoSQL database products that interest users can benchmark regarding the scalability in the context of the patterns presented in this work.

An study could be conducted in order to verify the possibility of merging the YCSB based extensions developed in this work into the official YCSB project, or creating a parallel project. An important contribution of the proposed activity would be to allow the workloads to use the operations provided by the specific NoSQL database being tested through the common interface provided by the YCSB client tool. That means that the most important contribution of the YCSB extension developed in this work (exposing the database client driver API) would be available through the already know interface provided by the YCSB framework. A wrapper (GAMMA et al., 1995) class could be implemented in order to encapsulate the database client driver API and the traditional key-value operations provided by the YCSB client API.

The YCSB++ tool presented in (COOPER et al., 2010) provides relevant features that could be merged with the YCSB extensions developed in this work. YCSB++ provides the capacity of synchronizing the execution of parallel workloads through Apache Zookeeper[2]. That solution is robust and could substitute the feature developed for synchronizing the starting of parallel workloads used in this work.

[2] https://zookeeper.apache.org/

REFERENCES

ALEXANDER, C.; ISHIKAWA, S.; SILVERSTEIN, M. **A Pattern Language**. [S.l.]: Oxford University Press, 1977. ISBN 0195019199.

ALUR, D.; MALKS, D.; CRUPI, J. **Core J2EE Patterns: Best Practices and Design Strategies**. 2nd. ed. [S.l.]: Prentice Hall, 2003. ISBN 978-0131422469.

BREWER, E. Cap twelve years later: How the "rules"have changed. **Computer**, IEEE, v. 45, n. 2, p. 23 – 29, fev. 2012. ISSN 0018-9162.

BREWER, E. A. Towards robust distributed systems (abstract). In: **Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing**. New York, NY, USA: ACM, 2000. (PODC '00), p. 7–. ISBN 1-58113-183-6. Disponível em: <http://doi.acm.org/10.1145/343477.343502>.

BUSCHMANN, F.; MEUNIER, R.; ROHNERT, H.; SOMMERLAND, P.; STAL, M. **Pattern-Oriented Software Architecture. A System of Patterns.** [S.l.]: John Wiley Sons Ltd, 1996. ISBN 9780471958697.

CAMERON, D. **A Software Engineer Learns Java 8. The Fundamentals**. [S.l.]: Cisdal Publishing, 2014.

CONNOLY, T.; BEGG, C. **Database Systems. A Practical Approach to Design, Implementation, and Management.** 4th. ed. [S.l.]: Addison-Wesley Publishing Company, 2005. ISBN 0321210255.

COOPER, B. F. Spanner: Google's globally-distributed database. In: **Proceedings of the 6th International Systems and Storage Conference**. New York, NY, USA: ACM, 2013. (SYSTOR '13), p. 9:1–9:1. ISBN 978-1-4503-2116-7. Disponível em: <http://doi.acm.org/10.1145/2485732.2485756>.

COOPER, B. F.; SILBERSTEIN, A.; TAM, E.; RAMAKRISHNAN, R.; SEARS, R. Benchmarking cloud serving systems with ycsb. In: **Proceedings of the 1st ACM Symposium on Cloud Computing**. New York, NY, USA: ACM, 2010. (SoCC '10), p. 143–154. ISBN 978-1-4503-0036-0. Disponível em: <http://doi.acm.org/10.1145/1807128.1807152>.

COSTA, C.; VIANNEY, J.; MAIA, P.; OLIVEIRA, F. Sharding by hash partitioning. a database scalability pattern to achieve evenly sharded database clusters. In: **Proceedings of the 17th International Conference on Enterprise Information Systems**. [S.l.]: SCITEPRESS, 2015. p. 313–319.

DEAN, J.; GHEMAWAT, S. Mapreduce: Simplified data processing on large clusters. **Commun. ACM**, ACM, New York, NY, USA, v. 51, n. 1, p. 107–113, jan. 2008. ISSN 0001-0782. Disponível em: <http://doi.acm.org/10.1145/1327452.1327492>.

DEWITT, D.; GRAY, J. Parallel database systems: The future of high performance database systems. **Commun. ACM**, ACM, New York, NY, USA, v. 35, n. 6, p. 85–98, jun. 1992. ISSN 0001-0782. Disponível em: <http://doi.acm.org/10.1145/129888.129894>.

DEY, A.; FEKETE, A.; NAMBIAR, R. Ycsb+t: Benchmarking web-scale transactional databases. In: **Proceedings of 2014 IEEE 30th International Conference on Data Engineering Workshops (ICDEW)**. IEEE, 2014. ISBN 978-1-4799-3481-2. Disponível em: <http://ieeexplore.ieee.org/document/6818330/>.

ELMASRI, R.; NAVATHE, S. **Fundamentals of Database Systems**. 6th. ed. USA: Addison-Wesley Publishing Company, 2010. ISBN 0136086209, 9780136086208.

GAMMA, E.; HELM, R.; JOHSON, R.; VLISSIDES, J. **Design Patterns. Elements of Reusable Object-Oriented Software**. [S.l.]: Addison-Wesley, 1995. ISBN 0201633612.

HARATY, R. A.; STEPHAN, G. Relational database design patterns. In: **Proceedings of 2013 IEEE 16th International Conference on Computational Science and Engineering (CSE)**. IEEE, 2013. ISBN 978-0-7695-5096-1. Disponível em: <http://ieeexplore.ieee.org/document/6755304/>.

HOHPE, G.; WOOLF, B. **Enterprise Integration Patterns. Designing, Building, and Deploying Message Solutions**. [S.l.]: Addison-Wesley Publishing Company, 2004. ISBN 0321200683.

KAUR, K.; RANI, R. Modeling and querying data in nosql databases. In: **Proceedings of the 2013 IEEE International Conference on Big Data**. IEEE, 2013. ISBN 978-1-4799-1293-3. Disponível em: <http://ieeexplore.ieee.org/document/6691765/>.

LAMLLARI, R. **Extending a Methodology for Migration of the Database Layer to the Cloud Considering Relational Database Schema Migration to NoSQL**. Dissertação (Master) — University of Stuttgart, Stuttgart, 2013.

LEACH, P. J.; SALZ, R.; MEALLING, M. H. **A Universally Unique IDentifier (UUID) URN Namespace**. RFC Editor, 2015. RFC 4122. (Request for Comments, 4122). Disponível em: <https://rfc-editor.org/rfc/rfc4122.txt>.

MARCHIONI, F. **MongoDB for Java Developers**. [S.l.]: Packt Publishing Ltd., 2015. ISBN 9781785280276.

MIOR, M. J. Automated schema design for nosql databases. In: **Proceedings of the 2014 SIGMOD PhD Symposium**. New York, NY, USA: ACM, 2014. (SIGMOD'14 PhD Symposium), p. 41–45. ISBN 978-1-4503-2924-8. Disponível em: <http://doi.acm.org/10.1145/2602622.2602624>.

NEIL, T. **Mobile Design Pattern Gallery. UI patterns for smartphones apps.** 2nd. ed. [S.l.]: O'Reilly Media, 2014. 403 p. ISBN 9781449363635.

PATIL, S.; POLTE, M.; REN, K.; TANTISIRIROJ, W.; XIAO, L.; LóPEZ, J.; GIBSON, G.; FUCHS, A.; RINALDI, B. Ycsb++: Benchmarking and performance debugging advanced features in scalable table stores. In: **Proceedings of the 2Nd ACM Symposium on Cloud Computing**. New York, NY, USA: ACM, 2011. (SOCC '11), p. 9:1–9:14. ISBN 978-1-4503-0976-9. Disponível em: <http://doi.acm.org/10.1145/2038916.2038925>.

PIRZADEH, P.; TATEMURA, J.; PO, O.; HACüMüs, H. Performance evaluation of range queries in key value stores. **J. Grid Comput.**, Springer-Verlag New York, Inc., Secaucus, NJ, USA, v. 10, n. 1, p. 109–132, mar. 2012. ISSN 1570-7873. Disponível em: <http://dx.doi.org/10.1007/s10723-012-9214-7>.

SADALAGE, P. J.; FOWLER, M. **NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence**. 1st. ed. [S.l.]: Addison-Wesley Professional, 2012. ISBN 0321826620, 9780321826626.

SHARMA, S. **Cassandra Design Patterns**. 1st. ed. [S.l.]: PACKT PUBLISHING, 2014. ISBN 9781783288809.

SHIRAZI, M. N.; KUAN, H. C.; DOLATABADI, H. Design patterns to enable data portability between clouds' databases. In: **Proceedings of the 2012 12th International Conference on Computational Science and Its Applications**. Washington, DC, USA: IEEE Computer Society, 2012. (ICCSA '12), p. 117–120. ISBN 978-0-7695-4710-7. Disponível em: <http://dx.doi.org/10.1109/ICCSA.2012.29>.

SILBERSTEIN, A.; TERRACE, J.; COOPER, B. F.; RAMAKRISHNAN, R. Feeding frenzy: Selectively materializing users' event feeds. In: **Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data**. New York, NY, USA: ACM, 2010. (SIGMOD '10), p. 831–842. ISBN 978-1-4503-0032-2. Disponível em: <http://doi.acm.org/10.1145/1807167.1807257>.

STRAUCH, S.; ANDRIKOPOULOS, V.; BREITENBüCHER, U.; SáEZ, S. G.; KOPP, O.; LEYMANN, F. Using patterns to move the application data layer to the cloud. In: **Proceedings of the 5th International Conferences on Pervasive Patterns and Applications (PATTERNS)**. [S.l.: s.n.], 2013. ISBN 978-1-61208-276-9.

TAURO, C. J. M.; PATIL, B. R.; PRASHANTH, K. R. A comparative analysis of different nosql databases on data model, query model and replication model. In: **Proceedings of the 2013 International Conference on Emerging Research in Computing, Information, Communication and Applications**. [S.l.]: Elsevier Publications, 2013. ISBN 9789351071020.

VERA, H.; BOAVENTURA, W.; HOLANDA, M.; GUIMARAES, V.; HONDO, F. Data modeling for nosql document-oriented databases. In: **Proceedings of Annual International Symposium on Information Management and Big Data**. Andina University of Cusco, 2015. Disponível em: <http://ceur-ws.org/Vol-1478/SIMBig2015_proceedings.pdf>.

XIA, F.; LI, Y.; YU, C.; MA, H.; QIAN, W. Bsma: A benchmark for analytical queries over social media data. **Proc. VLDB Endow.**, VLDB Endowment, v. 7, n. 13, p. 1573–1576, ago. 2014. ISSN 2150-8097. Disponível em: <http://dx.doi.org/10.14778/2733004.2733033>.

ZABLOCKI, J. **Couchbase Essentials**. [S.l.]: Packt Publishing Ltd., 2015. ISBN 9781784394493.

APPENDICES

APPENDIX A – 95th Percentile Latency Charts

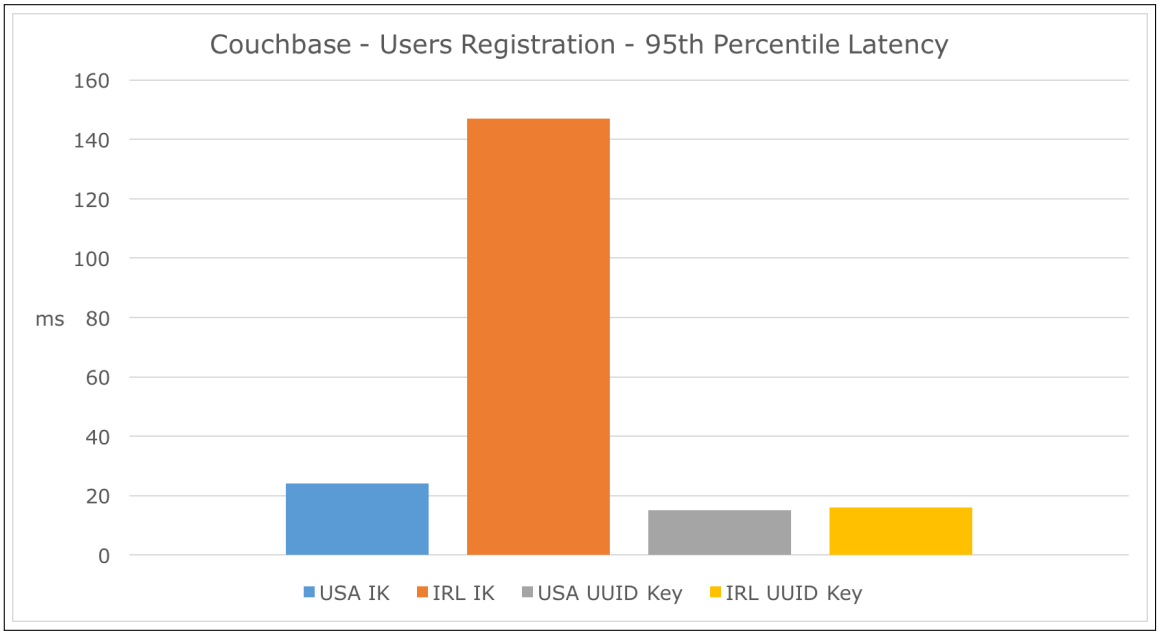
This appendix reunites the charts that demonstrate the comparison between the patterns approaches and their counterparts basic approaches regarding the 95th percentile latency metric, which for space issues were not displayed in the main text.

A.1 UUID KEY PATTERN

A.1.1 Availability Test

Figure 56 shows the comparison between the 95th percentile latency of the IK and UUID Key patterns regarding the concurrent submission of users registrations. Similar to the average latency (Figure 26), Figure 56 shows that the Ireland IK workload presents a high value for the 95th percentile latency, more than 140 milliseconds. On the other hand, both instances of the UUID Key workload present low 95th percentile latency values, less than 20 milliseconds, which is more than 7 times less than the Ireland IK workload.

Figure 56 – Comparison between Incrementing Key and UUID Key users registrations workloads 95th percentile latency.



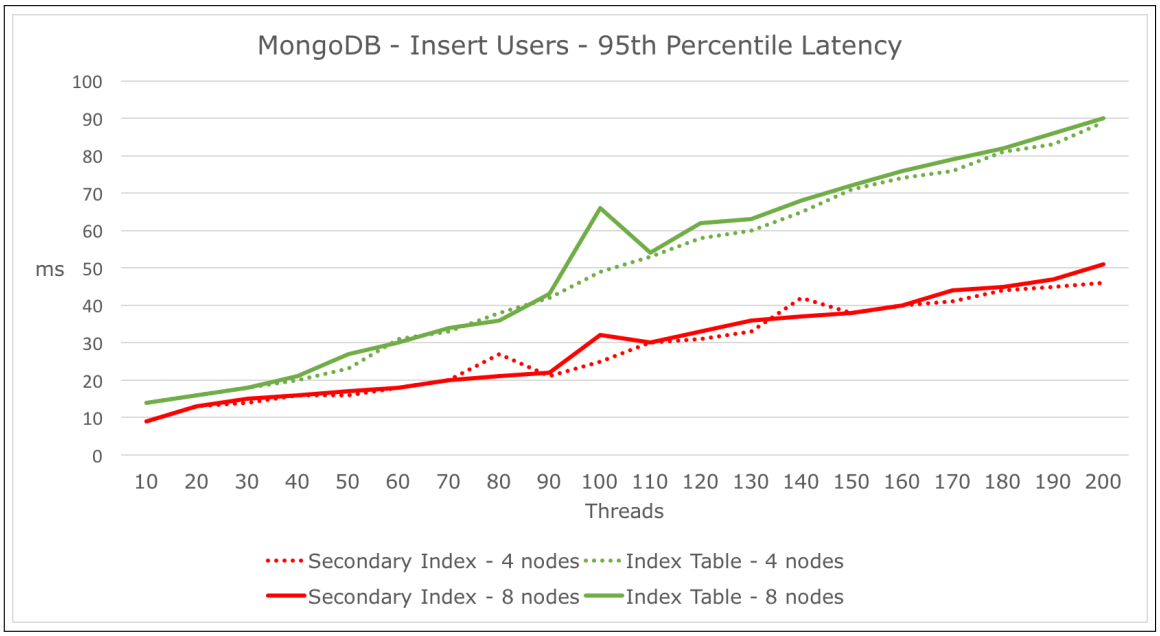
Source: Created by the author

A.2 INDEX TABLE PATTERN

A.2.1 Local Range Index and Index Table Pattern

Figure 57 shows the comparison between the 95th percentile latency of the Secondary Range Index and Index Table approaches regarding the concurrent insertion of users records. Figure 57 shows that, as for the average latency (Figure 30), the 95th percentile latency for both databases (Couchbase and Cassandra), is lower for the Secondary Index approach. However, the values shown in the chart are expected and acceptable. The superiority of the Secondary Index pattern is justified by the additional write request executed by the Index Table workload.

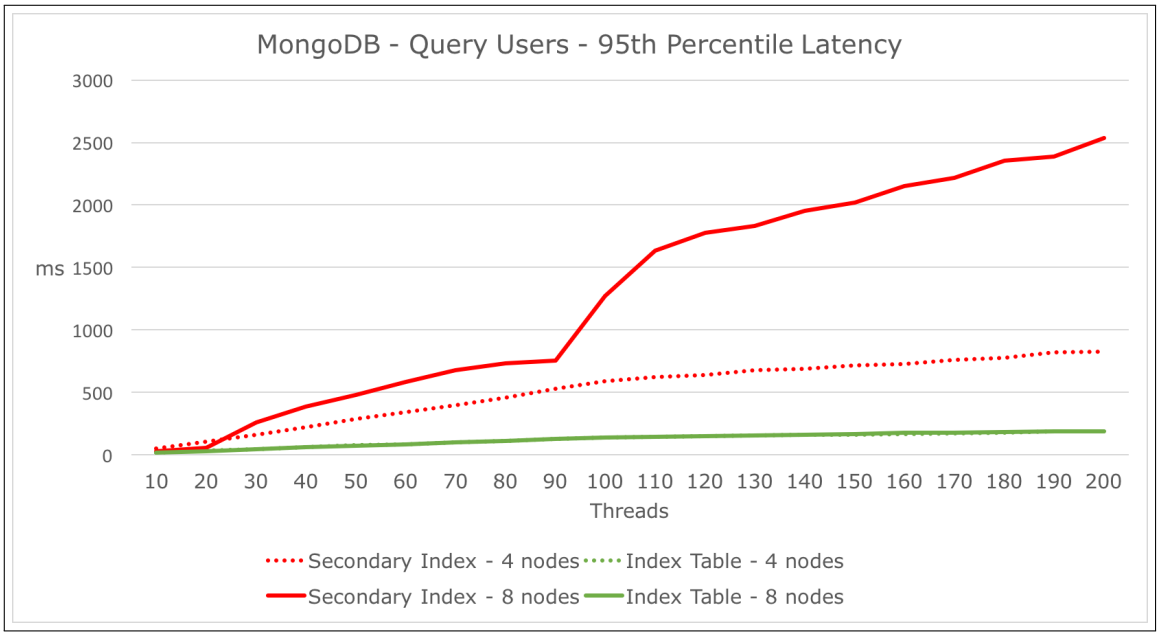
Figure 57 – Comparison between Secondary Index and Index Table insert users workloads 95th percentile latency.



Source: Created by the author

Figure 58 shows that for 95th percentile latency, the Secondary Index pattern presents a much more aggressive grow than the Index Table pattern when querying the users records. At 100 threads, in Cassandra, the Secondary Index workload presents a value about 6 times greater than the 95th percentile latency for the Index Table pattern. In Couchbase, the 95th percentile latency presented by the Secondary Index workload is about 20 times worse than the value presented by the Index Table pattern.

Figure 58 – Comparison between Secondary Index and Index Table query users workloads 95th percentile latency.



Source: Created by the author

A.2.2 Local Hash Index and Index Table Pattern

Figure 59 shows the 95th percentile latency when inserting user records for the Secondary Index and Index Table patterns for MongoDB clusters with 4 and 8 nodes. Both approaches have presented a linear and expected growth of the 95th percentile latency with the increase in the number of client threads.

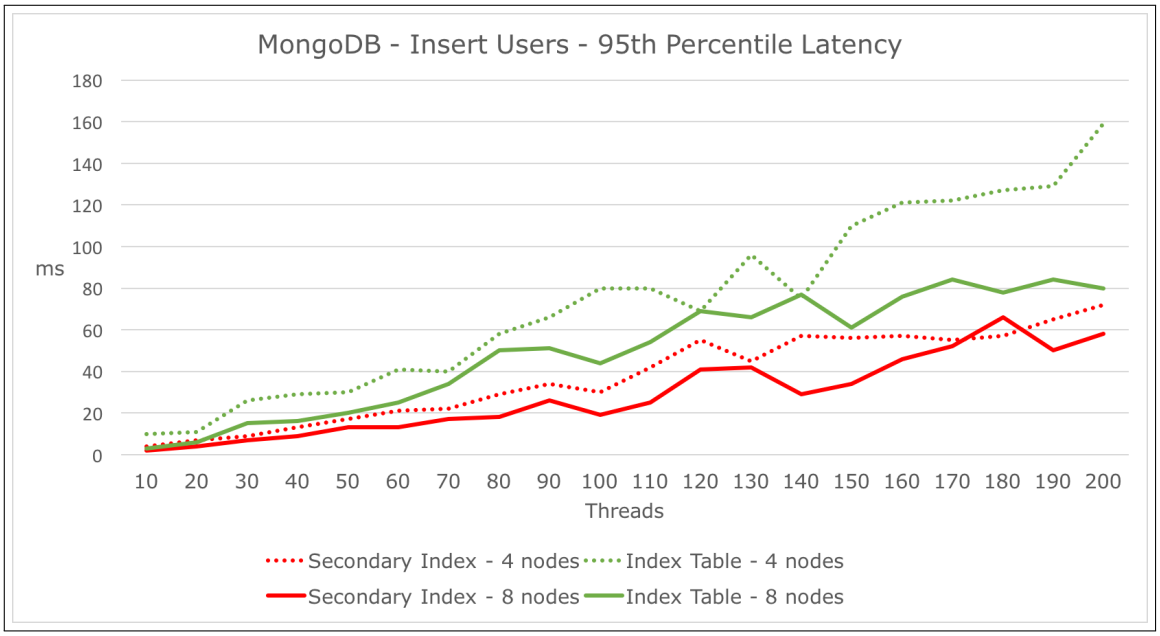
Figure 60 shows that when querying the user records, for the Index Table pattern, the 95th percentile latency has remained stable with the cluster growth. On the other hand, the 95th percentile latency has increased very sharply for the Secondary Index pattern. For the 8 nodes cluster, at 200 threads, the 95th percentile latency of the secondary index approach is more than 10 times greater than the 95th percentile latency of the Index Table pattern.

A.3 ENUMERABLE KEYS PATTERN

A.3.1 Write Concurrency Tests

Figure 61 shows the comparison between the 95th percentile latency of the LBA and Enumerable Keys patterns regarding the concurrent submission of multiple comments. The 95th percentile latency presented a behaviour similar to the average latency (Figure 44). At

Figure 59 – Secondary Index and Index Table 95th percentile latency with 4 and 8 nodes for insert users workloads in MongoDB.



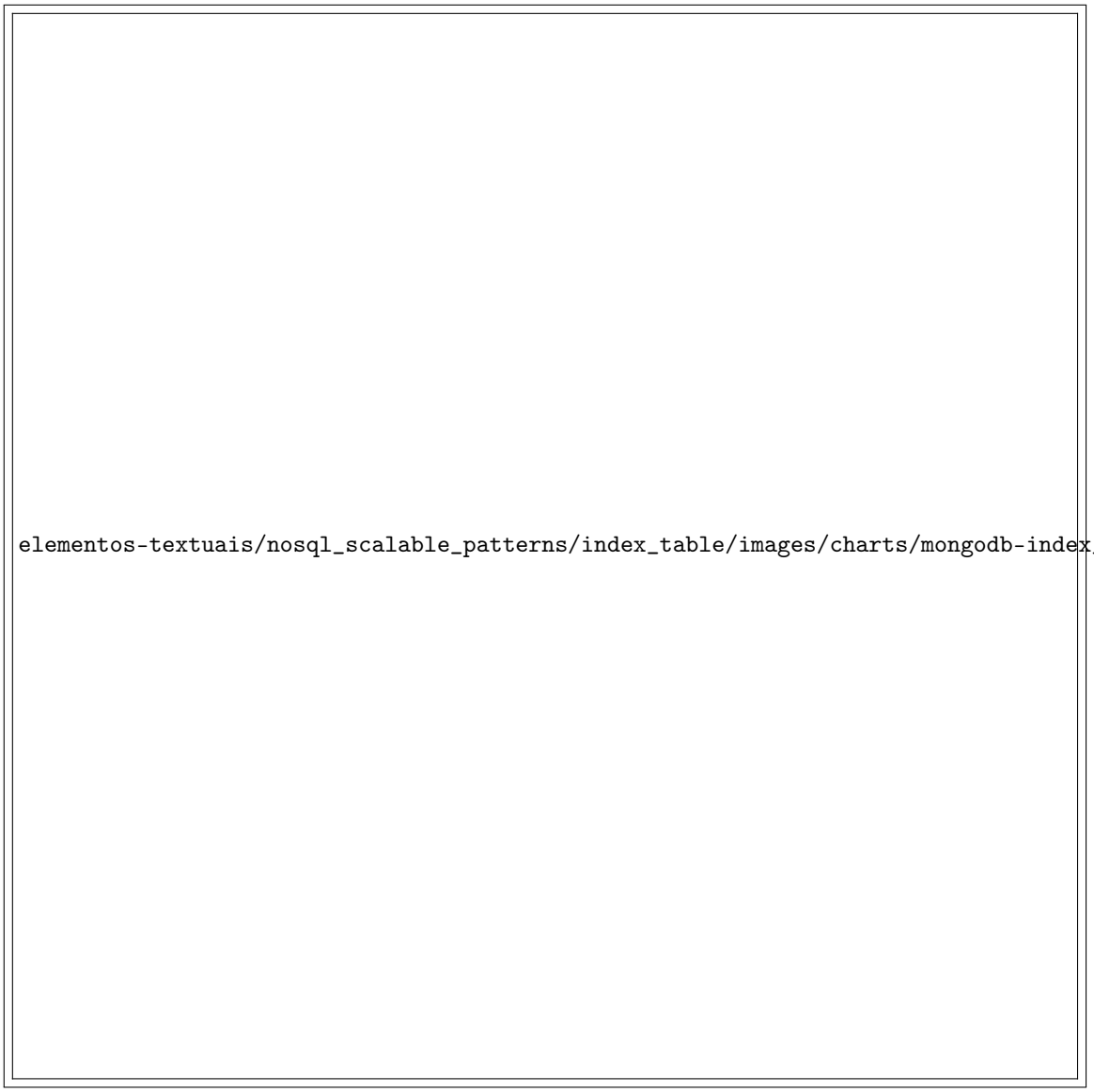
Source: Created by the author

100 threads, in Couchbase, the LBA pattern presents a 95th percentile latency value about 450 times greater than the value presented by the Enumerable Keys pattern. With MongoDB, at 100 threads, the LBA pattern presents a value about 70 times greater than the value presented by the Enumerable Keys pattern.

A.3.2 Data Volume Tests

Figure 62 shows the comparison between the 95th percentile latency of the LBA and Enumerable Keys patterns when the volume of data increases over time. For the LBA pattern, the 95th percentile latency increases according to the number of comments stored in the list, while for the Enumerable Keys pattern it remains stable. At the end of the test, with more than 9000 documents already stored, the 95th percentile latency is about 40 times bigger for the LBA pattern compared to the Enumerable Keys pattern in Couchbase, and about 4 times bigger in MongoDB. The 99th percentile latency for MongoDB, shown in Figure 63, justifies the fact that the 95th percentile latency presents a lower value than the average latency (Figure 46) from 4000 documents onwards

Figure 60 – Secondary Index and Index Table 95th percentile latency with 4 and 8 nodes for query users workloads in MongoDB.

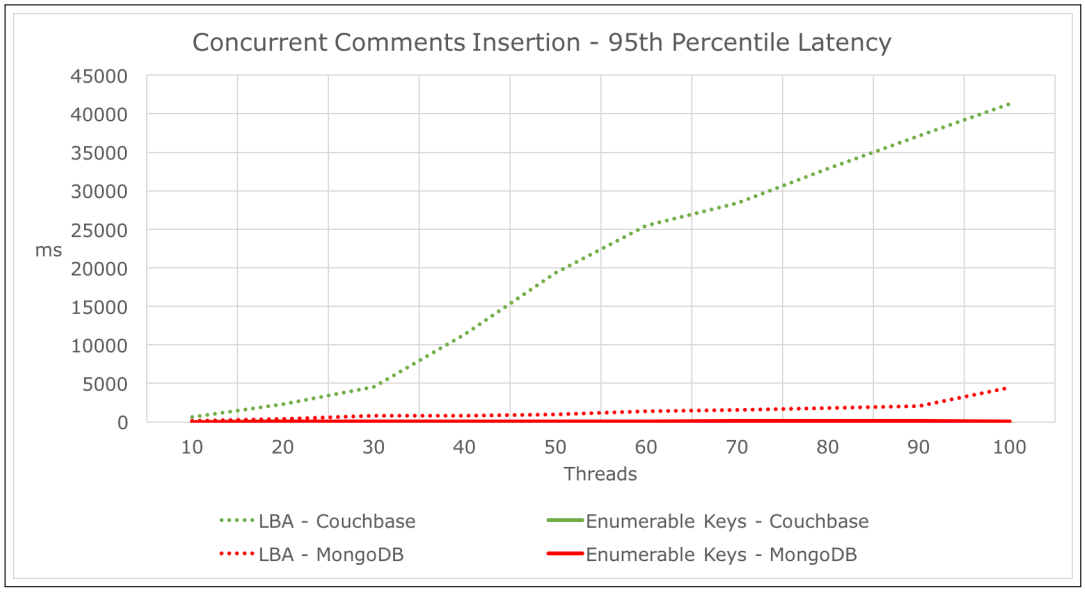


Source: Created by the author

A.3.3 Data Retrieval Tests

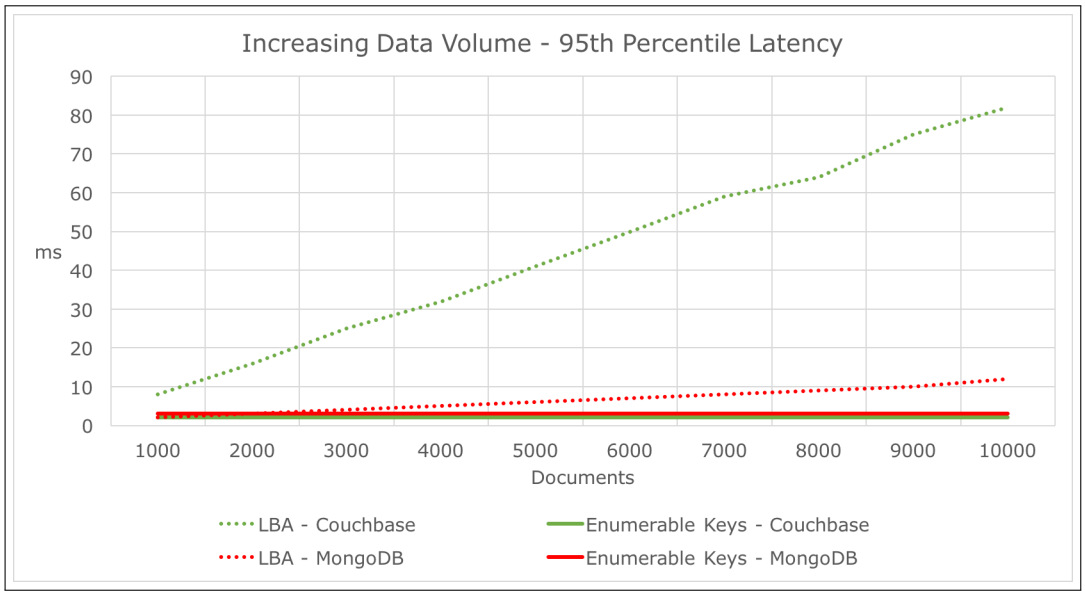
Figure 64 shows the comparison between the 95th percentile latency of the LBA and Enumerable Keys patterns regarding the concurrent pagination of comment documents. Like the behaviour of the average latency (Figure 48), for read operations the Enumerable Keys pattern provides satisfactory 95th percentile latency.

Figure 61 – LBA and Enumerable Keys concurrent comments insertion workloads 95th percentile latency.



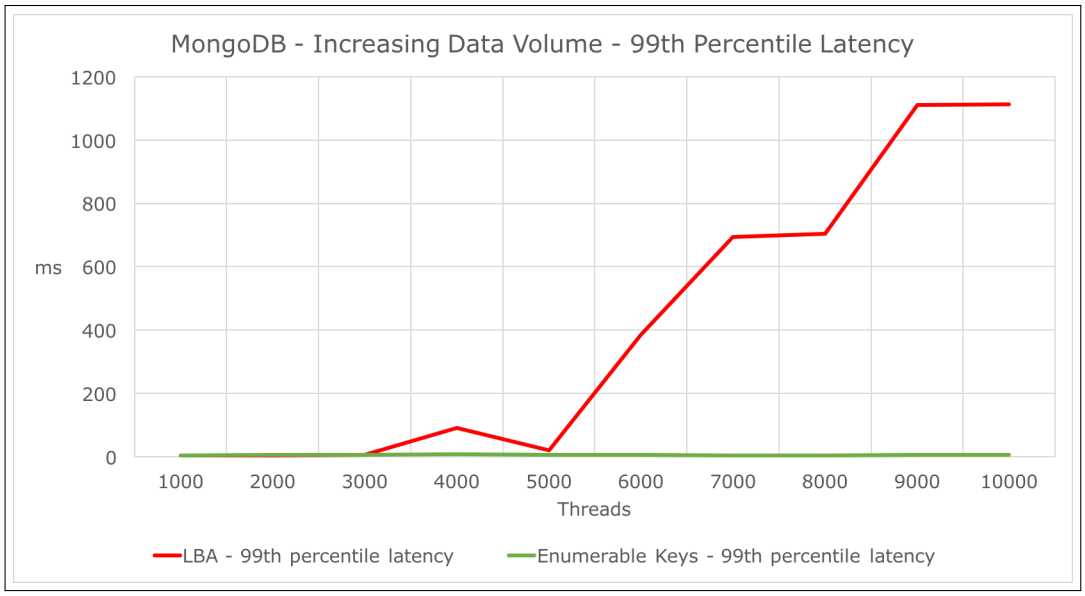
Source: Created by the author

Figure 62 – LBA and Enumerable Keys increasing data volume workloads 95th percentile latency.



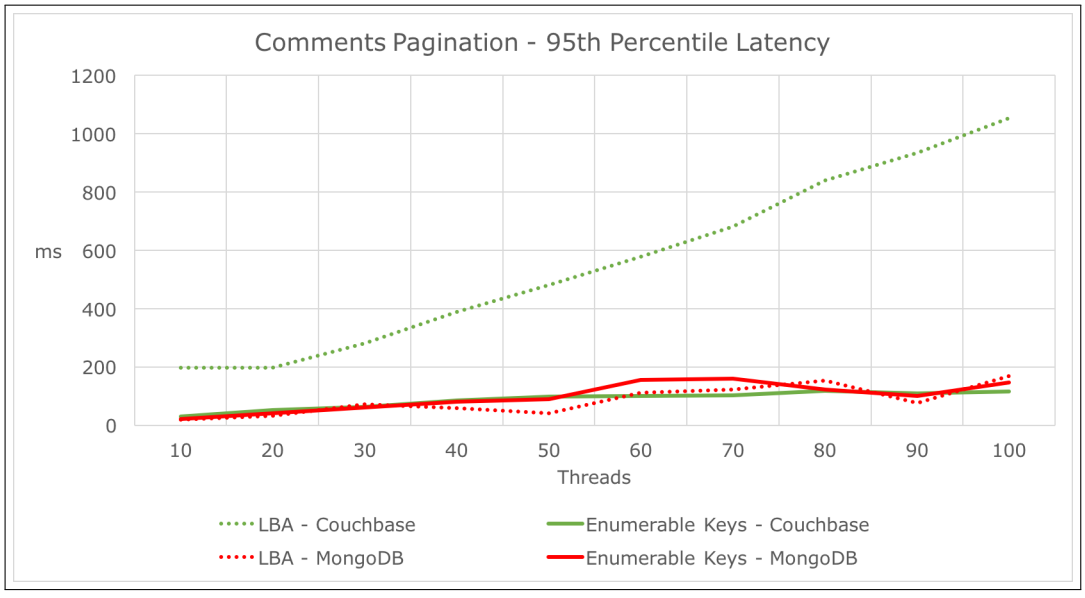
Source: Created by the author

Figure 63 – LBA and Enumerable Keys increasing data volume workloads 99th percentile latency for MongoDB.



Source: Created by the author

Figure 64 – LBA and Enumerable Keys comments pagination workloads 95th percentile latency.



Source: Created by the author