POLITECNICO DI MILANO

Scuola di Ingegneria dell'Informazione
Corso di Laurea Specialistica in Ingegneria Informatica
Dipartimento di Elettronica, Informazione e Bioingegneria

Extending NoSQL to Handle Relations in a Scalable Way

Models and Evaluation Framework

Advisor: Elisabetta DI NITTO
Co-Advisor: Santo LOMBARDO

Master Thesis by:
Oscar LOCATELLI - 745939

Academic Year 2011/2012

Dedico questa tesi alla mia famiglia e ai miei amici più cari.

Alle persone che mi sono sempre state vicino.

Le persone che con me hanno condiviso gioie e malumori in questi anni.

Le persone che sempre lo faranno.

Dedico questa tesi anche a me stesso per non essermi rilassato quando tutto sembrava facile e per non aver ceduto davanti alle difficoltà.

Ringraziamenti

Ringrazio la relatrice di questa tesi Elisabetta Di Nitto per avermi affidato questo lavoro, avermi aiutato fino all'ultimo e per avermi fatto respirare un po' di ricerca nel contesto italiano. Ringrazio il correlatore Santo Lombardo per aver condiviso le sue idee con me, per aver creduto in me e per aver discusso con me quando non eravamo d'accordo. Ringrazio il professor Danilo Ardagna, tutti i ragazzi e le ragazze del dipartimento e gli altri tesisti per aver condiviso con me il loro tempo.

Ringrazio tutti quelli che mi hanno accompagnato durante questa esperienza al Politecnico di Milano. I miei compagni di studio Paolo Moretti e Tania Suarez.

Ma soprattutto ringrazio tutte le persone che mi hanno dato la forza per sopportare lo sforzo finale, in ordine alfabetico Aurelia, Caterina, Erika, Luca, Simone, Stefania.

Grazie

Milano, 2 Aprile 2013

Oscar

Table of Contents

List of Figures

List of Tables

List of Algorithms

Abstract

With Web 2.0 growth of the amount of data accessible on the web has occurred. We live in the Big Data, Big Users, and Cloud Computing era. In this situation, storage solutions such as RDBMS showed their limits concerning the scalability over multiple nodes. Storage systems known as NoSQL databases are becoming increasingly important because are designed to scale well. However, they are not the solution to every problem of data management. In fact, the lack of standardization of the currently available NoSQL implementations, force developers to handle low-level data management issues thus resulting in a higher complexity of programming NoSQL compared to RDBMS solutions. A challenging research objective is therefore to improve the programmability and manageability of NoSQL still keeping their remarkable characteristics in terms of scalability and capability of handling large volumes of data. This thesis aims at offering a contribution toward the achievement of this challenge. In particular, we focus on how to render relations between entities in a NoSQL still avoiding the need of introducing join operators that would impair their scalability. We study two different approaches for doing so, called MinR and MaxR. MinR minimizes data replicas while MaxR maximizes replicas. Also, we will present a rigorous methodology to compare the two strategies. We will expose and motivate a series of tests designed to investigate interesting aspects of the two techniques and will provide the results of the test run using two different types of NoSQL database, MongoDB (a Document-based) and Cassandra (Column-oriented). The main conclusion is that MaxR better then MinR with sparse relations. Also, MaxR is the right choice to frequently read small amounts of data. MinR is good to often maintaining consistency. But from numbers we have also concluded, for example, that for Cassandra usually MinR is better than MaxR and, viceversa, for Mongo the MaxR model is the right choice. Finally, we present an evaluation framework developed for the execution of the tests. The framework, developed in Microsoft .NET, is easily extensible to use with other databases and add more tests.

Estratto

Il web 2.0 ha avuto come effetto una considerevole crescita della mole di dati fruibile sul web. Viviamo nell'era del Big Data, del Big Users e del Cloud Computing. In questa situazione, soluzioni di storage tradizionale quali gli RDBMS mostrano i loro limiti quando è necessario utilizzare sistemi distribuiti. Sistemi di storage conosciuti con il termine NoSQL stanno diventando sempre più importanti. Essi però non sono la soluzione ad ogni problema di gestione di dati distribuiti. Infatti, la mancanza di meccanismi standard nelle soluzioni esistenti pone lo sviluppatore di fronte a problemi di basso livello nella gestione dei dati che ne complica la programmazione. Un obbiettivo di ricerca interessante è migliorare questi aspetti che sono carenti nei NoSQL, pur mantenendo la loro notevole propensione alla scalabilità. Questa tesi vuole offrire un contributo in questo verso. In particolare ci focalizzeremo sull'introduzione del concetto di relazione tra entità nei NoSQL senza l'utilizzo delle operazioni di join, punto debole della scalabilità. Proponiamo due tecniche alternative per questo, MinR e MaxR. MinR minimizza le repliche dei dati, MaxR le massimizza. Presenteremo anche una metodologia rigorosa per confrontare i modelli. Esporremo e motiveremo una serie di test progettati per valutare gli aspetti più interessanti delle due tecniche e riporteremo i risultati di questi test ottenuti eseguendoli su due differenti database NoSQL: MongoDB (un Document-based) e Cassandra (un Column-oriented). Il risultato principale sarà che MaxR è migliore se utilizzato con relazioni sparse e se si necessità di frequenti letture di piccole quantità di dati. MinR è migliore se si vuole mantenere la consistenza. Dai risultati dei test si può anche concludere che per Cassandra è più adatto il modello MinR e viceversa Mongo si comporta meglio utilizzando MaxR. Verrà infine presentata una piattaforma di test (Evaluation Framework) per l'esecuzione dei test. Il framework è sviluppato nell'ambiente Microsoft .NET ed è facilmente estendibile con nuovi test e per l'utilizzo con diversi database.

Chapter 1

Introduction

Web is dramatically changing in the last 15 years. The advent of web 2.0 has resulted in a considerable growth of the amount of accessible data. This is due to the birth of services where the user has an active role and contributes to the production of content (social network, public-opinion questionnaires, etc.). All these data are important to monitor users tastes and habits, so companies are increasingly interested in tracking them. Today we are in the Big Data, Big Users, and Cloud Computing era. The result is a huge quantity of heterogeneous data to be acquired, stored, analyzed and transformed.

In this situation, storage solutions such as traditional RDBMS have showed their limits when the systems must be distributed over many nodes. Furthermore, the rigidity of Relational Databases structure is not very suitable to handle heterogeneous or not structured data.

Storage systems known as NoSQL databases are becoming increasingly important. They are data store systems that are not classifiable as the classical RDBMS (Relational Database Management System). NoSQL means "Not SQL" for some and "Not only SQL" for some others. The controversy lies in the fact that some researchers think that NoSQL databases should fully replace RDBMS while some others think they should only assist them. Our opinion is that each application case shows its own peculiarities and that sometimes NoSQL or RDBMS alone would be the right solution while some other times they should be used together.

NoSQLs are simpler than RDBMSs, they do not provide many of the complex data management mechanisms that are provided by RDBMS systems, thus, they are the best choice to store data that do not need this complexity.

In most of cases NoSQL are developed from the ground up to be distributed, scale out databases. This is a direct consequence of the independence between the different collections of objects they store. In a NoSQL database there is no concept of relation between different objects. So, they are independent. There are no constraints between objects and so it is very simple to split the database in many parts and distribute them over a network of nodes. Furthermore NoSQLs are best suitable to store heterogeneous and unstructured data, since they have a flexible schema (schema-free, or semi-structured schema).

However, NoSQL databases are not the solution to every problem of data management. In fact, they completely miss a common query language like SQL in RDBMS. SQL is based on Relational Algebra, a strong mathematical theory that ensures completeness of the query language and that offers many optimization techniques to support query execution. It represents one of the main reasons why the RDBMS systems have acquired increasing importance. A developer can move from one database system to another with reduced effort at least for basic operations.

Another lack of NoSQLs is the extreme heterogeneity of the existing solutions for what concerns the organization of the data model, the query model and the data access recommended patterns. This forces the developer to handle manually low-level data management issues like indexing, query optimizing data structures, relations between objects, and so on. This results in a higher complexity of NoSQL compared to RDBMS solutions for what concerns programmability and management of the data store.

A challenging research objective is therefore to improve the programmability and manageability of NoSQLs still keeping their remarkable characteristics in terms of scalability and capability of handling large volumes of data. This thesis aims at offering a contribution toward the achievement of this challenge. In particular, we focus on how to render relations between entities in a NoSQL still avoiding the need of introducing join operators that would impair their scalability. We study two different approaches for doing so, called MinR and MaxR.

The work presented in this thesis is part of research project called Marijà. In particular, the aim of the thesis is to present and compare MinR and MaxR as the two logical models to manage the relationships between objects on a single node of the Marijà system. We analyze the single node because we start from the Marijà project assumption of perfect scalability of a query on a network of nodes and the full independence of a node from each others.

The models are at two opposite ends of the replication concept. MinR minimizes data replicas while MaxR stores a copy of entity data in each relation row to reduce the number of database requests. MinR is very similar to the way of storing many-to-many relations of the relational model. The difference is that the same strategy is used to store all kind of relations. Also, there are no database constraints on the entities existence. Objects remain independent from the database side and then scalability profile does not change. MaxR instead provides all relation data within the same row of the relation collection. Data are ready to be read but replicas must be kept in sync and the quantity of data to transfer is bigger than in the MinR strategy.

We had to compare models and we decided to do it creating algorithms of the basic CRUD operations on relations. For each identified CRUD operator we create the algorithms fto execute them on both MinR and MaxR models and we extract a cost function. In future these functions will be traduced in a set of cost metrics that will be used to choose the correct logical model for a particular relation.

Analyzing cost functions we designate a set of experiments aimed to compare models with respect of relation properties and evaluate overall performance of the single node.

Tests are executed on two very different concrete NoSQL systems: MongoDB, a Document-based database that maximize performance of the single machine and Apache Cassandra, a BigData Column-oriented database that works well with a wide network of nodes. Results of these experiments are discussed in details and we report the conclusions that we have reached from these.

The tests are executed with the Evaluation Framework, developed for this thesis, that consists of the Marijà Worker classes and the Testes classes.

The Marijà Worker is responsible to setup database and perform CRUD operations on the database. The framework is extensible to support a new database, a developer must implement the Database Provider, that consists of three interfaces, one for database configurations and the others for implementing the algorithms for MinR and MaxR models.

The Tester is the benchmark platform. New tests can be implemented and then added in the Tester to be executed. Tests already implemented are those that we describe in the thesis.

The Marijà Worker is to be deployed on the Marijà node as low level executor. Also the Tester is to be deployed on the node and it will use the functionality of Marijà Worker to perform benchmarks.

Analyzing the results of testing it is possible to choose which model is best suited to store a given relation and what is the maximum amount of data that the node can handle (the sharding point). This is the first information needed to configure a Marijà node.

Original Contributions

This work includes the following original contributions:

- The definition of two common logical models for relation handling in NoSQL databases that maintain a high scalability profile. The two models are meant to be used together in the same database choosing for each relation what is the best one.
- A complete way for comparing these models in concrete NoSQL implementations with the following two main objectives: choosing the best strategy to store a relation with respect its properties and estimating the maximum quantity of data to store in a single node (before to operate sharding) that is compliant with some performance constraints.
- An extensible Evaluation Framework to benchmark a Marijà node with an extensible architecture that allows to add custom tests and implement, with minimal effort, new database provider (to support different databases).

Outline of the Thesis

This thesis is organized as follows:

- In **Chapter 2** we describe the state of the art. First, the reason for using NoSQL are exposed. Then we present a classification of existing products with respect to their data model and main elements characterizing NoSQL. Finally, we discuss in details the difference of data model classes and the most representative products for each class.

- In **Chapter 3** we define the problem of introducing relations in the serialized world of NoSQL. Then we report a small overview of the Marijà project and the assumption of single node independence from which this thesis is based.

- In **Chapter 4** we present the two alternative models to store relations on a single Marijà node, MinR and MaxR. Then we propose a rigorous methodology to compare the models based on CRUD operations. We present the idea of using the Relational Algebra as bridge model to obtain a complete Read comparison. Finally, we describe the algorithms of the CRUD operators on both models and we discuss the differences.

- In **Chapter 5** we present what tests are to be performed in the case of concrete NoSQL products and how they are performed. We report the results of preliminary tests to validate the theoretical differences found in previous chapter. Then we expose and motivate a set of detailed tests designed to investigate interesting aspects of the two models and we provide the results of the test runs using two different types of NoSQL databases, MongoDB (a Document-based) and Apache Cassandra (Column-oriented).

- **Chapter 6** is dedicated to the presentation of the evaluation framework developed for the execution of the experiments. The framework, developed in Microsoft .NET, is easily extensible for use with other databases.
- In **Chapter 7** we summarize the main results we have achieved and we describe our personal experience in using NoSQLs. Finally we present some planned future work.

Chapter 2

State of the Art

2.1 Introduction

In this chapter the main characteristics of NoSQL databases together with the background of theories and techniques behind them will be introduced. Then a classification of NoSQL databases from the data-model perspective will be presented and discussed with a special focus on the feature-set the most representative implementations provide.

Some important aspects, like security, costs and others that are not interesting for our purpose and will not be treated.

2.2 Why NoSQL

There are some interesting features that are common to most of the NoSQL databases:

- **Designed for Scalability**: as we will describe in next section, the NoSQL way to store data is perfect for scaling out on many nodes.

- **Schema Flexibility**: the data schema is not fixed as in RDBMS. It can efficiently store heterogeneous objects and there is no need of complex migration to evolve the structure. NoSQL schema can be referred as no-schema, schema-free or semi-structured schema depending of the NoSQL data model category. These terms will be explained in data model descriptions.

- **NoSQL as a cache of RDBMS system**: NoSQLs store objects and the objects are identified by a key. This is very similar to what is done by a cache. NoSQLs can be used as a cache over a traditional db (cf. Redis and Memcached in Section 2.4.1).

- **Caching Facilitie of NoSQLs**: some NoSQLs integrate complex caching mechanisms to enhance overall performance. They use in-memory data structures to decrease the latency of operations and then flush many operations together to a durable storage.
- **High Availability Oriented**: as we will show in the next sections, most of the NoSQL databases, at least the Big Data Oriented, has the High Availability as the main objective. To provide this they do not guarantee all ACID properties, but use the a relaxed set of properties named BASE (Basically Available, Soft state, Eventual consistency).
- **Cloud Oriented Database**: NoSQL databases are becoming increasingly important because they are used largely in the Cloud Infrastructure as elastic storage. Elastic means that the number of used nodes are auto-adaptive and this is simpler to do with NoSQLs than with RDBMS, because of all other points described in this section.
- **MapReduce Features**: Some NoSQLs integrate the possibility to distribute queries and other operations over nodes using a MapReduce algorithm. MapReduce is described in Section 2.3.2.

NoSQL in different scenarios There are many scenarios in which NoSQL databases fit well. Each NoSQL is suited for some scenarios, but not for all. The main scenarios are:

- **Enterprise Big Data NoSQL**: These systems are built to handle a very huge amount of data and have as main objectives high availability, scalability and elasticity [20]. Commonly they are not only database systems but include at least an optimized method to access the file system or even they are full infrastructures configured for replication and migration of data. The main representatives of this category are Amazon DynamoDB and Google BigTable as full services offered in Amazon Web Services and Google AppEngine cloud ecosystems respectively and Cassandra or Hbase over Hadoop as systems that can be deployed everywhere.
- **Caching NoSQL**: In-memory NoSQL databases like Redis and Memcached are perfect for creation of a global cache layer on top of a distributed database (relational or not) to speedup reads and writes of entities loaded in memory.

- **RDBMS Replacement NoSQL**: In this category I place the NoSQL databases that provide some features of RDBMS that enable their use in a general purpose scenario (where RDBMS win). Some of the characteristics we refer to are: manual or automatic indexes for a column (or attribute in a document context), complex query model, strict consistency, transactions, triggers and so on. The candidates are MongoDB and RavenDB that are document-based and some others like Scalaris that can support full ACID properties.

- **Specialized scenario NoSQL**: Some implementations have unique features that enable special scenarios, for example CouchDB has an out-of-the-box online-offline synchronization mechanism that enable a two-way automatic synchronization between more databases that were disconnected (network partition) and updated.

In the Table 2.1 we classify the some NoSQL products in different scenarios.

Scenarios	**Matching Databases**
Enterprise Big Data	DynamoDB Project Voldemort Google BigTable Cassandra Hbase Hypertable
Local or Global Caching	Memcached Redis
RDBMS Replacement	MongoDB RavenDB Scalaris
Specialized scenario	CouchDB

Table 2.1: Classification from the Scenario perspective

2.3 Classification dimensions for NoSQL

2.3.1 Categories of NoSQL from the data model perspective

The classification of existing NoSQL databases is not a trivial task. They have different architectures and features and each of them is suitable for a specific use. Maybe the most immediate approaches are to compare them from the usage scenarios and from the data model perspective.

A rich classification of NoSQL Data Models is presented by Stephen Yen (cf. [44]) and another less detailed one is introduced by Rick Cattel (cf. [16]) and it is summarized in Table 2.2. Both classifications are good, but lack an important category, the graph-based databases. We have adopted a classification similar to the one by Cattel but augmented with a new type of category, the Graph-based one (cf. Table 2.3).

Our Categories	**Cattel Categories**	**Yen Categories**
Key-Value	Key-value Store	Key-Value-Cache Key-Value-Store Eventually-Consistent K-V-S Ordered-Key-Value-Store Data-Structures Server
Document-based	Document Stores	Document Store Object Store
Column-oriented	Extensible Record Stores	Wide Columnar Store Tuple-store
Graph-based		

Table 2.2: Comparison of my taxonomy with the ones by Stephen Yen and Rick Cattel

2.3.2 Main Elements Characterizing NoSQL

In the next sections these categories will be explained and, for each of them, some of the major implementations will be described and summarized in following key points:

- **Data-model Category (Data Model)** with special variations
- **CAP Theorem and Eventual Consistency (CAP)**: CAP Theorem choices (CP or AP) and related notes on consistency

Category	**Matching Databases**
Key-Value	Amazon DynamoDB Project Voldemort Memcached Redis Scalaris
Document-Based	CouchDB MongoDB RavenDB
Column-Oriented	Bigtable Hbase Hypertable Cassandra
Graph-Based	Neo4j

Table 2.3: NoSQL Categories Database Matching

- **Data Partitioning and Scalability (Partitioning)**: it is comprehensive of both horizontal (sharding) and vertical auto partitioning of data to scaling on more commodity machine
- **MapReduce**: MapReduce implementation and server-side code execution capabilities
- **RDBMS Related Features (RDBMS)**: the common mechanisms that is present in all RDBMS and that are implemented in the analyzed NoSQL such as principal and secondary indexes, view materialization, triggers, transactions, ...
- **Architectural**: notes on the architecture of the analyzed NoSQL like, in instance, the distribution on a network of node or the existence of a single point of failure in the system
- **Other**: other interesting features

The *Data-model Category* point is already introduced in previous paragraphs and it will be explained in next section. *CAP Theorem and Eventual Consistency*, *Data Partitioning and Scalability*, *MapReduce*, *RDBMS Related Features* points are explained in details in the following paragraphs. *Architectural* and *Other* points do not need ad additional description.

Data Partitioning and Scalability

As we introduced in the Chapter 1 the RDBMSs have a limited scalability and they are not well suited for storing unstructured or semi-structured data.

In details, to compose a relation in the RDBMSs, one or more *join* operations must be performed and this is why the RDBMSs does not scale well in a distributed context. Executing join between data located in different nodes involves a huge data transfer that grows with the growth of nodes number, breaking down the scalability. Some optimizations exist, like "Semi-Join" algorithms [1] but the problem is that significant amount of data between different nodes should still be compared.

NoSQL databases were developed to be distributed, scalable databases. The main concept of NoSQL (with exception of Graph-oriented, cf. Section 2.4.4) is the independence of each stored data object from everyone else. Data are serialized in collections and there are no connections from an object to another. So, partitioning data and getting a scalable database is extremely easy.

Data can be partitioned horizontally and/or vertically with respect to the data model properties (the way to store data).

Sharding is an horizontal partitioning of data in a database. Each individual partition is referred to as a *shard* or *database shard*.

An hashing algorithm on keys is used to split data in more shards, this hash can be optimized to balance the load on nodes (random partitioner) or to preserve keys order, enabling range queries (order preserving partitioner).

In the Figure 2.1 we show an example of sharding on a Key-Value structure, that will be presented in Section 2.4.1. Here we show that different Key-Value pairs can be stored on different storage nodes.

Vertical partitioning is the alternative to sharding. It consists in dividing data by columns. For instance, Figure 2.2 shows that the first two columns of all tuples are stored in node 1 while the last three are stored in node 2.

Horizontal and vertical partitioning can be applied together in particular cases, like in Column-Oriented NoSQL databases as shown in Section 2.4.3.

CAP Theorem and Eventual Consistency

[1] the semi-join requires more operations to be performed on local nodes, however the data transfer rate is reduced [18]

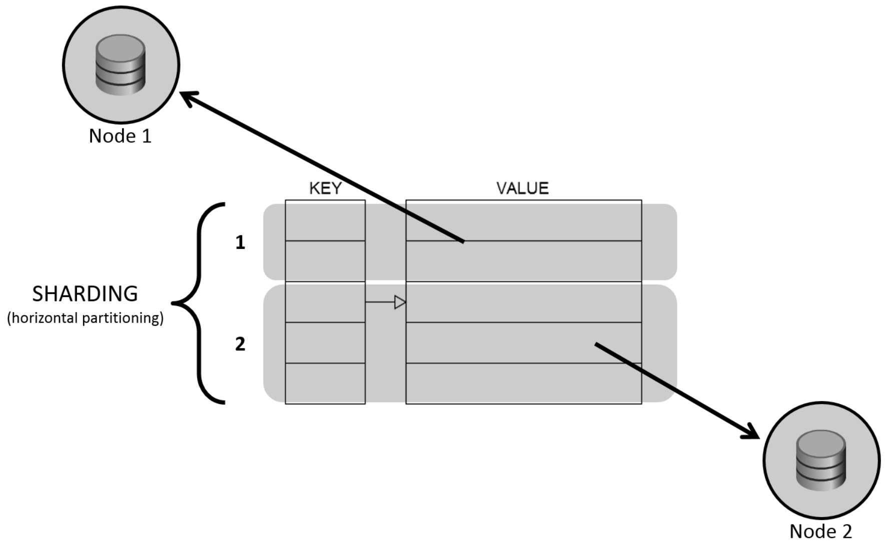


Figure 2.1: Data Sharding: horizontal data partitioning based by key

Brewer's CAP Theorem Most of NoSQLs are distributed systems and we cannot present them without a brief discussion about the CAP theorem.

The idea on which the theorem is based was introduced by Eric Brewer in 2000 [15]. This conjecture has been formalized in 2002 by Seth Gilbert and Nancy Lynch [30].

The CAP theorem, also known as Brewer's CAP theorem, states that it is impossible for a distributed computing system to simultaneously provide all three of the following guarantees: Consistency, Availability and Partition Tolerance (from these properties the CAP acronym has been derived).

- Consistency, all nodes see the same data at the same time
- Availability, a guarantee that every request receives a response about whether it was successful or failed
- Partition Tolerance, the system continues to operate despite arbitrary message loss or failure of part of the system that create a netwrok partition

Only two of the CAP properties can be ensured at the same time. The resulting three possible combinations are denoted with CA, CP, AP and for many people CA and CP are equivalent because loosing in Partitioning Tolerance means a lost of Availability when a partition takes place.

So, the NoSQL databases, acting as distributed systems, must choose between either support: AP or CP. In the products overview presented in

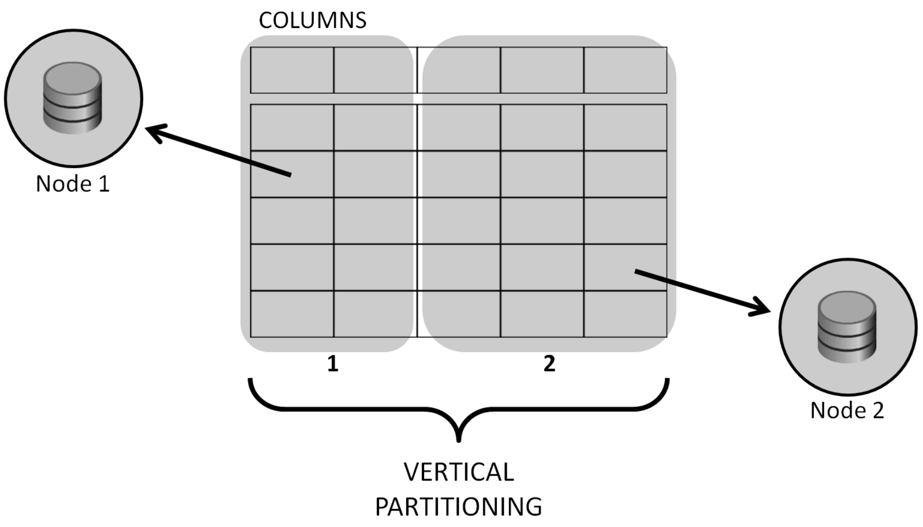


Figure 2.2: Data Vertical Partitioning: vertical data partitioning by columns

the next sections we will analyze this choice for each of the described NoSQL.

Eventual Consistency and BASE Properties The main objective of big NoSQL distributed infrastructures is to ensure high availability even in the presence of a very high number of fails in the distributed system.

ACID properties provide the consistency choice for partitioned databases. Ensuring consistency and partition tolerance means loosing of availability (cf Section 2.3.2). So ACID is not the correct choice for these systems.

The answer is BASE (Basically Available, Soft state, Eventual consistency) [39]. In shorts BASE is an optimistic way to see consistency. For an interval of time data can be not consistent, but after this interval all returns to consistent. This is the Eventual Consistency concept. We explain this concept with an example: if we have more copies of the same data in a distributed database and we want to update them, in the ACID way all the replica are logically written together. All replicas are in old state before the execution of the update and all are in new state after this. In the BASE way instead, there is a time interval in which some replicas are updated and some others are still in the old version.

To show this behavior a BASE system do not use two-phase locking (or its distributed version). However, it must ensure that sooner or later all data return to be consistent (Basically Available).

MapReduce

MapReduce is an emerging solution to distribute algorithms over a large network of nodes.

MapReduce is not new in the context of distributed computing, but it has been re-discovered brought back to the scene thanks to Google. In [27] Google presents the Map Reduce idea and its own implementation. This is assumed to run on a large cluster of machine. In [27], we read *"a typical MapReduce computation processes many terabytes of data on thousands of machines"* and *"Programmers find the system easy to use: hundreds of MapReduce programs have been implemented and upwards of one thousand MapReduce jobs are executed on Google's clusters every day"*.

Google had the merit to demonstrate that a "simple" infrastructure composed of several clusters combined with a MapReduce paradigm is the right way of looking at big data problems.

Influenced by the Google experience, an Apache project started with the idea to realize an open-source product similar to the Google one. As a result, Hadoop [14] has been released. It is a complete framework for distributed storage and processing of large data sets across clusters. It is mainly composed of:

- Hadoop Distributed File System: A distributed file system
- Hadoop YARN: Job scheduler and cluster resources manager
- Hadoop MapReduce: Hadoop MapReduce environment

Moreover, it includes many other projects 100% compliant with the basic architecture.

Nowadays, Hadoop has became the open source solution used by private users as well as by big ICT company. For instance, Amazon offers VMs with complete Hadoop solution, Cloudera offers Cloud solution based on Hadoop and also Microsoft dismissed Dryad, its research project for writing parallel and distributed programs, to support Hadoop.

In the NoSQL context MapReduce is very popular. Google use it in Big Table to distribute the work on the nodes. HBase and Hypertable do the same with the underlying Hadoop layer. Also Cassandra can be integrated with Hadoop. MongoDB use it to distribute query an any other task. CouchDB use it for querying and for creating and updating its indexes. RavenDB use it only for indexing.

MapReduce is on the way to become the standard framework of algorithm distribution.

RDBMS Related Features

As we described in Chapter 1, an important issue in NoSQL is the lack of a common query and manipulation language valid for all NoSQL databases, like SQL is for RDBMS.

Furthermore, the RDBMS systems provide a very high number of mechanisms which are absent or nearly so in NoSQL:

- **Indexing**: Database index is a data structure that improves the speed of data retrieval operations at the cost of slower writes and the use of more storage space. In details all row keys of a table are organized in an read-efficient structure, such as a tree or hash map

- **Views**: View is a database object that contains definition of a complex query created with a combinations of many tables. Sometimes views can be materialized that means they also contain the results of these queries.

- **Transactions**: Transaction comprises a unit of work performed within a database management system and treated in a coherent and reliable way independent of other transactions. All operation in a transaction have to be completed or else all of them are to be canceled.

- **Triggers**: Database triggers are programmed tasks that is automatically executed in response to certain events and that work on data

2.4 Classification of NoSQL

2.4.1 Key-Value Databases

The Key-Value approach is a simple and widely used paradigm to get data, from key-value stores to caching systems, where data are represented in form of a map between keys and values.

The main benefit of this model is that read and write access to values can be easily optimized to gain top level performance (with BTrees or distributed hash tables). Moreover, it is very simple to split data horizontally (sharding) and store them on multiple machines when the number of records grow. (cf. Section 2.3.2) Other categories can be treated as an evolution of this basic model.

The API offered for data access and manipulations is extremely simple and offers the following mechanisms: *get/put/delete* by key. The value is an

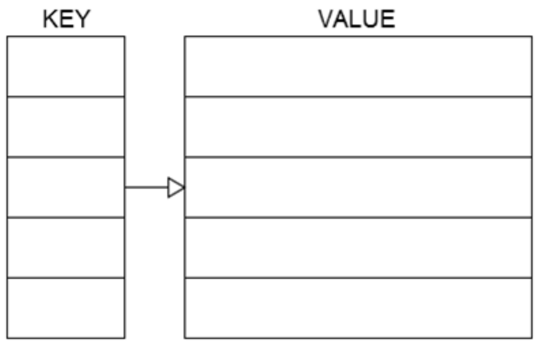


Figure 2.3: Key-Value Data Model schematic

opaque blob [2], it means that data cannot be filtered by value.

In newer implementations, in addition to the basic find-by-key method, there is also the possibility to find multiple values by a key-range. To achieve this the partitioning cannot be random but it is necessary to split data maintaining an order. To fetch values mapped by a key-range with random partitioning, the query need to be executed on all nodes. Otherwise with an Order Preserving Partitioner [3] a key-range can be mapped on the correct subset of nodes, reducing latency and traffic on the system.

The choice to provide only these simple functions is to keep high performance and scalability profile.

This type of database is completely no-schema, only in some case key is organized in namespace.

Amazon DynamoDB

Amazon offer on his Amazon Web Service (AWS) infrastructure different storage possibilities: Amazon S3, Amazon SimpleDB and Amazon DynamoDB.

Like Amazon says, the difference between DynamoDB and S3 is: *"Amazon DynamoDB stores structured data, indexed by primary key, and allows low latency read and write access to items ranging from 1 byte up to 64KB. Amazon S3 stores unstructured blobs and suited for storing large objects up to 5 TB . . . large objects or infrequently accessed data sets should be stored*

[2]In the less purist implementations like Redis the value is not opaque, this subcategory is also called Data Structures Server, because data structures are known by the system

[3]Order Preserving Partitioner is the name used in Cassandra, that is classifiable as Column-oriented NoSQL Database, but the idea is valid for all solutions that apply the data sharding

in Amazon S3, while smaller data elements or file pointers ... are best saved in Amazon DynamoDB." [8]

And also between DynamoDB and SimpleDB: *"Both services are non-relational databases that remove the work of database administration. Amazon DynamoDB focuses on providing seamless scalability and fast, predictable performance. It runs on solid state disks (SSDs) for low-latency response times, and there are no limits on the request capacity or storage size for a given table. This is because Amazon DynamoDB automatically partitions data ... In contrast, a table in Amazon SimpleDB has a strict storage limitation of 10 GB and is limited in the request capacity it can achieve ... it may be a good fit for smaller workloads that require query flexibility. Amazon SimpleDB automatically indexes all item attributes and thus supports query flexibility at the cost of performance and scale."* [8]

Amazon DynamoDB is the Amazon Enterprise Big Data Storage Solution. It is developed on idea presented in the paper "Dynamo: amazon's highly available key-value store" [28], that became also an inspiration for several other products. Main objectives are high availability for write operations [4], allows of concurrent modifications of data, handle of big data with auto-sharding and automatic conflict resolution to remove the work of database administration. Only query on key and key-range, that are optimized with indexing, are allowed.

Most important techniques used in DynamoDB is summarized in table 2.4. A description of these techniques can be found in [28].

Amazon DynamoDB summary

- **Data Model**: Key-Value with a hash key column and optionally a range key column for sorting data.

- **CAP**: AP (cf. Section 2.3.2). BASE (cf. Section 2.3.2), eventual consistency grade is configurable with quorum mechanism [5] from eventual to strict, multi replica of data on more machines, possibility of live add node to increase system performance, always write on partitions with consistent hashing (hash-ring).

[4]High availability for write operations: this is related to the partition failure aspect of CAP. Write to many nodes at once is allowed, so users should always be able to write somewhere. And they will never see a write failure.

[5]Quorum mechanism: wait a configured number of server updates before return a positive result of update operation to client

Problem	**Technique**	**Advantages**
Data Partitioning	Consistent Hashing	Incremental Scalability
High Availability for writes	Vector clocks with reconciliation during reads	Version size is decoupled from update rates.
Handling temporary failures	Sloppy Quorum and hinted hand-off	Provides high availability and durability guarantee when some of the replicas are not available.
Recovering from permanent failures	Anti-entropy using Merkle trees	Synchronizes divergent replicas in the background.
Membership and failure detection	Gossip-based membership protocol and failure detection.	Preserves symmetry and avoids having a centralized registry for storing members

Table 2.4: Amazon's Dynamo - Summary of Techniques (from [28])

- **Auto-partitioning**: Data are automatically sharded on commodity machines with key hashing and are auto re-sharded on live add node. Conflicts are resolved automatically with a "last wins" policy

- **MapReduce**: it is possible to use Hadoop MapReduce with Amazon Elastic MapReduce that has a DynamoDB out-of-the-box integration [7]

- **RDBMS**: No transaction or secondary index support, only hash key and range key are indexed

- **Architectural**: DynamoDB is completely peer to peer architecture without a single point of failure, technically is a Fully Distributed and Shared Nothing Architecture. All nodes accept queries and a smart client can call the right node directly, but "stupid" client can call a load balancer to map his query.

- **Other**: strict internal service level agreements (SLAs) regarding "performance, reliability and efficiency" have to be met in the 99.9th percentile of the distribution [6], use of vector clocks

Project Voldemort

Project Voldemort is a key-value store initially developed for and still used at LinkedIn. [34] [35]

Like DynamoDB it is designed to provide high availability for write operations, concurrent modifications of data, auto-sharding and automatic conflict resolution. Only optimized queries are allowed and vector clocks is used to find conflict. But if a conflict is not resolvable by datastore, it is delegated to client application and the strongly consistent but inefficient two-phase commit (2PC) is used for reconciliation.

Project Voldemort summary

- **Data Model**: Key-Value, keys are organized in namespaces and both keys and values can be complex, compound objects, as well consisting of lists and maps
- **CAP**: AP. BASE, multi replica, always writeable, consistent hashing
- **Auto-partitioning**: auto-sharding without aouto rebalancing. Automatic (last wins) or custom client logic for conflict resolution
- **MapReduce**: no MapReduce or server code execution
- **RDBMS**: Secondary indexes depends on persistence chosen, no transactions, triggers or views
- **Architectural**: Peer to peer architecture without single point of failure
- **Other**: Persistence layer is pluggable with BerkleyDB, MySQL or even a custom implementation, use of vector clocks

In-Memory stores: Memcached, Redis, Scalaris

In parallel to persistent key-value database there are some in-memory store solutions. Main actors are Memcached and Redis that provide in-memory high-performance, distributed memory object caching system. Scalaris is

[6]To meet requirements in the average or median case plus some variance is good enough to satisfy more than the just majority of users according to experience made at Amazon.

different, it is slower then others but has a complete distributed transactional support.

For the first subclass, the entire dataset is loaded on main memory and all operations is performed in RAM. This is the pro and con of these solutions, they are extremely fast, for example Redis support 100.000 access per second on single machine, but the dataset size is limited by memory (if the are too much data, they can't be loaded or else they are substituted with LRU policy), at least to maintain the maximum performance without use of virtual memory [7]. A durability persistence layer on disk is optionally present but these products are more read-only oriented.

Memcached [29] is the precursor of genre, it is solid and widely used. It is great in a read-only scenario with all data persisted on a physical database. It acts like a big distributed cache. Many other NoSQL databases use a data caching mechanism to decrease latency on cached requests and have also a "Memcached compatible" feature, which means they offer a fully compatible Memcached API. For example Couchbase, that offers a specific build of Apache CouchDB 2.4.2, recommends to substitute all Memcached nodes (one per machine) with a Couchbase Server Cluster to add new features and resolve some problems like cold cache on node failure and lack of scalability [22].

Redis [41] is newer than Memcached and it is more featured: it provides powerful data types and powerful commands to leverage them like Hashes, Sorted Sets, Lists, and more. It's not a pure key-value store because the values are not opaque and can be manipulated. Out-of-the box persistence solution. Transactions with optimistic locking. It is comparable to an in-memory database not only a cache. Redis seems to be also a little faster than Memcached and performs well even with write operations. Redis is single thread process, so an instance for each core is to be used on multi core machine, like if they were different machines.

Scalaris [42] is a scalable, transactional, distributed key-value store. It uses a structured overlay with a non-blocking Paxos commit protocol for transaction processing with strong consistency over replicas. It does replication synchronously so data are guaranteed to be consistent. Scalaris also

[7]Virtual memory: it is a virtualization technique widely used for extending the size of main memory visible to client application, part of data are stored on disk and loaded in memory when is needed

supports transactions with ACID properties on multiple objects. Data are stored in memory, but replication and recovery from node failures provides durability of the updates.

In-Memory stores summary

- **Data Model**: Volatile Key-Value with optional durability layer.
- **CAP**: CP (with persistence in a cluster solution). In memory context, consistency is not a problem, but in distributed scenario strict consistency is not provided in Memcached and Redis. Also availability and partition tolerance are a nonsense because any instance is has all of data or a part of those but they are loaded from a persistent store, therefore these properties depend on persistence layer. Scalaris offers full ACID properties
- **auto-partitioning**: manual sharding on client side toward multiple instance
- **MapReduce**: no support to MapReduce or server code execution, but can be used on top of a MapReduce enabled architecture, like Hadoop, in a complementary manner.
- **RDBMS**: Redis offer view and transactions on local instance, Scalaris offer distributed transactions
- **Architectural**: Multi instance with read-only access is the standard distributed scenario
- **Other**: The operations on data can be complex because the data are in memory

2.4.2 Document-Based Databases

Document-based are considered by many as the next logical step from simple Key-Value stores.

The “Key” doesn’t change so much, in most case is REST friendly because the primary interface on most Document-based database is REST over HTTP.

Is the “Value” part that is really enhanced from base model: it is no longer opaque to the system. A Document is a more complex and meaningful data structure, with more attributes like an XML document, each of these can be filtered, but to enable this, the database system must adopt some techniques

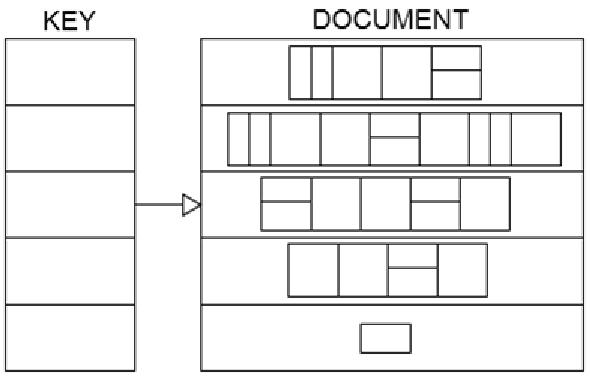


Figure 2.4: Document-Based Data Model schematic

dear to the RDBMS, secondary indexing on top, which lower the overall performance. Documents are organized in one or more collections.

This type of databases are schema-free, because documents in the same collection can be heterogeneous, with different inner structure. Documents can be nested in some implementations.

Usually documents are represented in JSON or XML forms or in a variant of them, like BSON for MongoDB. An example of document (in JSON):

```
{
  id: 1,
  title: 'My first blog post',
  body: 'This is my first blog post !!!',
  date: '2012-02-11 22.14.05',
  comments: [
    {by: 'lety', body: 'Good', date: '2012-02-11 22.15.04'},
    {by: 'song', body: 'Yeah', date: '2012-02-11 22.15.54'},
    {
      by: 'jess',
      body: 'Congratulations, we love your new blog',
      date: '2012-02-11 22.24.43'
    }
  ],
  tags: [
    {id: 1, name: 'first'},
    {id: 2, name: 'info'}
  ]
}
```

The tendency is to bring the NoSQL world near to the complexity of the RDBMS, without too losing in scalability and performance. Query model and API are always very rich and a set of high level features is provided out-of-the-box, like indexes, views, triggers, transactions, but each implementation provides a different subset of these.

Reference model: CouchDB (2.4.2) and MongoDB (2.4.2 are the most representative implementations.

CouchDB

Document-based NoSQL databases, of which CouchDB is part, aim to bridge the features gap with the RDBMS while maintaining a schema-free and scalability.

CouchDB is based on a powerful mechanism of view materialization, that use a MapReduce script written in Javascript language for view creation. They can be created dynamically at runtime and they are updated (materialized) on first reading and marked to be updated when concerned data are changed, so the writes are not slowed down and the reads are always consistent (but first reads are slowed down).

CouchDB is used for the Ubuntu One synchronization service, that is focused on syncing of two systems that has been disconnected for a period of time. They call this master-master replica because there is not a clear hierarchy, updates can be done on both systems. The concurrency control is a MVCC [8] with automatic conflict resolver (when it is possible, otherwise a manual reconciliation is needed).

CouchDB is also a web server. It can host HTML5 Applications called CouchApps, that is treated like simple data, so they are replicated and distributed over CouchDB synchronized installations. This is useful to create a load balancer for a high-throughput website, that goes well with a with a big data store.

Two companies carry out projects based on CouchDB: Cloudant develop BigCouch [19], which adds automatic scale-up strategy [9] inspired by Amazon DynamoDB paper [28]. The second is Couchbase [21] that adds either scale-up and scale-down on mobile strategies. Couchbase adds also a built-in data cache, a Memcached compatible API (Chouchbase previously was Mem-

[8]MVCC: Multiversion concurrency control, is a concurrency control method to provide concurrent access to the database using more version of any value

[9]Scale-up: automatically distributions of data across commodity servers or virtual machines.

base, a persistent implementation of Memcached protocols) [22], geo spatial indexing and others to CouchDB.

CouchDB summary

- **Data Model**: Document-based (without hierarchy of documents)
- **CAP**: AP. MVCC [10] (no garbage collection), strict consistency on master, eventual on slave, master-master replica and sync with offline database
- **auto-partitioning**: not out-of-the-box, but third party libraries exist, Lounge and BigCouch
- **MapReduce**: MapReduce is used for view materialization but is not exposed like a pure server side execution service
- **RDBMS**: lock free transactions are provided with the use of append only log and serialization of writes. Btree indexes on keys. For secondary indexes, the creation of views is needed.
- **Architectural**: CouchDB is optimized for read-heavy scenario, where consistence is needed on read
- **Other**: Only keys and key ranges can be queried but with the views, complex queries can be prepared, too

MongoDB and RavenDB

MongoDB [4] and RavenDB [40] are another face of Document-based NoSQL databases with respect of CouchDB.

One big difference is that CouchDB is MVCC based, instead MongoDB (and RavenDB) is more of a traditional update-in-place store. MVCC is very good for certain classes of problems: problems which need intense versioning; problems with offline databases that resync later; problems with a large amount of master-master replication. Along with MVCC comes some work too: first, the database must be compacted periodically, if there are many updates. Second, when unrecoverable conflicts occur, they must be handled by the client manually.

[10]MVCC: Multiversion concurrency control, is a concurrency control method commonly used to provide concurrent access to the database to implement transactional memory

MongoDB (and RavenDB) updates an object in-place, when possible, without fetch the entire document and without using multiversioning. An high update rates requirement is common in several scenarios and this mechanism helps to increase the overall performance.

Mongo's replication works great but, without the MVCC model, it is more oriented towards master-slave and auto failover configurations than to complex master-master setups.

Another difference: CouchDB only optimized queries (on keys and key ranges) are allowed, to support complex query, view materialization is needed. Here there is more hype on flexibility of query model to allow ad-hoc queries and sorts.

Like in RDBMS there is a query planner that, regarding history, creates and uses automatic indexes on documents and document attributes in a collection, so also ad-hoc queries and sorts can be fast after the first running.

Obviously the creation and maintenance of these indexes adds a certain amount of overhead for insert update and delete operations. Thus, indexes are best for collections where the number of reads is much greater than the number of writes. An index can be updated synchronously with the corresponding write operation to ensure the read consistency, but the write operation is so slowed down. Otherwise, when eventual consistency is allowed, the index can be updated in background. In that case, for a time interval, reading the data could return the old value, but when this interval will be elapsed the consistency will be ensured.

MongoDB demand to the client the choose of how much update operations (on replica set [11]) wait before confirmation, but this has nothing to do with the index update. Automatic indexing is always synchronously. A manual index can be declared as background index, but this will be updated asynchronously only on the primary set and not on the replica set.

RavenDB performs data indexing in a background thread, which is executed whenever new data comes in or existing data are updated. Running this as a background thread allows the server to respond quickly even when large amounts of data have changed, however in that case a stale indexes is used to execute a query. An index is marked with stale state when it needs to be updated and returns at normal state when is really updated. RavenDB idea is that "stale is better than offline". When a query is based on a stale index is also marked as stale, but can be also configured to wait the updated

[11]Primary and replica sets: primary set is the owner and responsible of a data set. Replica sets are a form of asynchronous master/slave replication, adding automatic failover and automatic recovery of member nodes.

data by client application.

The conclusion is that MongoDB is better for read-heavy applications, RavenDB for write-heavy applications, regards performance.

Both support auto-scaling out-of-the-box and also auto-replica but with different approaches. MongoDB offer two alternative way, simple master-slave replica and the newer replica sets [6]. RavenDB chooses peer-to-peer approach, so all machines accepts client calls.

MongoDB offers MapReduce API to execute distributed jobs, RavenDB use it only internally to create index and map queries on machines.

RavenDB also offer triggers, transactions and full-text search. And it can be extended with custom Microsoft .NET DLLs called "Bundles", simply by dropping them in a directory, for example there is an example of cascade delete bundle directly on the website of project. In MongoDB, transactions can be emulated with atomic operations and conditional update and suggested patterns like in [3] and [5]

API is similar, with object mapping and operations deferring concept. A query can be built in more step and only at the end the query is executed on server side. For example, specify at the end of the query how much items need to be skipped and taken.

MongoDB and RavenDB summary

- **Data Model**: Hierarchical Document-based
- **CAP**: MongoDB is CP and RavenDB is AP. BASE, eventual consistency in MongoDB and stale indexes in RavenDB, replica master-master or master-slave, optimistic locking and concurrency
- **auto-partitioning**: automatic sharding on more machine. RavenDB can use a custom sharding strategy
- **MapReduce**: MongoDB offers MapReduce as client side feature, RavenDB uses it for views and indexes, but not exposes the feature to the client
- **RDBMS**: Rich set of RDBMS features like secondary fixed and dynamic indexes, trigger and more. MongoDB doesn't provide transactions, but only a pattern to simulate two phase commit them with atomic operations and conditional updates. RavenDB offer local and distributed transactions
- **Architectural**: MongoDB is optimized for write-heavy scenario, RavenDB for read-heavy scenario

- **Other**: API similar to some of the ORM [12] APIs, for example like LINQ to Entities, with object mapping and operations deferring, after the creation of predicates tree. RavenDB supports Full-Text Search of Lucene.NET and it is extendible with creation of bundles

2.4.3 Column-Oriented Databases

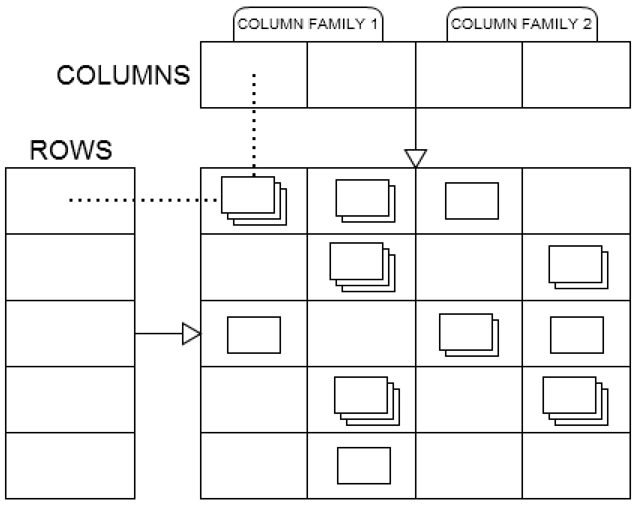


Figure 2.5: Column-Oriented Data Model schematic

Column-Oriented is not only a NoSQL database category, it is a way to treat data: the approach to store and process data by column instead of row has its origin in analytic and business intelligence where column-stores operating in a shared-nothing massively parallel processing architecture can be used to build high-performance applications. Notable products in this field are Sybase IQ and Vertica (cf. [38]). However the Column-Oriented NoSQL class is less purist and more flexible, it integrates both column and row orientation, and can be described like a "sparse, distributed, persistent multidimensional sorted maps" (cf. [32]).

These characteristics put the Column-Oriented category like the best suited to scale up and treat big data volume, because the partitioning of data can be applied vertically on columns as well as horizontally on rows.

Also performance on range queries is the best: sequential reads (scan on a column) are fast because data in a column is stored together and columns

[12]ORM: Object-Relational Mapping

compress better than rows because the data are (in most of case) similar.

Key-Value approach to fetch data is adequate for the some scenarios, but sometimes a more structured way to organize key is needed. The Column-Oriented data model variations are all inspired on the Google BigTable paper [17], that identify a value with a complex key, a triple: row/column/timestamp. In practice is a Key-Value with a multilevel key.

- **Column-Family:** contains columns of related data. It is a tuple (pair) that consists of a key-value pair, where the key is row-key and is mapped to a value that is a set of columns. In a RDBMS could be compared to a table but each row (value related to row-key) could have different columns. Column-families need to be created at design time and it is very important that they are properly designed, because they have a direct impact on vertically partitioning. Columns in the same column families are compacted together and are stored one near the others.

- **Column:** each column is a tuple (pair) consisting of a column name and a set of values, identified by a timestamp for multiversioning system. Columns can be added or removed dynamically.

- **[Supercolumn]:** in Cassandra (not in BigTable) column-families can contain columns or supercolumns (not at the same time). Supercolumn have a name and is a map of columns. Supercolumns can be added or removed dynamically, too.

- **Row:** is row-key and every value related to it (in multiple column-families).

The store is sparse, so if a column or a column-family is empty there is no need of space on disk.

Query model and API is usually simple like in Key-Value model: *get/put/scan*[13]*/delete*.

Get or filter can only be done on keys (row-name / column-name / timestamp). The “Value” is opaque like in the Key-Value model, but the schema is semi-structured in columns, then the read operations can use projection with granularity of the single column. A value can be identified bye this path: *keystore - column-family - column - timestamp*.

Data are automatically stored and sorted by row-name and column-name. This intend that the choice of keys is extremely important to get data quickly

[13]scan: the scan operation is an iteration on more rows/columns, with a start and stop key

and ready to use (see example below).

Reference model: Google BigTable (2.4.3), Cassandra (2.4.3).

Column-oriented schema design example Take a look on a Twitter like example, with tweets related to users. Users and tweets data are stored respectively in Users (Figure 2.4.3) and Tweets (Figure 2.4.3) Column Families.

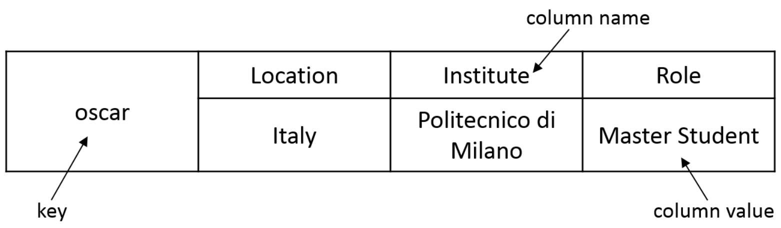
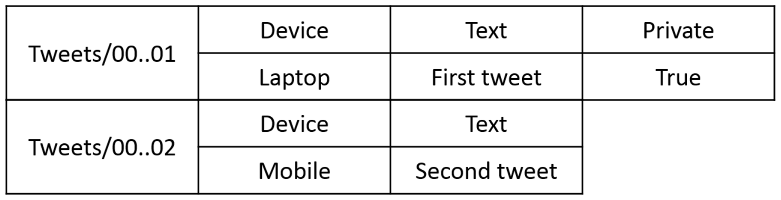


Figure 2.6: Users column family



Tweets/00..01	Device	Text	Private
	Laptop	First tweet	True
Tweets/00..02	Device	Text	
	Mobile	Second tweet	

Figure 2.7: Tweets column family

The typical scenario is the request of last tweets in general or by a specific user. An smart choice for the tweet key can be a sequential value, so tweets will be stored and sorted in chronological order and a get by key range, that is optimized, return the correct results. With a random generated key the tweets will be stored in not useful order.

Note that the columns contained in the two rows of Tweets are different, an example of heterogeneous data.

In these schema an information is missing: the relation between tweet and owner (user), no foreign-key on Tweets column-families. This can be saved for example in a new Column-family, UserTweets (Figure 2.4.3), mapped by the user name (row-key) and that contain a supercolumn Timeline with the keys of user tweets (column values).

With this schema to get the lasts '20' tweets by user 'oscar' will be necessary to execute two query *(a LINQ style notation is used for simplicity)*:

```
var tweetIds = cfdb.UserTweets.Get("oscar")
                   .Fetch("Timeline")
                   .Take(20)
                   .OrderByDescending()
                   .Select(x => x.Value);

var tweets = cfdb.Tweets.Get(tweetIds);
```

To get the lasts '20' tweets in general, only need one query on Tweets:

```
var tweets = cfdb.Tweets.Take(20).Select(x => x.Value);
```

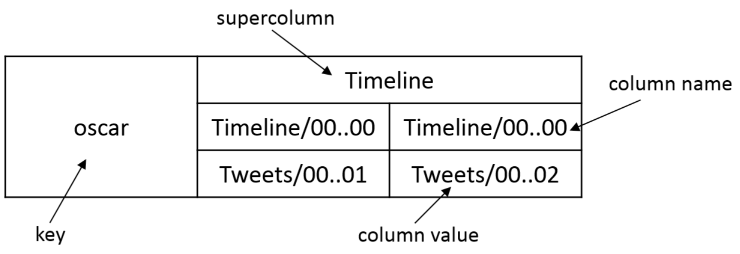


Figure 2.8: UserTweets column family, to store the relations

Horizontal and Vertical Partitioning The Column-oriented data model is the best suited for data partitioning. It supports both horizontal and vertical partitioning (cf. Section 2.3.2) because it is stored by columns instead of rows. A column is independent from each others ans it is organized by row keys. It can be placed on a different node and can be also splitted horizontally on the row key. In the Figure 2.9 there is a graphical example.

Google BigTable

In 2006 Google released a whitepaper [17] which was quite influential to the NoSQL ecosystem and it assumed the role of biggest player in the NoSQL Big-Data context. BigTable is described as "distributed storage system for managing structured data that is designed to scale to a very large size: petabytes of data across thousands of commodity servers". Google currently uses it in several proprietary services, like Gmail, Google Search, Google Reader, Google Docs and more. It is available for developers only on Google App Engine PaaS [14].

[14]PaaS: Platform as a Service

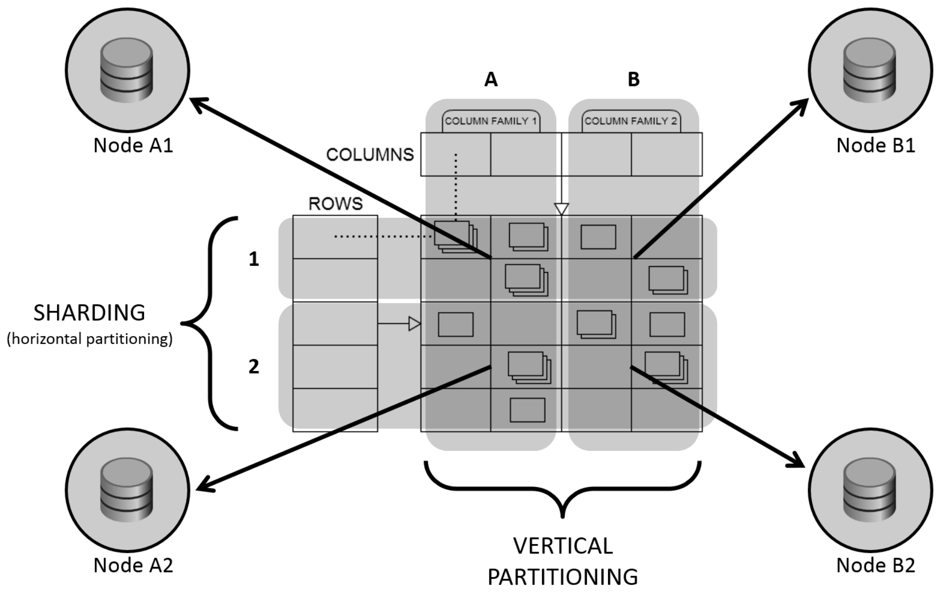


Figure 2.9: Column Based Horizontal and Vertical Partitioning

"A Bigtable is a sparse, distributed, persistent multidimensional sorted map." It is the Column-oriented store in the NoSQL manner, that scale on both rows and columns (cf. 2.4.3). It is built on top of Google File System (GFS) and stored in an immutable datastructure called SSTable. Access to shared resources in regulated by Chubby, a distributed lock system (that use Paxos-style consensus protocols [36, 43]). Values are multiversioned. The application can define how many entries of data, marked with a timestamp, should be keep [17].

Different from DynamoDB, BigTable ensures consistency and partition tolerance (CP in CAP Theorem, cf. 2.3.2) instead of grants the high-availability in partitioned network scenario. It has a master node that monitors and coordinates the activities of all region servers, so the architecture is not fully peer to peer. The system could go offline until a new master node is elected or network problems are resolved.

BigTables are auto replicated in five copies with one master. Nodes can be added live to increase performance ensuring automatic sharding on rows and scaling vertically. MapReduce is available to read from and write to a BigTable.

Special feature of BigTable, compared to other products, is data compression, that makes it possible to occupy less disk space and also improve performance because it work directly on data without the need to uncompress first.

Google BigTable summary

- **Data Model**: Column-oriented
- **CAP**: CP. BASE with multiversioned value and garbage collection, ensure Consistency and Partition tolerance but not grants "Always Availability", five active replica with one master, live add node.
- **auto-partitioning**: auto sharding (horizontal) and vertical partitioning with auto re-balancing
- **MapReduce**: is available to read from and write to a BigTable
- **RDBMS**: No secondary indexes, no transaction only atomic operations on single row
- **Architectural**: No P2P, there is a master node
- **Other**:

BigTable derivatives: HBase and Hypertable

Google BigTable can be used only on Google infratructure. HBase and Hypertable are two different projects with the same objectives: implement BigTable outside of Google.

To do this the solution adopted by them is to use Hadoop as underlying layer, that is comparable with the Google infrastructure. A short brief is shown in Section 2.3.2 and in Table 2.5 we present the comparison of Hadoop and Google stacks.

	Google	**Hadoop**
Distributed Filesystem	GFS	HDFS
Distributed Locking System	Chubby	Zookeeper
Algorithm Distribution Framework	MapReduce	Apache MapReduce

Table 2.5: Google and Hadoop stacks comparison

BigTable derivatives summary

- **Data Model**: Column-oriented (like BigTable)

- **CAP**: CP. BASE with multiversioned value and garbage collection, ensure Consistency and Partition tolerance but not grants “Always Availability”
- **auto-partitioning**: auto sharding (horizontal) and vertical partitioning with auto re-balancing
- **MapReduce**: With Apache Hadoop MapReduce
- **RDBMS**: No transactions, no secondary indexes, Hypertable as no triggers, HBase has coprocessor that is comparable to triggers
- **Architectural**: Very close to BigTable
- **Other**: Can be deployed everywhere there is an Hadoop installation

Cassandra

Cassandra [11] is firstly developed by Facebook and after become open source as an Apache project. At this time, it is in use at Facebook, Twitter, Netflix, Urban Airship, Constant Contact, Reddit, Cisco, OpenX, Digg, CloudKick, Ooyala, and more companies.

Based on a mix and match of Google and Amazon technologies, trying to obtain best solution for BigData storage. It is fully peer to peer and ensures high-availability, eventual consistency, elasticity (incremental scalability) and optimistic replication like DynamoDB. It adopts BigTable column-oriented data model introducing the supercolumn concept (cf. section 2.4.3).

An important feature of Cassandra is the possibility of configuring, case by case, the grade of consistency between “One”, “Any” and “All”. That is the number of consistent values that the system should wait before returning results. With “All” grade, strict consistency can be obtained.

DataStax distributes and supports a version of Apache Cassandra. There are two versions of DataStax distribution, a free Community Edition and an Enterprise Edition. The Enterprise Edition integrates Cassandra with other Apache projects such as Hadoop (to enable the use of MapReduce). Both versions include the OpsCenter, a visual management and monitoring tool for Cassandra that enables developers to manage and monitor their Cassandra clusters [24].

Cassandra summary

- **Data Model**: Column-oriented (Like BigTable) with supercolumns

- **CAP**: AP (but consistent values can be obtained configuring the request). BASE (Like DynamoDB)
- **auto-partitioning**: auto sharding (horizontal) and vertical partitioning with auto re-balancing
- **MapReduce**: MapReduce can be enabled by using Cassandra on Hadoop
- **RDBMS**: Secondary indexes
- **Architectural**: Like DynamoDB it is a full peer to peer architecture.
- **Other**: Writes are faster than reads

2.4.4 Graph-based Database

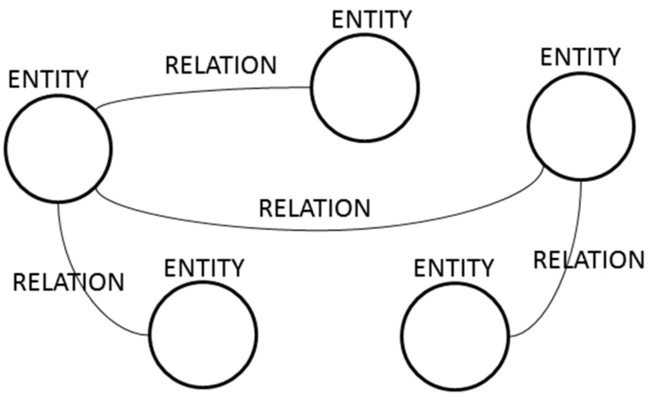


Figure 2.10: Graph-based Data Model schematic

It is important to cite Graph-based category but it will not be treated in this document because of the different evolutionary path taken with respect the others. Key-Value, Document-Based and Column-Oriented categories aim at the entities decoupling to facilitate the data partitioning and have less overhead on read and write operations, whereas Graph-based take the modeling the relations like principal objective. Therefore techniques to enhancing schema with a Graph-based database may not be the same as used with Key-Value and others.

The graph data model is best able to model domain problems that can be represented by graph as social relationship, maps etc. Particular query languages allow querying the data bases by using classical graph operators as neighbor, path, distance etc. Unlike the NoSQL systems we presented, these systems generally provide ACID transactions. Among the others we mention Neo4j [31], an open-source implementation of Graph-based NoSQL database. Nodes store data and edges represent relationships. The data model is called "property graph" to indicate that edges could have properties. Neo4j provides a REST interface or a Java API.

2.5 The Lack of Relation Concept

After exposing the benefit deriving from the usage of NoSQLs as an alternative to RDBMS and after giving a complete classification of existing products, we have to show also the main negative aspect.

The principal issue is the lack of the relation concept. This is also the principal reason that enables most of benefits described in previous section, but sometimes relations are necessary.

After we have described this in details we will expose all others lacks.

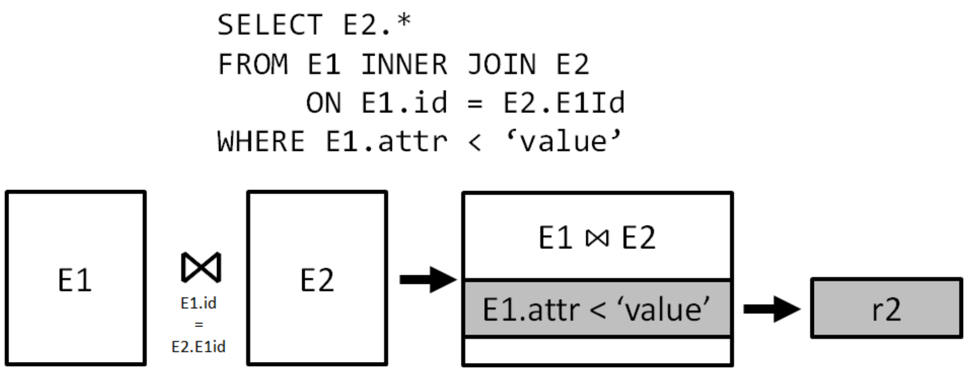
The main concept of NoSQL (with exception of Graph-oriented, cf. Section 2.4.4) is the independence of each database entities from everyone else. Entity data are serialized in collections and there are no links from an object to another. So, partitioning data and getting a scalable database should be extremely easy (cf. Section 2.3.2).

However, many real cases need to relate multiple objects together. In RDBMSs the relation concept is a centerpiece and it is achieved always in a standard way depending on the type of referential constraint, (1,1), (1,N) and (N,M). These constraints are directly enforced by database systems and cannot be violated.

The problem is that in NoSQL databases the relation concept does not exists at all. To enable relations in it is always necessary to find a way to serialize this information and to use it, because there is no general way to achieve it.

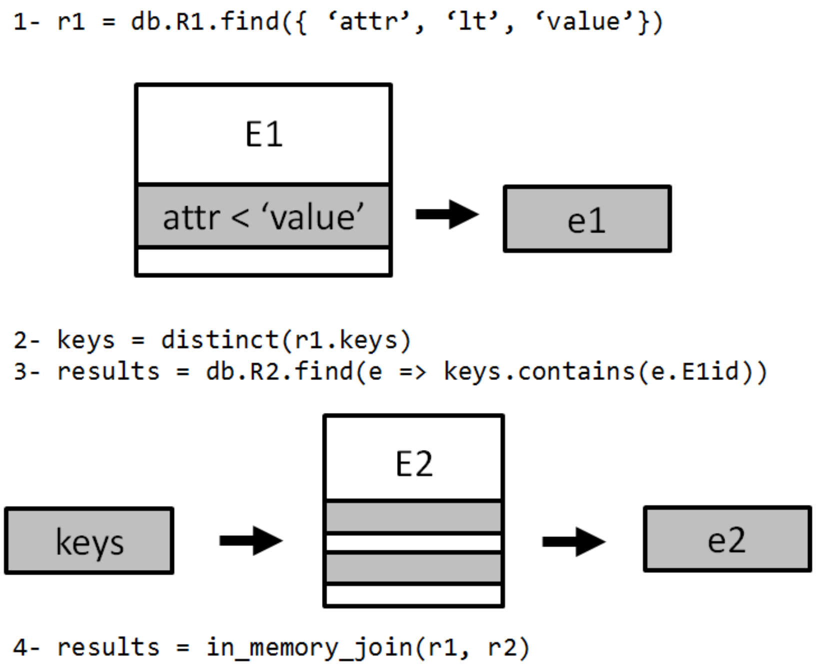
In the common way the solution is to fetch tuples from the first entity set $E1$, extract and distinct the join attributes (named $keys$) and fetch objects of secondary entity set $E2$ that satisfy the join predicate $keys.contains(E2.E1id)$ and then combine $E1$ and $E2$ to obtain final results. The execution of query is shown in Figures 2.11 and 2.12 respectively in SQL and NoSQL way.

What we are trying to do is to find a general way to handle any type of relations in NoSQL databases maintaining high scalability profile. In the next chapter we will describe our solution.



```
SELECT E2.*
FROM E1 INNER JOIN E2
     ON E1.id = E2.E1Id
WHERE E1.attr < ‘value’
```

Figure 2.11: Execution of a query on a relation in SQL way



```
1- r1 = db.R1.find({ ‘attr’, ‘lt’, ‘value’})
2- keys = distinct(r1.keys)
3- results = db.R2.find(e => keys.contains(e.E1id))
4- results = in_memory_join(r1, r2)
```

Figure 2.12: Execution of a query on a relation in NoSQL way

2.6 Summary

In this chapter we have proposed a NoSQL classification with respect the data models. For each class we have described the characteristics of main existing products. We have also presented the motivations to use or not NoSQL databases instead of classical RDBMS. finally we have introduced the problem of handle relation in NoSQL databases that is the starting point of the next chapter.

Chapter 3

Controlling Serialization of Data in NoSQL Databases

3.1 Definition of the Problem

NoSQL are mainly used to meet the requirements of high performance, when RDBMSs show their limits. As we have seen in Section 2.3.2 the main problem is the difficulty to scale efficiently when the amount of stored data grows. NoSQLs instead are designed to scale well.

Nevertheless, as highlighted in the previous chapter, the NoSQL world has no standard concept of relationship between different database entities (cf. Section 2.5) and, even more important, it has no compositional operators, like the join. All the application objects are just serialized in independent collections and all relationships between objects are to be managed by the application layer. Moreover, only one collection can be fetched at a time.

The problem is to automatically handle relations and other features that must be manually implemented using a NoSQL database in a distributed scenario.

In this chapter first we will present the commonly adopted solutions to handle relations using a NoSQL database. These techniques are not general and they must be implemented at the application layer.

Then we present our approach that aims at providing a general representation of the relations for NoSQL databases that is context independent and is especially scalable. Given this general representation, in the next chapter we will define two alternative logical models to achieve it.

3.2 Common NoSQL Relationship Serialization Techniques

The relationship between different objects in a NoSQL database are commonly built with a variant of two techniques: Embedded Objects and Client-Side Joins.

Embedded Objects

One of the most “NoSQL like” relationship implementation pattern is “Embedded Objects”. It consists in copying all the objects related to an entity directly in the entity body. For example if we have two entity sets *users* and *profiles* that are related one to one:

```
users: [{                    profiles: [{
  id: 1,                       name: 'Oscar',
  login: 'oscar',              surname: 'Locatelli',
  date: '...'                  user_id: 1
}, ... ]                     }, ... ]
```

Performing Embedded Object we obtain a unique collecion *users* that contains user objects with associated profile nested in the object body:

```
users: [{
  id: 1,
  login: 'oscar',
  date: '...',
  profile: { // this is the (1,1) relation
    name: 'Oscar',
    surname: 'Locatelli'
  }
}, ... ]
```

If the relation is (1,N) it is more commonly known as “Nested Collection”. An example of (1,N) relation is the one below, that describes a blog containing several comments:

```
posts: [{
  id: 1,
  title: 'My first blog post',
  body: '...',
  comments: [ // this is the (1,N) relation
    {by: 'lety', body: '...', date: '...'},
      {by: 'song', body: '...', date: '...'},
      {by: 'jess', body: '...', date: '...'},
      {by: 'paul', body: '...', date: '...'}
    ],
    tags: [ // this is the (N,M) relation
      {id: 1, name: 'first'},
      {id: 2, name: 'info'} // this is replicated
    ]
  }, ... ]
```

Embedding objects is preferable in the case of (1,1) or (1,N) relations because there is no replication of the secondary entities. In case of (N,M) relations the total number of entity replicas is to maintain low, because if a secondary object is updated, all its replicas need to be updated.

```
posts: [{
  id: 1,
  title: 'My first blog post',
  body: '...',
  comments: [ ... ],
  tags: [ // this is the (N,M) relation
    {id: 1, name: 'first'},
    {id: 2, name: 'info'} // this is replicated
  ]
},
{
  id: 2,
  title: 'My second blog post',
  body: '...',
  comments: [ ... ],
  tags: [ // this is the (N,M) relation
    {id: 3, name: 'second'},
    {id: 2, name: 'info'} // this is replicated
  ]
}, ... ]
```

Moreover this technique is good when the read of the entire relation takes place usually starting by the primary entity. For example fetching *tags* related to firstly filtered *posts* is very efficient [1].

[1]For the example we use a MongoDB-like sintax for db operations and a LINQ-like one for application logic

```
db.posts.fetch('tags').find({ 'id', 'eq', 2 })
```

Doing the opposite on the same collection is not so good. For example take a *tag* and then take the related *posts* needs a complex pattern matching, that the most of NoSQL database not embed. Commonly load all *posts* from database and then filter them in application is needed.

```
all = db.posts.fetch('tags').findAll()
results = all.filter(p => p.tags.any(t => t.id = 2))
```

Document-oriented database usually provide the querying on nested sub-objects. It is not so good if there is replication (the objects to scan is more than the real number), but it is possible and if that is not the principal query pattern on the relation it could be a valid way. Here an example with complex pattern matching directly on db.

```
all = db.posts.fetch('tags').find({ 'tags', 'match', 'id = 2'})
```

This query can be executed efficiently using another collection to store the tags that embed the inverse relation (relation of one tag with its posts) that can contain also not the entire post entities but for example only a list of the ids.

```
tags: [{
  id: 2,
  name: 'info',
  posts: [ 1, 2 ] // ids of related posts
}, ... ]

keys = db.tags.find({ 'attr', 'eq', 2 })
results = db.posts.findByKeys(keys);
```

However these two collections require to be synchronized, so an overhead on the update operations is shown. This is the common read/write performance tradeoff introduced by denormalization.

The primary objects could become very big if they have an high number of secondary ones related. This can create many problems. First, a maximum size for the single object storable in db may exist. Also, if the database does not allow partial reading of the object, the performance can drop dramatically in both reading and writing. And finally, the scalability problem: for a huge amount of secondary objects, the object could need to be splitted and stored in several nodes, but this could be not possible.

All these problems are present in most of simple Key-Value stores. Document-oriented, usually, and Column-based stores allow to project only necessary attibutes/columns and also to update only what is needed. Column-based stores also cancel the scalability issue because a row (object) can have very high number of columns (attributes) and it can be partitioned also vertically on different nodes if is needed.

Commonly this technique is used with small cardinality relations, in order of tens, to work on all databases.

Client-side Joins

In most cases the management of relationships between objects are deferred to the client application layer. Like in RDBMS the key of primary objects is stored as dependent-object attribute, or for (N,M) relations an external collection with the pairs of keys is used. Once the first object has been loaded (or the external pair of keys), it must be parsed to extract sub-objects keys and then the secondary objects can be read.

This mechanism is general purpose and does not hide any issue, but all the logic must be implemented, every time, on the client application. Two-passes (or three-passes) queries for fetching relation is not the only thing that needs to be implemented. In the insert operations the client must check that the (1,1) or (1,N) structural relation constraints is not violated and when an object is deleted also all the connected (N, M) relationship pairs need to be removed.

In conclusion we can say that this technique is the simplest one but needs a lot of custom work in every implementation.

In order to help developers to simply handle relations, some high level techniques are embedded into database APIs. For example MongoDB has a convention for representing a reference to a document, named *DbRefs*. It is simply a sub-document embedded in the primary with the collection, the key and, optionally, the database of the secondary object to connect. This is used to implement an high level operator in drivers used to fetch automatically the sub-object. But there is no difference with respect to saving the object id and fetch it manually as explained in official documentation [2].

```
{
  _id: ObjectId("5126bbf64aed4daf9e2ab771"),
  // ... other attributes
  creator:
  { // this is the DbRef
    $ref: "creators",
    $id: ObjectId("5126bc054aed4daf9e2ab772"),
    $db: "users"
    }
  }
```

3.3 Marijà: A Framework for Data Distribution

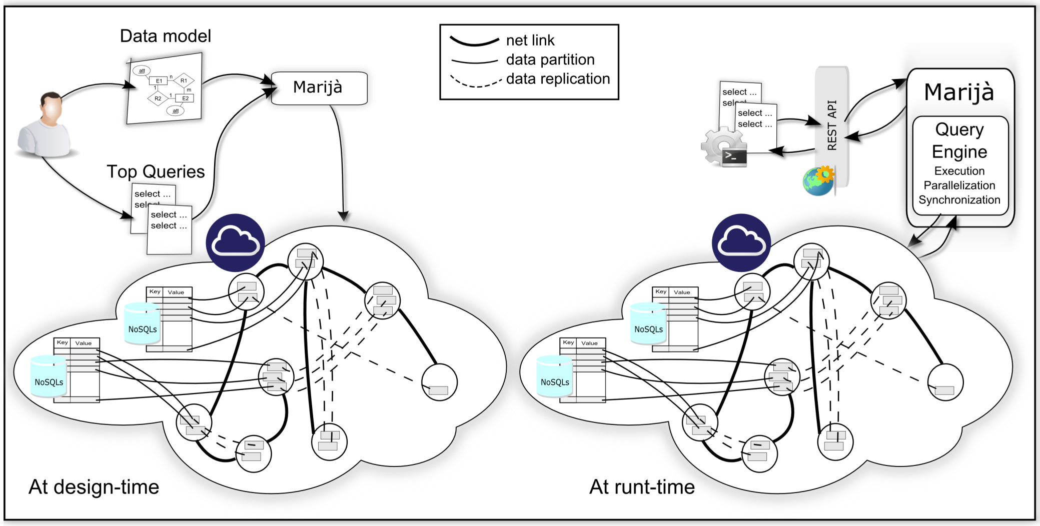


Figure 3.1: Marijà Architecture

As mentioned in the Section 3.1 much of logic necessary for data manipulation in NoSQL databases is left to the programmer, who needs to implement all in application software. An RDBMS for example is responsible to maintain the referential integrity of the data, manage concurrent access, perform efficient and complex queries. Furthermore, in NoSQLs does not exist a common manipulation language, as SQL is in RDBMS. The reason is that the underlying logical models are different for all the products. In Chapter 2 we presented a classification of NoSQL data-models that should clarify the point.

The Marijà framework aims at addressing these issues in a distributed scenario. It is composed of two main layers, one supporting the design time activities and the other the runtime operation of NoSQL databases (see Figure 3.1). The design time layer supports developers in creating the logical schema of the database starting from a data model, typically provided in terms of an entity-relationship diagram, and from the definition of the most used queries for the system under development.

Based on these two types of input, Marijá determines the tables that are most suited to optimize the execution of the queries and these tables will be distributed on several nodes. It might happen that different queries require different, partially overlapping, tables to be defined. In this case, all possible tables are created and populated, and the runtime layer of Marijá is in charge of: *i)* ensuring that data updates are consistently propagated to all interested tables and *ii)* managing the execution of queries on the most suitable tables.

Marijà addresses the problem by using MapReduce algorithms to manage data. As Marijà states, a deeply parallelization of tasks is possible under the assumption of "single node independence".

At high level, we can describe the "single node independence" concept as the property that an operation applied to the entire system is equal to the aggregation of the same operation applied to several nodes. Formally, assumed an operation $OP(X)$, it is equivalent to $OP(X_1) \oplus OP(X_2) \oplus ... \oplus OP(X_n)$, where X_i is a node of a system and X is the entire system.

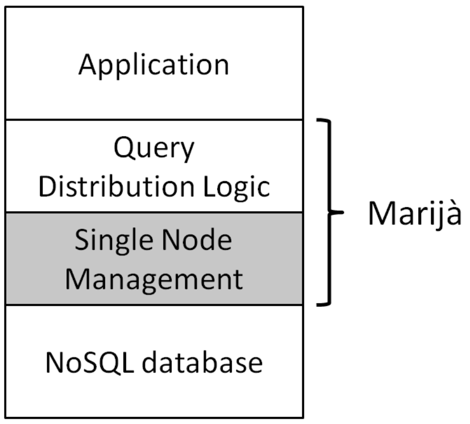


Figure 3.2: Marijà Layers

In Figure 3.2 we show the logical layers of a Marijà system. At the top level there is the Application layer, at the bottom layer there is a concrete NoSQL database. In the middle of them we find Marijà that is divided in "Query Distribution Logic" and "Single Node Management". Marijà acts as database common interface from the application side and as database management system from NoSQL side. As highlighted in the figure, the objective of this thesis is to support the "Single Node Management" layer.

Marijà Query System Overview

Here we report a brief on Marijà Query System.

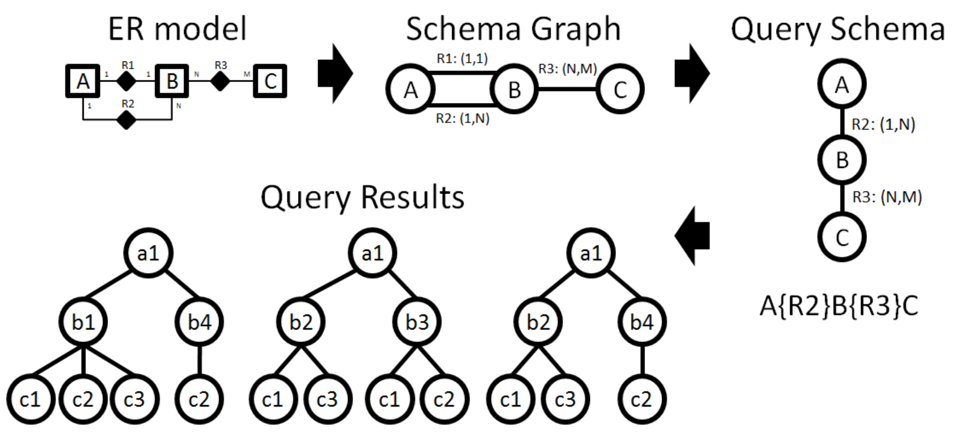
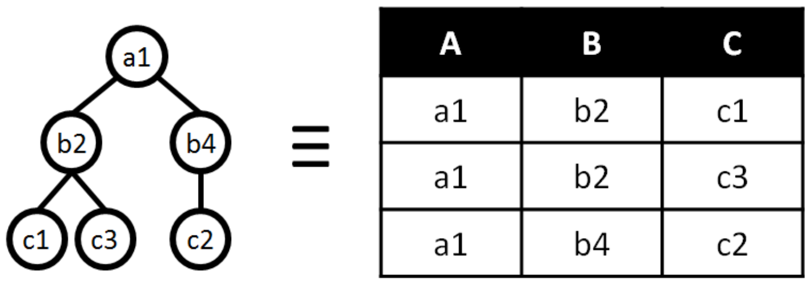


Figure 3.3: Marijà: Query Schema

Marijà design starts with the ER model of data. Marijà needs that all the relations are in binary form. An equivalent representation of a general ER schema that use only binary relations always exists.

Once that ER model is in "binary form" this could be directly traduced in a non-directed graph with entities as nodes and relations as edges. Doing a query on this graph is equal to take a portion of it, a sub-graph.



A	B	C
a1	b2	c1
a1	b2	c3
a1	b4	c2

Figure 3.4: Marijà: Mapping a tree on a table

In details a Marijà query is described by a point of view, the entity from which to start, and a path, the list of relations to be traversed to fetch the other entities. To build the query sub-graph no edges can be visited more than one time. An example of query path is:

$$A\{R2\}B\{R3\}C$$

where A,B and C are entities and R2,R3 are relations.

Now to select the query sub-graph we start from the node that represents the point of view, take the relations (edges) in the path and the nodes between them. This sub-graph is named "Query Schema" and contains all the entities and relations that must be queried.

The materialization of results will follow this schema but from this will produce a tree starting by the point of view entity with all others entities linked.

This flow is shown in Figure 3.3.

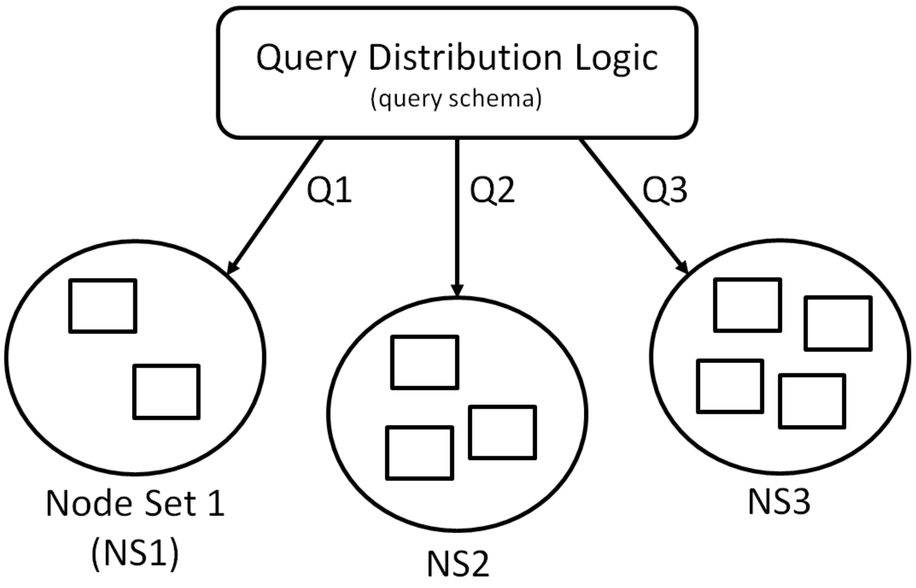


Figure 3.5: Marijà: Query Distribution

Query Execution Marijà works in a distributed scenario and it is deployed on a network of nodes. These nodes are divided in sets and each node set is responsible of the execution of one or more designed queries (see Figure 3.5). All nodes in a node sets can answer to the same query set.

Node sets are chosen with an analysis of a set of pre-engineered queries. Relations that will be present more frequently in same queries will be allocated to the same node sets.

The query execution will be provided by a smart Map/Reduce mechanism (cf. 2.3.2). The "map" function selects what node sets are needed to answer query, if it is one of the top queries, it is sent directly on the associated node set, else it will be decomposed in pieces and sent to all node sets needed. Then the "reduce" function collect and combine results and return an answer to the client (see Figure 3.6).

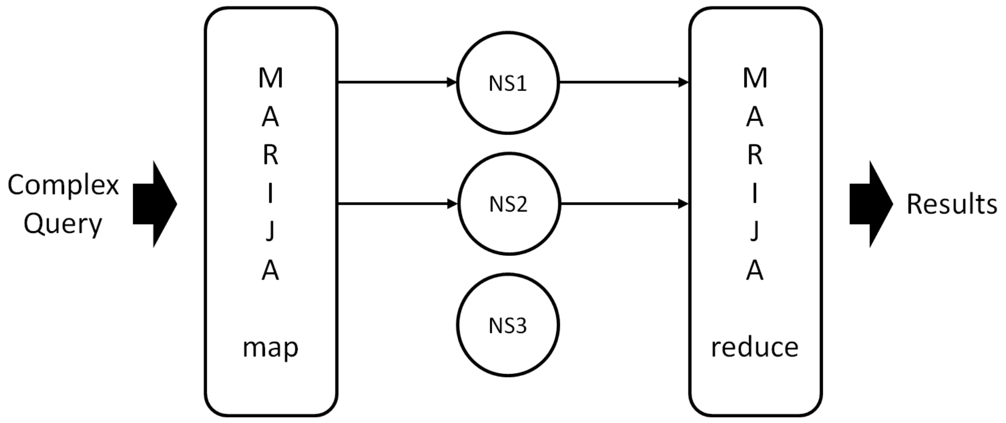


Figure 3.6: Marijà: Map-Reduce Execution

Obviously in a BigData scenario, a Node Set could be responsible of large amount of data, so data are partitioned on more nodes (see Figure 3.7).

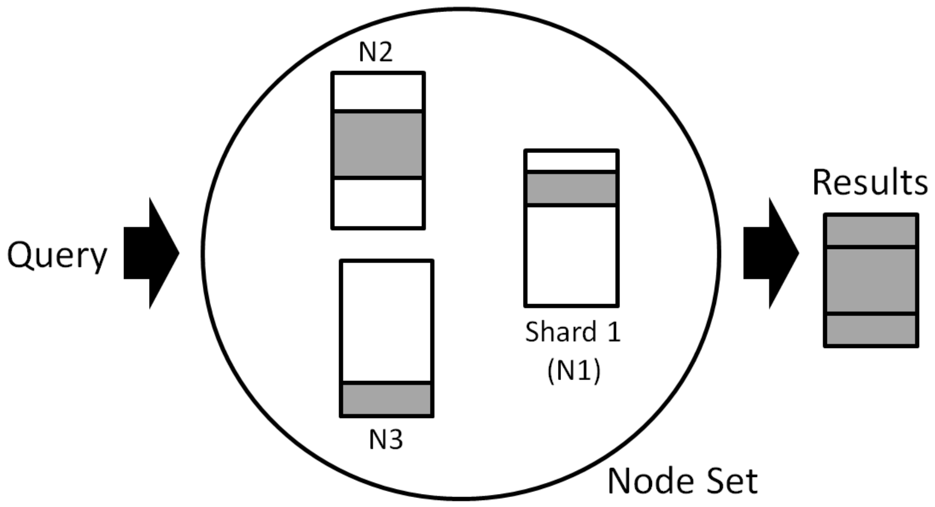


Figure 3.7: Marijà: Collaboration of Nodes

The sharding is not based on random order, it is done on relation keys ordered to balance the number of entities involved. A single node must hold locally all entities that linked by local relations to avoid distributed join to build local result.

All these choices have been made to ensure the independence of the individual node in working on relations of which it is responsible (single node independence assumption). Starting from this hypothesis, we need to choose a suitable logical model to store relations on single node.

In the next section we will continue from here.

3.4 Summary

In this chapter we presented an overview of the Marijà Project. Marijà is a framework to handle relationships in a distributed scenario, preserving scalability. The project assumption of "single node independence" is the starting point of the this thesis work.

In the next chapter we will propose two alternative strategies to implement the single-node model.

Chapter 4

Single Node Logical Models: MinR and MaxR

4.1 Single Node Logical Models: MinR and MaxR

In this chapter we show the theoretical contributes of this thesis.

As assumed in previous chapter, a single Marijà node is independent from each other. Focusing on single node, we propose two opposite strategy to model relationships. The first, “Minimal Replication” (MinR from here), which involves only the primary copy of the data of each entity and the second, “Maximal Replication” (MaxR), which includes a replica of the entity for each tuple of the relation to which it belongs. The two models will be described after in this section and also, by analyzing common details between the two models, a common schema of relation will be presented.

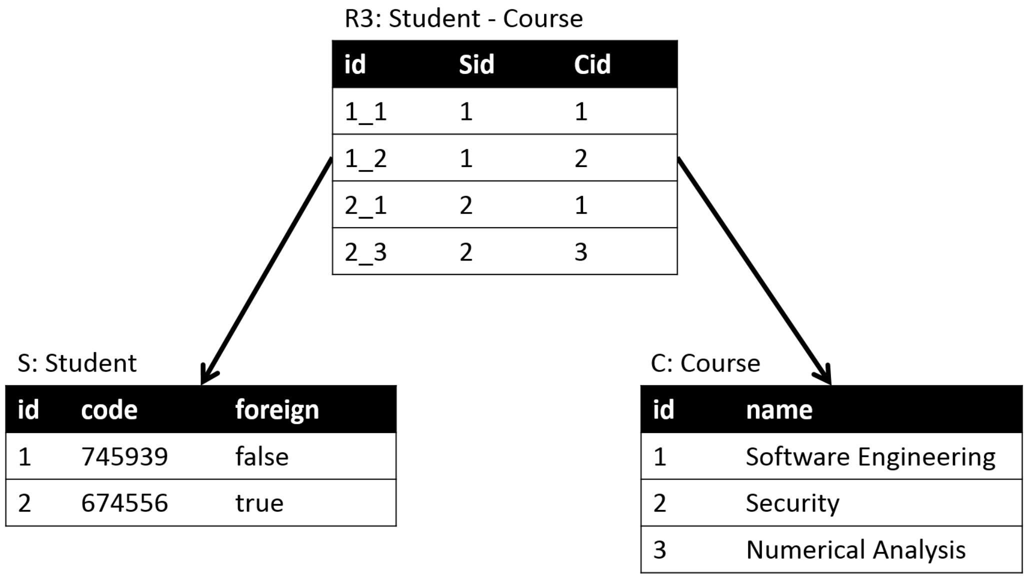
Then we will propose a method for comparing models. The method consist of the mapping of CRUD (Create, Read, Update and Delete) operators on MinR and MaxR models and analyze costs of algorithms.

From this comparison we will conclude what are the operators that need more investigation. In the next chapter we will define and execute some tests on concrete NoSQL implementation.

4.1.1 Minimal Replication (MinR)

As we shown in Figure 4.1 this representation is very similar to many-to-many standard RDBMS relation. The differences are that the MinR is used to model all kind of relationships.

The schema of a relation consist of:



R3: Student - Course

id	Sid	Cid
1_1	1	1
1_2	1	2
2_1	2	1
2_3	2	3

S: Student

id	code	foreign
1	745939	false
2	674556	true

C: Course

id	name
1	Software Engineering
2	Security
3	Numerical Analysis

Figure 4.1: MinR: Implementation Example

- one table for each entity, with id and attributes.
- one table to store the relation tuples, each of them contains an identifier, the keys of the related entities and the optional attributes of the relation (Figure 4.1).

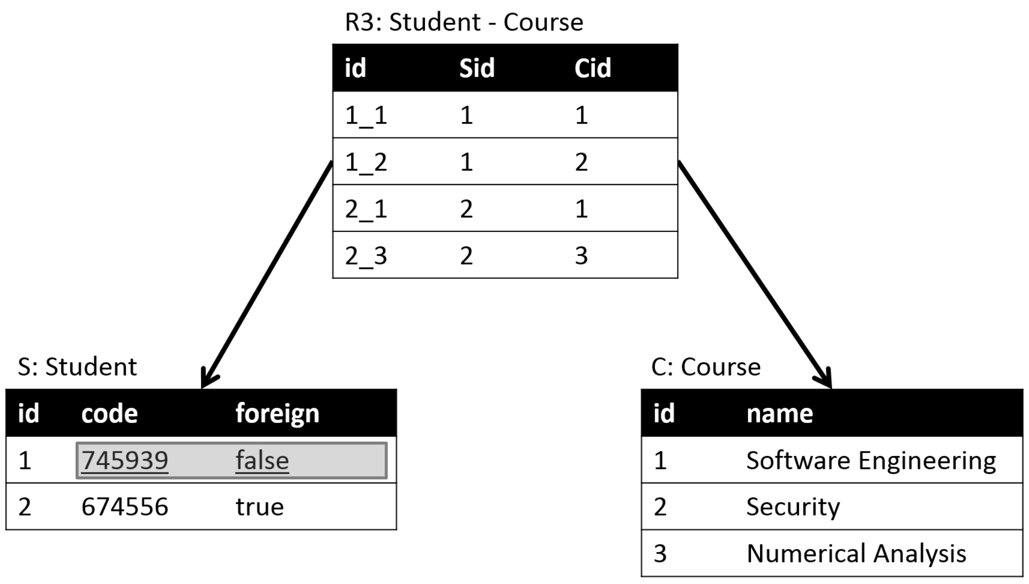
The write speed and consistency are the highlights of this model. The consistency is certainly strict (not eventual) since the model has no replicas.

Updating an entity in MinR means saving changes to the tuple in the entity table. In the Figure 4.2 we show an example of entity update. We have to update data of the Student entity with $id = 1$. The only thing to do is to update the tuple in the S entity table with $id = 1$.

Reading a relation instead consist of in more than one fetch operations:

1. loading the tuple in the relation table
2. loading related tuples from the respective entity tables, using entity keys stored in relation tuple

In the Figure 4.3 we show an example of relation read. We have to fetch the R3 relation with $id =' 1\_1'$. So, we have to read the tuple with $id =' 1'_1$ from R3 table, then we have to fetch the related entities: the tuple in S with $id = 1$ and the tuple in C with $id = 1$.


R3: Student - Course

id	Sid	Cid
1_1	1	1
1_2	1	2
2_1	2	1
2_3	2	3

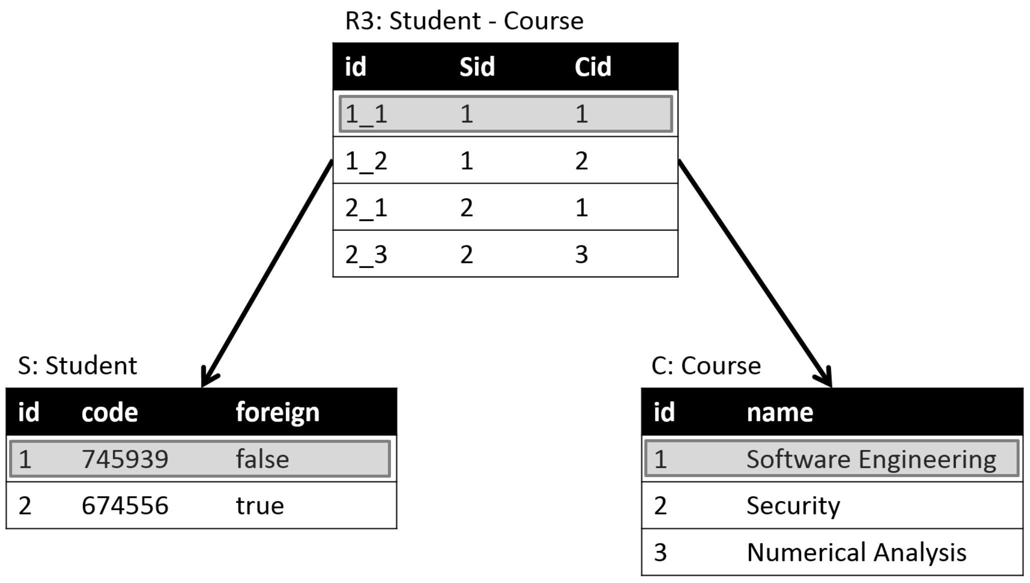
S: Student

id	code	foreign
1	745939	false
2	674556	true

C: Course

id	name
1	Software Engineering
2	Security
3	Numerical Analysis

Figure 4.2: MinR: Entity Update


R3: Student - Course

id	Sid	Cid
1_1	1	1
1_2	1	2
2_1	2	1
2_3	2	3

S: Student

id	code	foreign
1	745939	false
2	674556	true

C: Course

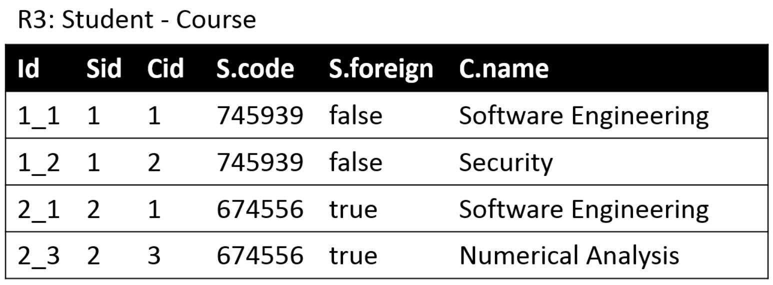
id	name
1	Software Engineering
2	Security
3	Numerical Analysis

Figure 4.3: MinR: Read Entity

Storing only one copy for each entity involves more work, such as we have seen in reading operation example, but ensures that data are always updated to the latest version.

4.1.2 Maximal Replication (MaxR)

In MaxR representation all data are denormalized, entity attributes are copied inside the tuples of the relations. All of these copies are replicas of the same entity and they must be kept in sync, but reading could be faster because there is no need to do the join.



R3: Student - Course

Id	Sid	Cid	S.code	S.foreign	C.name
1_1	1	1	745939	false	Software Engineering
1_2	1	2	745939	false	Security
2_1	2	1	674556	true	Software Engineering
2_3	2	3	674556	true	Numerical Analysis

Figure 4.4: MaxR: Implementation Example

The first version of this model included only the tables of relations 4.4. Later, we also introduced the entity tables, similar to those present in MinR and called here autorelations 4.5.

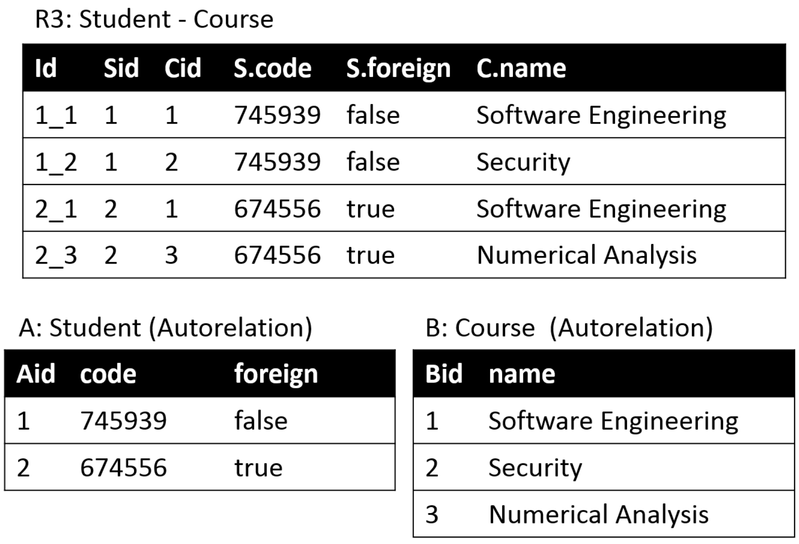
This because if there is a need to read an entity alone and not as part of a relationship, we can read directly from the autorelation with a simple direct access by key (Read Entity).

The extra weight given by this new replica is irrelevant because it is just an update in addition, when in MaxR there is already all other replicas to update. In addition autorelation copies will always be the first to be updated and therefore will always be consistent and write operation will wait only updating the primary copy before releasing control to the application while the additional copies will be updated in the background.

In relation reading nothing changes, there is no need to access to autorelations.

The schema of the model consist of:

- one table for each entity that contains the primary copies, with id and attributes.
- one table to store the relation tuples, each of them contains an identifier, the keys of the related entities, the optional attributes of the


R3: Student - Course

Id	Sid	Cid	S.code	S.foreign	C.name
1_1	1	1	745939	false	Software Engineering
1_2	1	2	745939	false	Security
2_1	2	1	674556	true	Software Engineering
2_3	2	3	674556	true	Numerical Analysis

A: Student (Autorelation)

Aid	code	foreign
1	745939	false
2	674556	true

B: Course (Autorelation)

Bid	name
1	Software Engineering
2	Security
3	Numerical Analysis

Figure 4.5: MaxR: Entity Autorelations

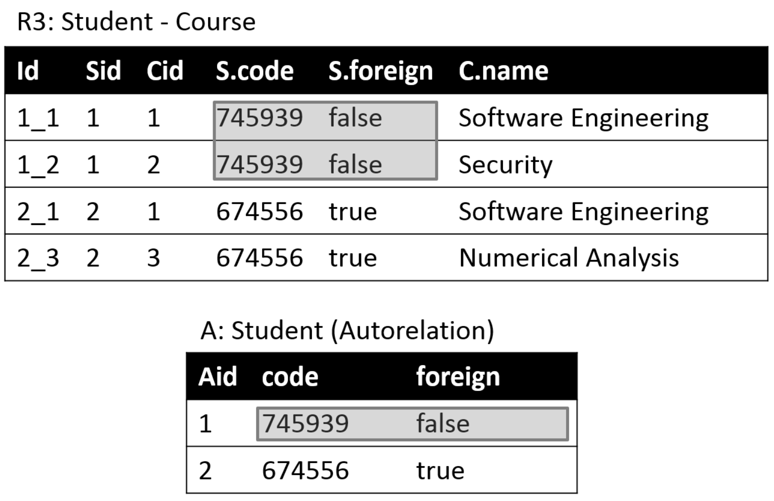
relation and the entities denormalization (Figure 4.5).

It is clear that the process of updating an entity will be heavier than in the Minimal Replication model, because there are many copies to be updated. In particular, the primary copy and all replicas in tables of relations to which it belongs must be upgraded.

As for the MinR model, in Figure 4.6 we report an example of entity updating for the MaxR model. We have to update data of the Student entity with $id = 1$. First, we have to update the tuple in the S autorelation table with $id = 1$. Then for each relation to which this entity belong (in this example we show only R3), we have to update S related data for all tuple with $Sid = 1$ (Sid identify the column of relation table that store S entity ids).

Reading a relation, in MaxR model, translates only in get the tuple in the relation table that will contain the data of the associated entity. In this case might happen temporarily to read stale data, since there are multiple copies of each entity and may not be all updated to same version. It's up to an eventual consistency checker solve the problem if necessary.

The same example we have shown for MinR model is presented in Figure 4.7) for the MaxR model. We have to fetch the R3 relation with $id =' 1\_1'$. So, the only operation to perform in MaxR is the fetch of the tuple with $id =' 1'_1$ from R3 table.



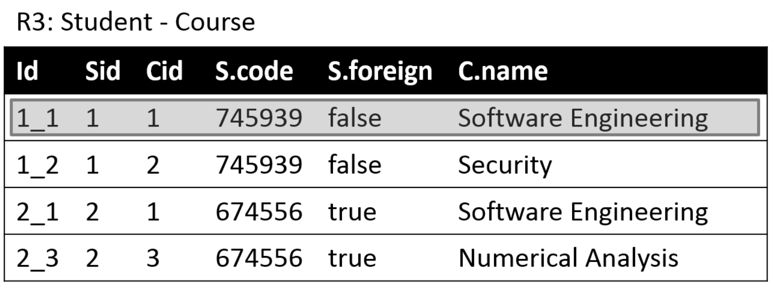
R3: Student - Course

Id	Sid	Cid	S.code	S.foreign	C.name
1_1	1	1	745939	false	Software Engineering
1_2	1	2	745939	false	Security
2_1	2	1	674556	true	Software Engineering
2_3	2	3	674556	true	Numerical Analysis

A: Student (Autorelation)

Aid	code	foreign
1	745939	false
2	674556	true

Figure 4.6: MaxR: Entity Update



R3: Student - Course

Id	Sid	Cid	S.code	S.foreign	C.name
1_1	1	1	745939	false	Software Engineering
1_2	1	2	745939	false	Security
2_1	2	1	674556	true	Software Engineering
2_3	2	3	674556	true	Numerical Analysis

Figure 4.7: MaxR: Read Relation

4.1.3 Common Schema between Models

The two models, while using opposite strategies regarding the replication, have a similar structure for the relations that we can abstract in the "Relation Common Schema" shown in Figure 4.8.

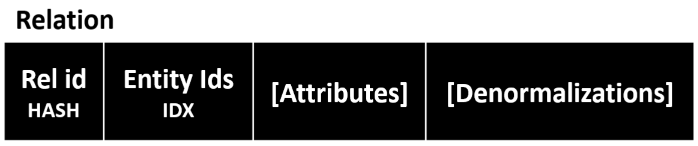


Figure 4.8: Relation Common Schema

The "Relation Common Schema" has an identifier, `Rel id`, a list of connected entities identifier, `Entity Ids`, optional attributes and optional entity denormalizations (in MinR these two optional parts are absent, in MaxR there exist).

The relation identifier is a composite key, created by composing the related entity keys, i.e. using concatenation. This is used as a sharding key like we seen in Marijà overview.

Reading relation by `Rel id` is the most efficient method, because it is performed with an hash access in constant time.

The field in `Entity Ids` block are indexed to enable an efficient access of relations with one or more of the connected entity keys. These indexes can be implemented in different way depending on the features offered by the actual NoSQL used for the implementation (see Sections 5.3.2 and 5.3.2 for indexing techniques used in our implementations) but the commonly an Hash Map with an ordered list of keys or a Btree are used.

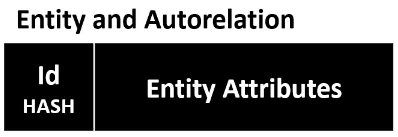


Figure 4.9: Entity Common Schema

Also MinR entity tables and MaxR autorelations can be abstracted with a single model named "Entity Common Schema". In the Figure 4.9 we show the "Entity Common Schema". It consist of an `Id` column that contains the entity keys and the `Entity Attributes` block that contains entity data. Obviously the entities could be fetched by key with an hash access.

The MaxR autorelations can be seen as a special relations with only one entity linked and with `Rel Id` equal to this only entity key (that is left out).

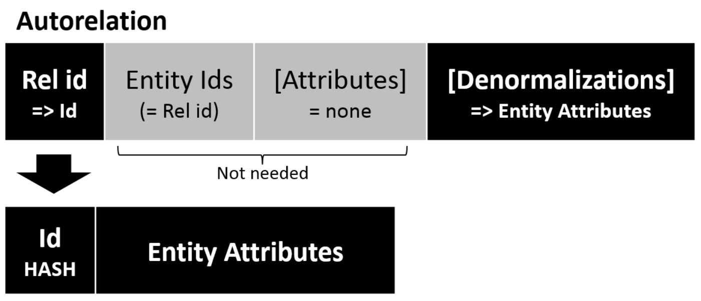


Figure 4.10: The equivalence of MaxR Autorelation Schema and Entity Common Schema

Then the “Autorelation Schema” is equivalent to “Entity Common Schema”, as shown in Figure 4.10.

Index related choice We decide to model efficient data access only for the primary keys and the `Entity Ids` block. This could seem to be a lack of the modeling, but there is a good reason for this choice.

As we have shown in previous chapter, the Marijà Framework optimize a set of queries by ensuring that local data of a node are the only that are necessary to perform queries allocated to that node. So, on the local node should be created efficient way to access data in the way aligned to the query.

However, each node must response to all possible queries on its data. Not only to the optimized ones. And must meet certain performance constraints. So, we have to test the worst case. And the worst case is the simple scan of data, without optimizations.

4.2 Comparing MinR and MaxR Models

4.2.1 Finding a Complete Method to Compare Models

The next step is figuring out when a model is preferred over another. We need to fully compare the performance of two models in reading and writing. To achieve this objective we base our comparison on the classic CRUD (Create, Read, Update, Delete) operations applied to both entities and relations.

The costs of these operations will be the basis for the design choices.

While Create, Update and Delete operations can be executed in an univocal way, Read operation shows several variants. For example a Read operation could be simply find an object by its key, another example could be the select all objects in a collection that satisfy a predicate on an object attribute. This is the reason because in the Section 4.6 we introduce the Relational Algebra as a bridge model to ensure completeness od Read comparison.

4.2.2 Assumptions and Notations

In this section we introduce the assumptions and notations that are valid in all this thesis.

Assumptions

- Relations are always binary.
- A relation has no attributes.
- Two relations are equal if they have the same entity identifiers.
- Entities have the same dimension in terms of memory occupation.
- The MinR in-memory join has no cost.
- Check for object existence needs the object fetching
- Update operations need the object fetching
- Delete operations do not need the object fetching

Notes The MinR in-memory join is the operation used to connect relation object and related entities. The relative cost is dependent on used data structure. For example a pointer-based data structure could really has no cost.

Notes The last three assumption are done in accord to general NoSQL databases features.

Notations

In the Table 4.3 we define symbols that we use to describe operations and algorithms. In the Table 4.1 we formalize name and description of the CRUD operators that are the subjects of tests. In the Table 4.2 we define the low level operations with a description, a costs function (Cost) and the amount of data to be processed (Data Processed). When we use a low level operation in an algorithm step, its “Cost” enters in the total cost of algorithm. Also a `proc_data(s)` is added to the total cost, with `s` equal to operation “Data Processed”.

Operations	**On entities**	**On relations**	**Description**
Create	createE	createR	create an object
Read	findE	findR	read an object by key
	selectE	selectR	read objects that satisfy a predicate
Update	updateE	updateR	update an object
Delete	deleteE	deleteR	delete an object

Table 4.1: Crud Operators

Operations	**Descriptions**	**Costs**	**Data Processed**
find(X)	fetch 1 tuple of X by key	$const$, hash access	$size(X)$
scan(X)	a scansion of X that filter and return data	$\#X * const$, serial access	$\#X * size(X)$
ekeys(R, kE)	fetch kE tuples of R by entity keys	$kX * find(R\_Xidx)$, index access	$kR * size(ID)$
proc_data(s)	elaborate and transfer data of s size		
insert(X)	create 1 object in X	$size(X) * write$	$size(X)$
update(X)	update 1 object of X	$size(X) * write$	$size(X)$
delete(X)	remove 1 object from X	$delete$	$size(ID)$

Table 4.2: Operations

Symbols	Descriptions
X	set
x	instance of X
E	primary entity set (predicates are related only with E)
R	relation set
r.ids	list of relation entity ids
R_Xidx	index of R by X keys
#X	X cardinality
{R}	set of entities related to relation R
{E}	set of relations related to entity A
size(X)	memory print of one tuples of X

Table 4.3: Symbols

4.3 Comparing Create Operator

Create new entity: createE(E)

MinR and MaxR: createE(E) needs to check if an entity with the same key already exists, if no the new entity is created. There is no difference between models.

```
createE(E) =
    find(E) + proc_data(size(E))) +   (1)
    insert(E) + proc_data(size(E)))   (2)
```

Algorithm 4.1 MinR and MaxR: Create Entity

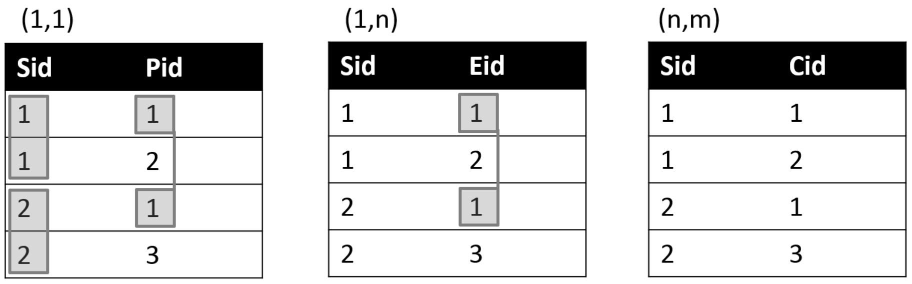
```
1: Find $e$ in $E$ by key
2: if if $E$ already exist then                ▷ (1)
3:     exit
4: end if
5: Insert entity $E$                           ▷ (2)
```

Create new relation: createR(R)

Creating a relationship means connecting two entities between them. Then a check is needed to ensure that the entities exist before insertion.

Relation Type Constraints In the RDBMS every type of relationship has a special structure. The relational type constraints are enforced directly by these structures.

Instead, in Marijà all relations are stored in the same manner. This is similar to the RDBMS many-to-many relation structure. To ensure that the type constraints are satisfied, a check is needed in the objects creation phase. We define a $checkConstraints(R, a_{id}, b_{id})$ function that predicates on R or the associated R_Xidx index to verify if there are any relational type constraint violations. This function will be called in the `createR(R)` operator (cf. Section 4.3). A graphical example of constraint violations are shown in Figure 4.3.


(1,1)

Sid	Pid
1	1
1	2
2	1
2	3

(1,n)

Sid	Eid
1	1
1	2
2	1
2	3

(n,m)

Sid	Cid
1	1
1	2
2	1
2	3

Figure 4.11: Relation type constraint violations

Definition of $checkConstraints(R, a_{id}, b_{id})$ **Function:** [1]

- **case (1,1)**: `not (R.Aid = a_id  or  R.Bid = b_id)`, if a and b are not already related with other entities. Check it with two *ekeys* access at R_Xidx.
- **caso (1,n)**: `not (R.Bid = b_id)`, if b is not already related with other entities. Check it with one *ekeys* access at R_Xidx.
- **caso (n,m)**: `not (R.Aid = a_id  and  R.Bid = b_id)`, if this relation within a and b is not already exist. Check it on R by composing relation key with entity keys and trying to $find$ it.

In the cost function we assume always worst case: $\#\{R\} * ekeys(R, 1)$.

[1]HP We assume it is a binary relation: $A \leftrightarrow B$, a_{id} and b_{id} are respectively the identifier of A and B entities to be connected

MinR: Check that the entities involved exist, verify the relation type constraint by checking if exist some tuples that satisfy the $checkConstraints(R, a_{id}, b_{id})$ control function e in R_Xidx. If all constraints are satisfied we insert the relation.

```
createR(R) =
    #{R}*(find(X) + proc_data(size(X)) +            (1)
    #{R}*(ekeys(R, 1) + proc_data(#{R}*size(ID))) +  (2)
    insert(R) + proc_data(#{R}*size(ID))             (3)
```

MaxR the algorithm is the same of the MinR one. The only difference is in the amount of data to be written in relation tuple: $\#\{R\} * size(X)$ instead of $\#\{R\} * size(ID)$. A minimal difference.

```
createR(R) =
    #{R}*(find(X) + proc_data(size(X)) +            (1)
    #{R}*(ekeys(R, 1) + proc_data(#{R}*size(ID))) +  (2)
    insert(R) + proc_data(#{R}*size(X))              (3)
```

Algorithm 4.2 MinR and MaxR: Insert Relation

```
1: // related entities existence
2: for each $X$ in $\{R\}$ do                                  ▷ (1)
3:     Find $X$
4:     if if $X$ already exist then
5:         exit
6:     end if
7: end for
8: // type constraint check
9: Check if $checkConstraints(R, a_{id}, b_{id})$ is verified  ▷ (2)
10: if if it is not verified then
11:     exit
12: end if
13: // insert relation
14: Insert relation in R                                       ▷ (3)
```

4.4 Comparing Update Operator

Update Entity: updateE(E)

In MinR there is no replication at all, then the changes is to be applied only once on the entity. In MaxR there are a lot of entity copies in the relations collections. All copies must be synchronized and so they must be updated together.

The Update Entity represent the other big difference between models. MinR is better then MaxR when there are at least some entity replicas.

We assume that fetching of entity is needed before the updating. In some implementations the update could be executed in-place [2].

MinR: updateE(E) is simply traduced in the fetching and the updating of the entity.

```
updateE(E) =
    find(E) + proc_data(size(E))) +   (1)
    update(E) + proc_data(size(E)))   (2)
```

Algorithm 4.3 MinR: Update Entity

```
1: Find $e$ in $E$ by key        ▷ (1)
2: Update the entity $e$         ▷ (2)
```

MaxR: Firstly the entity is fetched from the autorelation E and directly updated. Then for all relations to which it belongs, all entity replicas are retrieved and updated.

```
createE(E) =
    find(E) + proc_data(size(E))) +                   (1)
    update(E) + proc_data(size(E))) +                 (2)
    #{E}*(
      ekeys(R, kE) + proc_data(kR*size(ID)))) +       (3)
      kR*find(R) + proc_data(kR*2*size(ID)))) +       (4)
      kR*update(R) + proc_data(size(E)))              (5)
    )
```

[2]Update in-place: fetching object is not needed and only the attributes to be updated are sent to database

Algorithm 4.4 MaxR: Update Entity

```
 1: // update primary entity
 2: Find $e$ in $E$ by key                                   ▷ (1)
 3: Update the entity $e$                                    ▷ (2)
 4: // update entity replicas in related relations
 5: for each $R$ in $\{E\}$ do
 6:     // retrieve related relations
 7:     Retrieve from $R\_Xidx$ the relation keys            ▷ (3)
 8:     Fetch the relations $rs$ from $R$ by keys            ▷ (4)
 9:     for each $r$ in $rs$ do                              ▷ (5)
10:         Update the relation $r$
11:     end for
12: end for
```

Update Relation: updateR(R)

The "Update Relation" operator cannot change the identifier of a tuple and the Entity Ids in a relation tuple, but only other attributes.

MinR and MaxR: updateR(E) is simply traduced in the fetching and the updating of the relation. There are no difference between models.

```
updateR(R) =
    find(R) + proc_data(size(R))) +   (1)
    update(R) + proc_data(size(R)))   (2)
```

Note that in MinR $size(R) = size(\{R\} * ID)$ and in MaxR $size(R) = size(\{R\} * size(E))$

Algorithm 4.5 MinR: Update Relation

```
1: Find $r$ in $R$ by key                                    ▷ (1)
2: Update the relation $r$                                   ▷ (2)
```

4.5 Comparing Delete Operator

Delete Entity: deleteE(E)

We assume that fetching is not needed to delete an object. We also assume that delete cost is not dependent on the size of tuple.

MinR and MaxR: deleteE(E) needs to check if the entity exists. Then, for all relations to which the entity belongs, all replicas are retrieved and updated

Finally perform the primary entity delete. There are no difference between models.

```
deleteE(E) =
    find(E) + proc_data(size(E))) +              (1)
    #{E}*(
      ekeys(R, 1) + proc_data(kR*size(ID))) +    (2)
      kR*(delete(R) + proc_data(size(ID))) +     (3)
    )
    delete(E) + proc_data(size(ID))              (4)
```

Algorithm 4.6 MinR and MaxR: Delete Entity

```
1:  Find $r$ in $R$ by key                                ▷ (1)
2:  if $r$ not exists then
3:      exit
4:  end if
5:  // delete all entity replicas in related relations
6:  for each $R$ in $\{E\}$ do
7:      // retrieve related relations
8:      Retrieve from $R\_Xidx$ the relation keys         ▷ (2)
9:      for each $r$ in $rs$ do                           ▷ (3)
10:         Delete the relation $r$
11:     end for
12: end for
13: Delete $r$                                            ▷ (4)
```

Delete Relation: deleteR(R)

Delete a relation only means remove the connection between entities without deleting these.

We assume that delete cost is not dependent on the size of tuple.

MinR and MaxR: deleteR(R) needs to check if the relation exists and then perform delete. There are no difference between models.

```
deleteR(R) =
    find(R) + proc_data(size(R))) +  (1)
    delete(R) + proc_data(size(ID))  (2)
```

Algorithm 4.7 MinR and MaxR: Delete Relation

```
1: Find $r$ in $R$ by key                              ▷ (1)
2: if if $r$ not exist then
3:     exit
4: end if
5: Delete $r$                                          ▷ (2)
```

4.6 Comparing Read Operator

4.6.1 Using Relational Algebra to Ensure Completeness of Read Comparison

As we introduced in the beginning of the chapter the Read operation shows several variants. To ensure that our reading comparison between models is complete, we have decided to use the Relational Algebra (RA) as intermediate model. Starting from basic operators of RA, any complex query can be defined as a function of these. So, analyzing the mapping of RA basic operators on MinR and MaxR we will identify a set of Read operations that cover all possible reading scenario.

The RA basic operators are:

- Union
- Difference
- Cartesian Product
- Selection
- Projection
- Rename

Concerning the other three operators, we can assume that rename has no cost at all. Sometimes the projection is provided directly by database, else it is also performed on the in-memory structure without difference in models. Finally the selection is executed within the fetch operation.

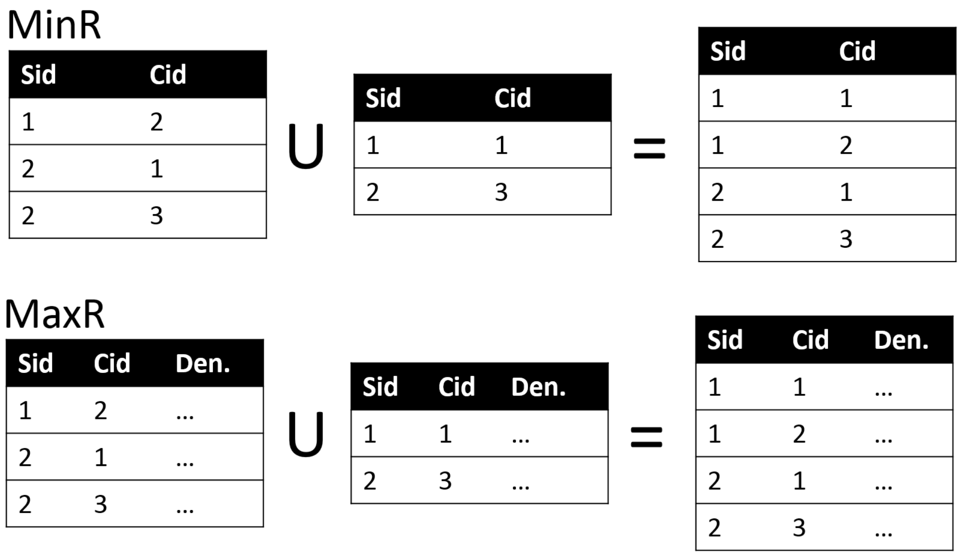
Then the Read operations that we will test are the simple find by key and the fetching of multiple objects with a query predicate.

Here, for completeness, a description of the relational algebra primitive operators.

Set Operators

Under this category fall Union, Difference and Cartesian Product. By analyzing these operators we have realized that they can be applied only once data is loaded into memory. They work only on the *Entity Ids* block of the Relation Common Schema, that is introduced in Section 4.1.3. So the operations performed in both MinR and MaxR are the same. By assumption two relations are equal if they have the same entity identifiers.

An example of Union operation on both models is shown in Figure 4.12 (we do not show the entity tables). As we can see the result tuples contain the same pairs of Entity Ids, so they are equivalent. Denormalizations block is not relevant, indeed the values are not shown. Since there is a possibility that data are not consistent in MaxR, in different tuples that denormalize the same entity x (with the same Xid in Entity Ids) we could find two different entity versions in Denormalizations block. However, this is not relevant for the success of a set operation. Simply in the results could be found the first version, the second, or even both of them.


MinR

Sid	Cid
1	2
2	1
2	3

U

Sid	Cid
1	1
2	3

=

Sid	Cid
1	1
1	2
2	1
2	3

MaxR

Sid	Cid	Den.
1	2	...
2	1	...
2	3	...

U

Sid	Cid	Den.
1	1	...
2	3	...

=

Sid	Cid	Den.
1	1	...
1	2	...
2	1	...
2	3	...

Figure 4.12: Set Operation Equivalence on MinR and MaxR

There is only an extra cost for MinR to delete unnecessary entities at the end of all operations, but it is irrelevant. If the implementation is well done, this final activity has a very small cost compared to the rest.

When reading without a constraint on consistency, then for example without waiting for the conclusion of active updates, it may happen that multiple replicas of the same entity have different attribute values. It is not relevant for the operators job, they only work on keys and finally the results will contain one version of those available.

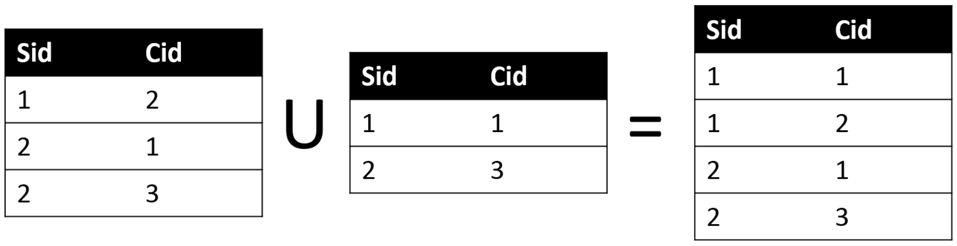
Remember that our goal is to compare the two models and on these operators the two strategies differ only in fetching relations. All set operators are constructed as follows:

```
R1 = fetch('R1', p1)
R2 = fetch('R2', p2)
results = apply_set_operator(R1, R2)
```

where `p1` and `p2` are predicate used to filter relations and `setOperator` is one between union, difference and Cartesian product.

In the next paragraphs we show how each set operator is mapped on our models with a short description and a graphical example. In the examples we will use the MinR model, but as we said before MaxR is equivalent.

Union (∪) returns data that belong to at least one relation, without repetitions. The two relations involved must be union-compatible that is, the two relations must have the same set of attributes. Example in Figure 4.13.



Sid	Cid
1	2
2	1
2	3

∪

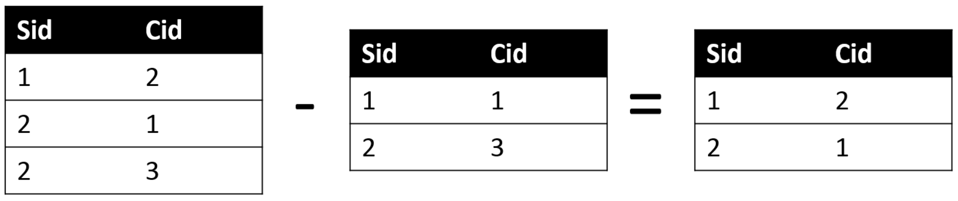
Sid	Cid
1	1
2	3

=

Sid	Cid
1	1
1	2
2	1
2	3

Figure 4.13: Union Example

Difference (\ ) returns data that belong to only the first set and not two the second one. The two relations involved must be union-compatible that is, the two relations must have the same set of attributes. Example in Figure 4.14.



Sid	Cid
1	2
2	1
2	3

-

Sid	Cid
1	1
2	3

=

Sid	Cid
1	2
2	1

Figure 4.14: Difference Example

Cartesian Product (×) returns all combination of tuple of the first relation concatenated with all of the second one. For the Cartesian product to be defined, the two relations involved must have disjoint headers that is, they must not have a common attribute name. In most case (i.e. for implementing a join) a renaming of attributes is needed before using Cartesian product. Example in Figure 4.15.

Sid	Pid
1	2
2	1
2	3

X

Cid	Eid
1	1
2	3

=

Sid	Pid	Cid	Eid
1	2	1	1
1	2	2	3
2	1	1	1
2	1	2	3
2	3	1	1
2	3	2	3

Figure 4.15: Cartesian Product Example

Special Relational Operators

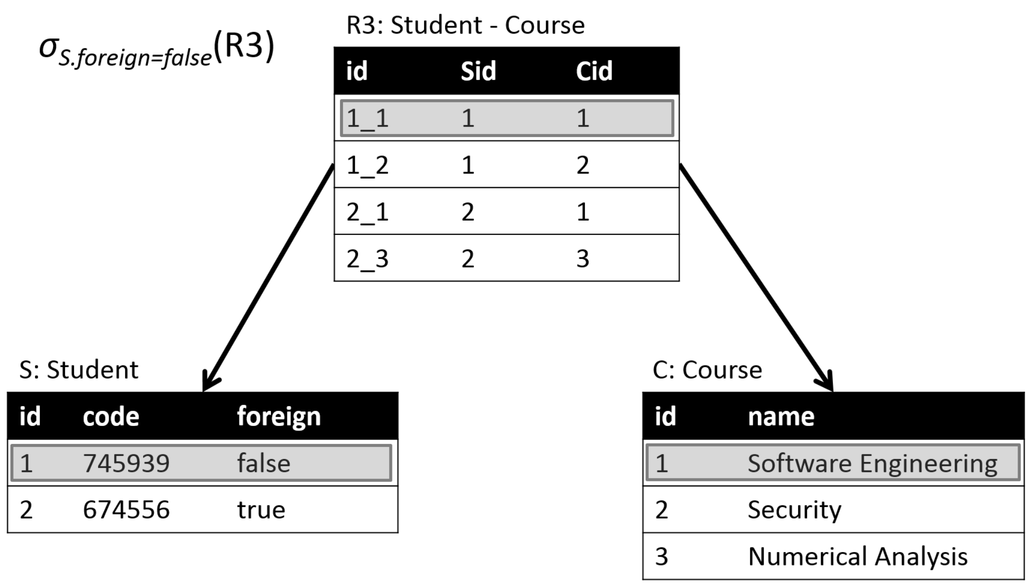
These operators are used to change structure of a single relation or to filter data. The operation that change the relation structure cannot be applied on the identifier of relation and entities (`Rel Id`, `Entity Ids` in Relation Common Schema and `Id` in Entity Common Schema) but only on other columns (Attributes and Denormalizations). This because all operators working on a Marijà relation must return another Marijà relation, following the Common Schema between Models (cf. Section 4.1.3).

They can be executed directly in fetch operation or also on an in-memory relation.

Selection (σ) is the operator used to filter data of a relation. It returns all tuples that satisfy a Boolean predicate on attributes. Example in Figures 4.16 and 4.17.

Projection (π) is the operator used to cut off some attributes from a relations. It returns all rows of a relation but with only the columns (attributes) that are passed as input. Example in Figures 4.18 and 4.19.

Rename (ρ) is the operator used to change the attribute names. It returns the entire relation but changing the header (attribute names). Example in Figures 4.20 and 4.21.



R3: Student - Course

id	Sid	Cid
1_1	1	1
1_2	1	2
2_1	2	1
2_3	2	3

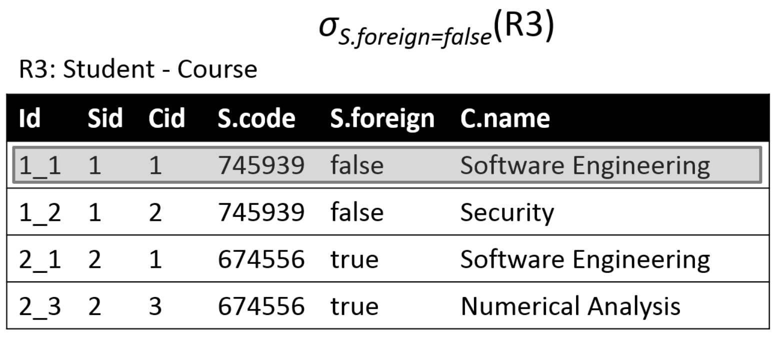
S: Student

id	code	foreign
1	745939	false
2	674556	true

C: Course

id	name
1	Software Engineering
2	Security
3	Numerical Analysis

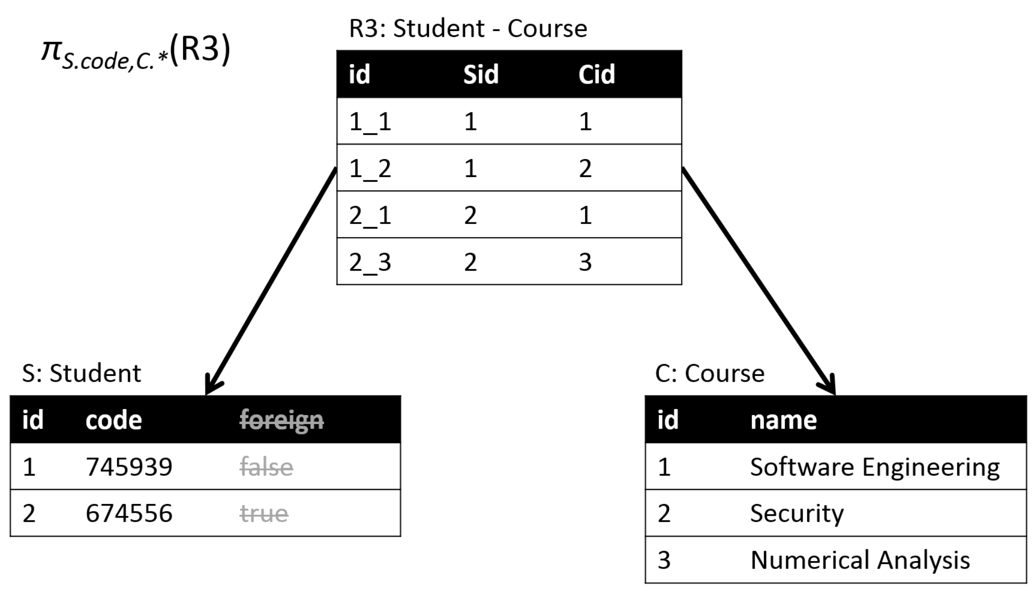
Figure 4.16: MinR: Selection Example


$\sigma_{S.foreign=false}(R3)$

R3: Student - Course

Id	Sid	Cid	S.code	S.foreign	C.name
1_1	1	1	745939	false	Software Engineering
1_2	1	2	745939	false	Security
2_1	2	1	674556	true	Software Engineering
2_3	2	3	674556	true	Numerical Analysis

Figure 4.17: MaxR: Selection Example



$\pi_{S.code,C.*}$(R3)

R3: Student - Course

id	Sid	Cid
1_1	1	1
1_2	1	2
2_1	2	1
2_3	2	3

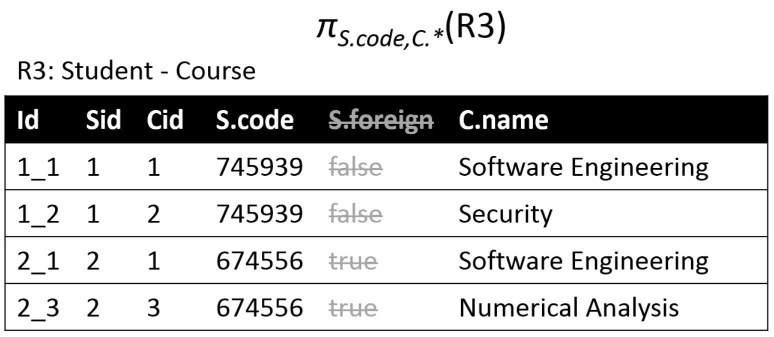
S: Student

id	code	~~foreign~~
1	745939	~~false~~
2	674556	~~true~~

C: Course

id	name
1	Software Engineering
2	Security
3	Numerical Analysis

Figure 4.18: MinR: Projection Example

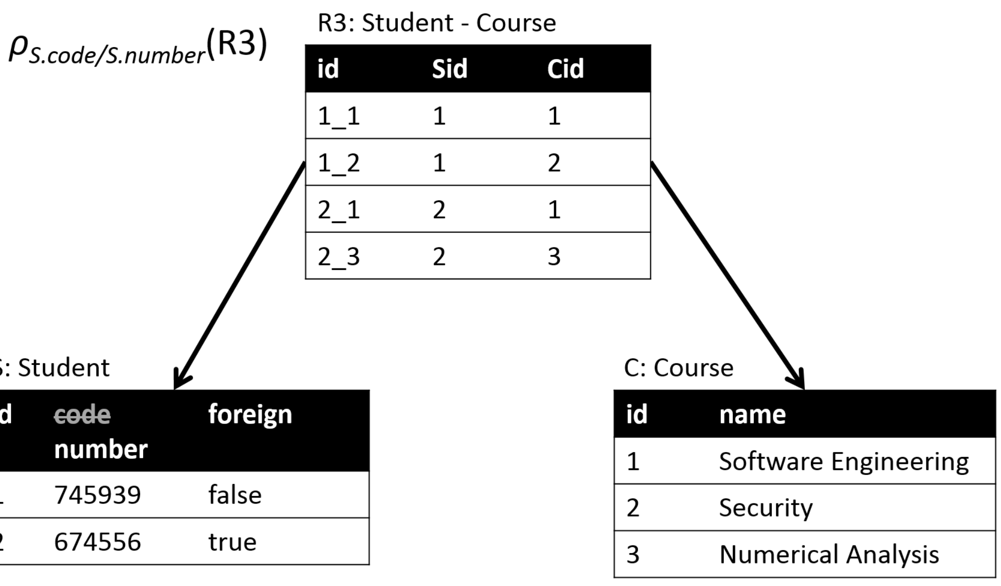


$\pi_{S.code,C.*}$(R3)

R3: Student - Course

Id	Sid	Cid	S.code	~~S.foreign~~	C.name
1_1	1	1	745939	~~false~~	Software Engineering
1_2	1	2	745939	~~false~~	Security
2_1	2	1	674556	~~true~~	Software Engineering
2_3	2	3	674556	~~true~~	Numerical Analysis

Figure 4.19: MaxR: Projection Example



$\rho_{S.code/S.number}(R3)$

R3: Student - Course

id	Sid	Cid
1_1	1	1
1_2	1	2
2_1	2	1
2_3	2	3

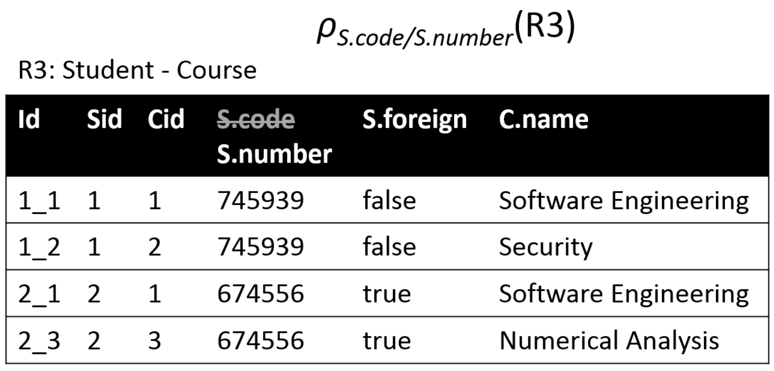
S: Student

id	~~code~~ number	foreign
1	745939	false
2	674556	true

C: Course

id	name
1	Software Engineering
2	Security
3	Numerical Analysis

Figure 4.20: MinR: Rename Example



$\rho_{S.code/S.number}(R3)$

R3: Student - Course

Id	Sid	Cid	~~S.code~~ S.number	S.foreign	C.name
1_1	1	1	745939	false	Software Engineering
1_2	1	2	745939	false	Security
2_1	2	1	674556	true	Software Engineering
2_3	2	3	674556	true	Numerical Analysis

Figure 4.21: MaxR: Rename Example

Identification of Read operators

In conclusion to what we have said in previous paragraphs the only operator that is needed to compare is the fetching of multiple relations with a predicate. We call this operator `selectR`. Entity table can be seen as a particular form of relation, an Autorelation (cf. Section 4.1.3), but as we will present the algorithm change a bit. So we will compare also this operation variant, named `selectE`.

Finally we add to comparison also the simple operation to find a single object by its key, that is the basic Read operation. We denote this as `findR` and `findE` for the finding of a relation and an entity respectively.

Find Entity: findE(E)

MinR and MaxR: findE(E) needs to compute the hash of entity key and retrive the object. There are no difference between models.

```
findE(E) = find(E) + proc_data(size(E))
```

Find Relation: findR(R)

MinR: findR(R) needs to compute the hash of entity key and retrive the object. There are no difference between models.

```
findR(R) =
    find(R) + proc_data(#{R}*size(ID)) +  (1)
    #{R}*(find(X) + proc_data(size(X)))   (2)
```

Algorithm 4.8 MinR: Find Relation

```
1: // retrieve relation tuple
2: Find $r$ in $R$ by key                          ▷ (1)
3: // retrieve related entities
4: for each $x\_id$ in $r.ids$ do                  ▷ (2)
5:     Find entity $x$ by key $x\_id$
6: end for
```

MaxR findR(R) only needs to fetch the relation tuple by key that contains also the related entities data.

```
findR(R) = find(R) + proc_data(#{R}*size(X)
```

Select Entity: selectE(E, p)

MinR and MaxR: selectE(E, p) needs to filter entity collection (or autorelation that is equivalent as said in Section 4.1.3) to satisfy *p* predicate. There are no difference between models.

```
selectE(E, p) = scan(E) + proc_data(kE*size(X)
```

Select Relation: selectR(R, p)

The selection of a relation could be done with a predicate that filter more than one entity, or the relation itself. For MaxR nothing change because all information are stored in relation tuple. For MinR the algorithm is different. However the case with a predicate over the *R* is not so interesting. It differ from MaxR one only for the additionally find by key needed to fetch the entities. In our implementations we have also cover this case and we will explain it in this section, but from here, if not differently specified, "Select Relation" means select relations that verify a predicate on one entity.

To simplify algorithms we present only the case where the predicate is relative to only one entity, the entity *E*. If the predicate is complex it can be always splitted into a combination of simple ones. The results of the fetch with first predicate can be filtered in a second time with others. Some optimization could be done, but needs more investigations.

In "Select Relation" we can see the first big difference between the models. In MaxR only the *R* collection scan is needed, in MinR the collection *E* is to be scanned and after there are others operation to perform. The difference between the two scans and the weight of others operations are the elements to compare.

MinR: In the case of predicate over an entity set, firstly the *E* set is to be scanned and filtered regards the selection predicate (*scan*). Then for each *E* key in the result, the related relation ids are retrieved from *R_Xidx*. Successively all relations in *R* is fetched by these keys (*ekeys*). Finally all other entities $\in \{R\}$ are be loaded with the keys extracted from relations fetched (*find*).

```
selectR(R, p) =
    scan(E) + proc_data(kE*size(E)) +             (1)
    ekeys(R, kE) + proc_data(kR*size(ID))) +      (2)
    kR*find(R) + proc_data(kR*2*size(ID))) +      (3)
    (#{R}-1)*kX*(find(X) + proc_data(size(X)))    (4)
```

Check the notations in Section 4.2.2 how data processed by *ekeys* operation is calculated.

Algorithm 4.9 MinR: Select Relation

```
 1: // scan and filter primary entity set
 2: Fetch entities $es$ in $E$ that satisfy predicate $p$          ▷ (1)
 3: // retrieve related relations
 4: Retrieve from $R\_Xidx$ the relation keys                      ▷ (2)
 5: Fetch the relations $rs$ from $R$ by keys                      ▷ (3)
 6: // retrieve related entities
 7: for each $X$ in $\{R\}$ do                                     ▷ (4)
 8:     if $X \neq E$ then
 9:         Get $X\_ids$ from $X$ keys in $rs$ without repetition
10:         for each $x\_id$ in $X\_ids$ do
11:             Find entity $x$ by key $x\_id$
12:         end for
13:     end if
14: end for
```

In the case of the predicate is relative to the relation, the R set is to be scanned and filtered, then all entities linked to results are to be fetched by key.

```
selectR(R, p) =
    scan(R) + proc_data(kR*2*size(ID)) +      (1)
    #{R}*kX*(find(X) + proc_data(size(X)))    (2)
```

Algorithm 4.10 MinR: Select Relation with predicate over relation

```
1: // scan and filter relation set
2: Fetch entities $rs$ in $R$ that satisfy predicate $p$   ▷ (1)
3: // retrieve related entities
4: for each $X$ in $\{R\}$ do                              ▷ (2)
5:     Get $X\_ids$ from $X$ keys in $rs$ without repetition
6:     for each $x\_id$ in $X\_ids$ do
7:         Find entity $x$ by key $x\_id$
8:     end for
9: end for
```

MaxR: selectR(R, p) only needs to fetch relation tuples that satisfy the p predicate, they contains also the related entities data.

```
selectR(R, p) = scan(R) + proc_data(kR*#{R}*size(X)
```

4.7 Criteria for Model Selection

	Operations			**Memory**	
CRUD	**MinR**	**MaxR**	**CRUD**	**MinR**	**MaxR**
createE	$=$	$=$	createE	$=$	$=$
createR	$=$	$=$	createR	$-$	$+$
findE	$=$	$=$	findE	$=$	$=$
findR	$1 + \{R\}$	1	findR	$=$	$=$
selectE	$=$	$=$	selectE	$=$	$=$
selectR	$\sim \#E + kE + kR$	$\sim \#R$	selectR	$--$	$++$
updateE	1	$e_{replica}$	updateE	$--$	$++$
updateR	$=$	$=$	updateR	$=$	$=$
deleteE	$=$	$=$	deleteE	$=$	$=$
deleteR	$=$	$=$	deleteR	$=$	$=$

Table 4.4: MinR vs MaxR: Summary of Difference

Analyzing algorithms and cost functions presented in this chapter we have built the Table 4.4. The table is divided in two sub-tables "Operations" and "Memory". In the first one we have summarized the difference of the models in the number of operations to be performed for every CRUD function. In "Memory" sub-table there are the difference in the amount of data processed and transferred.

What we can conclude is that the only big difference are in the `updateE` and `selectR`. To Update Entity in MinR one write is enough, in MaxR all replicas in the associated relations are to be updated. The gap between the models increases with the growth of the replica number. In Select Relation instead there is not a clear winner. MinR involves more operations but it needs to process less data than MaxR needs.

These two CRUD operators will be investigated with detailed tests in Chapter 6.

Other differences shown in the table are not so relevant. In MaxR `createR` only has to transfer more data than in MinR because it also has to store entity copies in the relation. The algorithms are the same. This means that if we have to create a lot of new entities in a small time the MinR is the best solution.

The gap in `findR` is even less than in `createR` because the amount of processed data is the same. In MinR more request are needed than in MaxR, one for each entity in addition to the relation fetch that is present in both algorithms. The costs tend to be aligned when the entities become big because the requests overhead loose importance. If an application needs to rapidly read very small objects MaxR model is the solution, because the multiple request overhead in MinR lower the performance. If the reads are slow or the quantity of data to read is bigger there is no advantage to use one model against the other.

These operators will not be tested extensively because the conclusions that can be reached are clear. There is nothing else to investigate.

4.8 Consistency and Other Considerations

Other important considerations can be achieved in the consistency topic.

The absence of replication in the MinR model ensures that the entity update is a single and atomic operation at logical level [3].

Instead, update an entity in the MaxR model involves writing all replicas. This complex operation cannot be performed atomically (without using a transaction mechanism). We call `time-to-consistency` the time that is needed to update all replicas of an entity and then to reach full consistency. A test is needed to measure `time-to-consistency` parameter (cf. Section 5.7) for a specific relation.

Also the total dimension of the database is different depending on of which model is used to store relations. Obviously a MaxR relation is bigger than MinR one because of the presence of entity replicas.

4.9 Summary

In this chapter we presented the two alternative strategy to model relationships in a single Marijà node, MinR and MaxR. We described their theoretical differences an we exposed a complete way to comparing the reading and writing operations on them.

[3]The operator implementation must ensure that this single operation is also atomic at the physical layer. Commonly the manipulation of a single tuple is already atomic in NoSQL databases.

For the comparison we based on CRUD operations identified for entities and relations also using the Relational Algebra to find a complete set of Read operations.

We have written the CRUD operation algorithms for both models. Analyzing them we have concluded that the models are complementary.

MaxR is better then MinR in a read-intensive scenario with small entities. Instead MinR is good for a write-heavy scenario.

More investigations are needed for the `selectR` operator. In selecting relations is not clear which model is the best. There is a difference in number of operations, in processed data and in the object set that is to be scanned. We expect that a model is better than the other only in some case.

Other selection criteria is the presence of time constraints on relation consistency. A MinR relation is always strictly consistent. In MaxR the denormalization of entities involves Eventual Consistency.

A test on `updateE` to measure how much MaxR is slower than the MinR is needed. This results can be useful to understand the time-to-consistency parameter introduced in Section 4.8.

Tests and test results are shown in Chapter 5.

Chapter 5

Executing Tests on Models

5.1 Testing Objectives and Context

In this chapter we will describe the tests that we decided to implement for comparing MinR and MaxR models.

Following the conclusions outlined in the Chapter 4, we carried out preliminary tests to compare the behavior of the two models (MinR and MaxR) with respect to each CRUD operator.

These preliminary tests (cf. 5.4) are needed to verify what we mentioned in Section 4.7: the only CRUD operators to analyze in detail are `selectR` and `updateE`, because they are the only that differ so much to be significant.

What we expect is to find a trade-off between the two models by evaluating reading tests (Selection Relation Operator) varying the number and size of entities and relations. From the Update Entity Operator tests instead we want to understand what is the burden of updating many replicas in the MaxR model compared to the single update required by MinR.

These tests will be used to decide which of the two logical models is the best for each kind of relation, given some of its properties. In addition of this, they will also lead to decisions about the "Sharding Point" and, more generally, on the maximum size of a single node relationships.

To execute tests we used the Evaluation Framework that we have developed, described in Chapter 6. In particular we have implemented the database provider for MongoDB and Apache Cassandra.

We had to choose what NoSQL to use for our tests. There were some prerequisites to meet:

- Support to *scan* feature. As we have said in Section 4.1.3 we have to test data reading without any read optimization technique to achieve our objectives

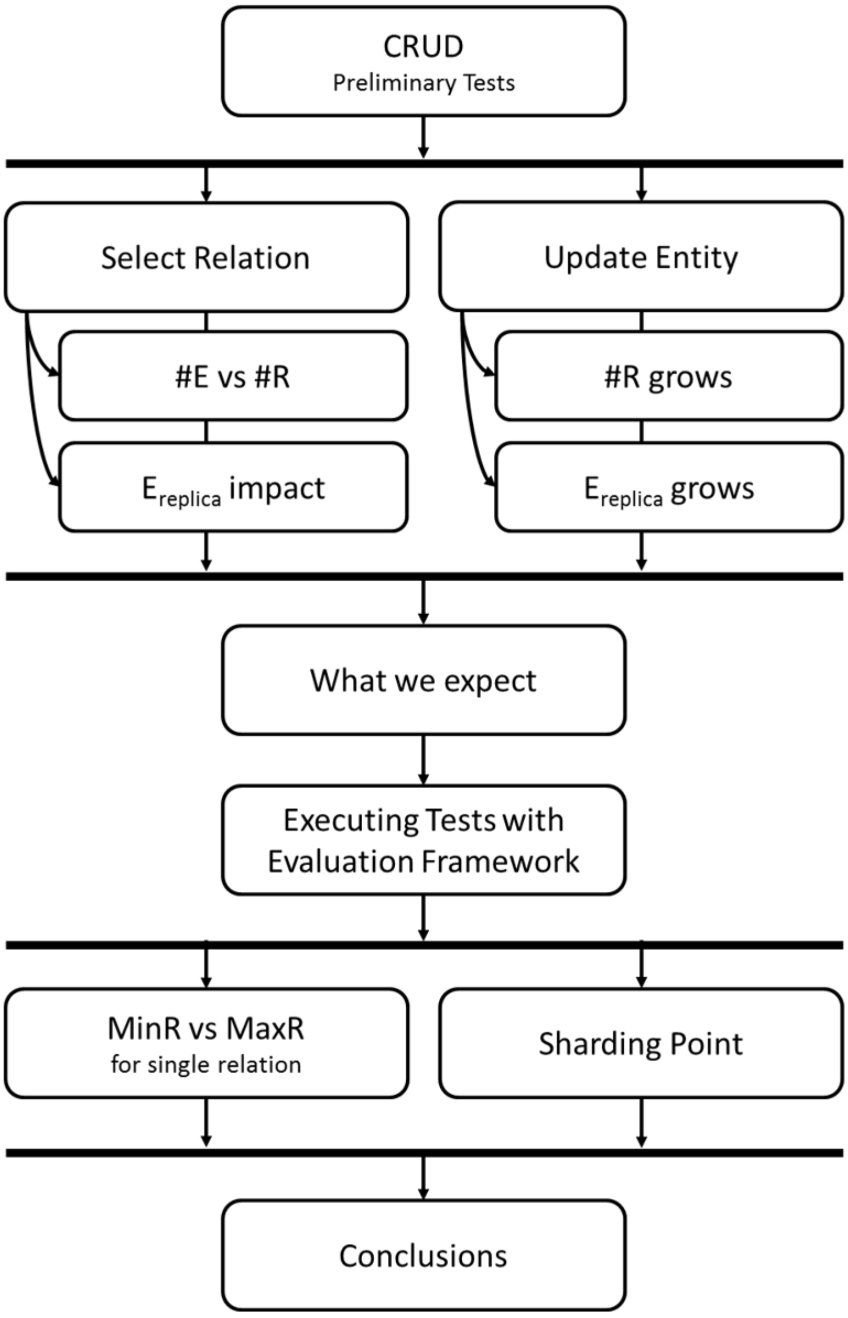


Figure 5.1: Testing Workflow

- A good documentation
- The availability of driver for Microsoft .NET, the technology used to develop the Evaluation Framework

We have chosen MongoDB and Cassandra because they meet our prerequisites and they are very different: MongoDB, a Document-based db optimized for readings and to maximize single node performance, and Cassandra, a Column-oriented db designed for a write-frequently distributed scenario. We have not chosen a Key-Value database because of the first prerequisite: most of the Key-Values do not provide *scan* feature.

We expect to see different results in the two implementations because the databases are very different. In particular Cassandra, as a Column-oriented database, needs additional data structures to ensure efficient readings (secondary indexes). As explained in Section 4.1.3 in our Common Schema only the *Entity Ids* block is indexed (in addition of the primary key), so we expect that Cassandra should not have great reading performance.

In Figure 5.1 we have reported a schema of the testing flow that summarize this points:

- Preliminary Tests on CRUD operators
- Select Relation and Update Entity detailed tests
- What we expect by analyzing tests theoretically
- Execution of tests with the Evaluation Framework on Marijà node
- Analysis on MinR vs MaxR and Sharding Point (the test objectives)
- Conclusions

5.2 Logging methods

To measure the time costs of the operators we have created two kind of logging methods. One at the level of the entire operators (coarse) and one for the individual operations on the database (fine).

We used fine grained logs on single database calls to understand what are the critical points in the algorithms. Once we have found them and fixed the tests to run, we have re-executed tests with coarse grained logging configuration. With this we have produced the results of our tests and evaluated the difference between models.

For example in "Select Relation" operator the MinR algorithm requires three calls to databases: $scan(E)$ to filter the primary entity set, $ekeys(R, keys_E)$ to find R tuples related to these entities and $findByKeys(F, keys_F)$ to fetch the F entities that are needed (that is assumed equals to $kF * find(F)$). The coarse grained logging registers the time of entire "Select Relation" and the fine grained logging registers the times of the three database calls.

Each block of code that we have wanted to log has been re-executed many times, how many times is a configuration parameter, and with the data obtained we have estimated mean, variance, minimum and maximum value, with these statistical formulas:

$$\begin{aligned}
\overline{x} &= \frac{1}{n}\sum_{i=1}^{n} x_i && : mean \\
\sigma^2 &= \frac{1}{n}\sum_{i=1}^{n} (x_i - \overline{x})^2 && : variance \\
min(X) &= min\,(x_i \in X) && : minimum \\
max(X) &= max\,(x_i \in X) && : maximum
\end{aligned}$$

Where n is the number of test rounds and X represents the set of data samples for the operation that we have to log.

5.3 Experimental Settings

5.3.1 Hardware and Software Settings

Tests are done on a single desktop machine with these characteristics:

Intel(R) Core(TM) i5-3750K CPU @ 3.40 GHz
8,00 GB DDR3
160GB Intel X-25M G2 Solid-State Hard Drive
Windows 8 Pro 64-bit

The NoSQL databases used for testing are MongoDB 2.2.3 (2013-02-01) and DataStax Community edition of Apache Cassandra 1.2.2 (2013-02-20) installed and running one at a time on the system.

The Evaluation Framework that we will present in Chapter 6 is written on the Microsoft .NET Framework.

To work with MongoDB we have used the official 10gen-supported C# / .NET driver for MongoDB version 1.7 [1]. The driver is very stable and mature and allows to use all the features of MongoDB. Several other drivers have been developed by the community, but we chose to use the official one.

For Cassandra there is no high-level official drivers, but there is a list of recommended projects as a preferred alternative instead of using raw Thrift interface [12] directly.

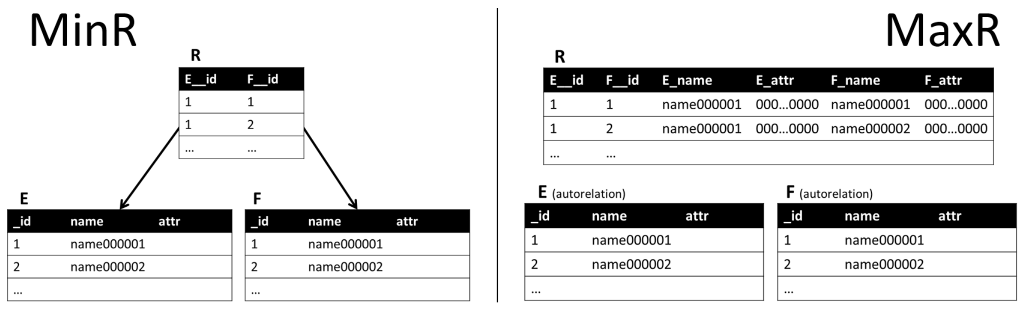
We chose FluentCassandra open source project [37], because it is simple to use and it implement all commands present in last version of Apache Cassandra.

We evaluated other alternatives such as Aquiles, Cassandraemon and cassandra-sharp, but the first two implement high-level mechanisms that might mask some behaviors which we want to keep under control and cassandra-sharp is not mature enough yet.

As we will see in the results below, we believe that even Fluentcassandra still has something to be fixed, in particular as regards long connections in time and managing a high amount of data. In our tests we had to refresh the connection with each new measurement because we noticed a kind of accumulation phenomenon on subsequent operations.

In practice when we did the same thing several times the times were always growing. Instead with the connection refresh between two measurements the times became very similar.

5.3.2 Database Schema used for Testing



MinR — R

E__id	F__id
1	1
1	2
...	...

MinR — E

_id	name	attr
1	name000001	
2	name000002	
...		

MinR — F

_id	name	attr
1	name000001	
2	name000002	
...		

MaxR — R

E__id	F__id	E_name	E_attr	F_name	F_attr
1	1	name000001	000...0000	name000001	000...0000
1	2	name000001	000...0000	name000002	000...0000
...	...				

MaxR — E (autorelation)

_id	name	attr
1	name000001	
2	name000002	
...		

MaxR — F (autorelation)

_id	name	attr
1	name000001	
2	name000002	
...		

Figure 5.2: Database Schema for testing

For our tests we will use a simple database schema. It consist of a single relation R between two entities E and F.

The relation has no constraint. We used an (n,m) relation to have more flexibility, but in the test results discussions we will explain any differences or consequences introduced by more restrictive integrity constraints: (1,1) and (1,n) relations.

The entities have a numeric key, a fixed name created with the pattern 'nameXXXXXX' where XXXXXX is the key with left zero-padding and an attribute 'attr' that will be initialized with a string of *Esize* characters, to test with different entity size. The parameters of this schema are:

Rcount used to establish the maximum #R of test

Esize used to change size(E)

Rounds used to set the number of test repetitions

CoarseLog used to set the type of logs (coarse or fine)

MongoDB Schema Mapping

In MongoDB (cf. Section 2.4.2) we have stored the entities and the relations objects in documents. We have created three Document Collections: E, F and R. For simplicity we have used the string type for all the attributes and a flatten structure of document. We store each column of the Common Schema blocks (cf. 4.1.3) as an attribute of MongoDB document. An example of tested schema is:

```
E: [{
  _id: ’1’,
  name: ’name000001’,
  attr: ’’
},{
  _id: ’2’,
  name: ’name000002’,
  attr: ’’
}, ... ]

F: [{
  _id: ’1’,
  name: ’name000001’,
  attr: ’’
},{
  _id: ’2’,
  name: ’name000002’,
  attr: ’’
}, ... ]
```

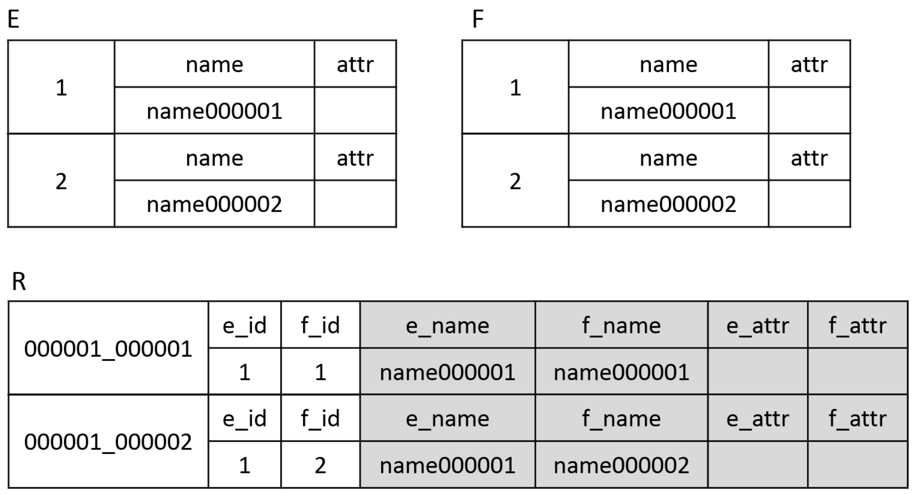
```
R: [{
    _id: ’000001_000001’,
    e_id: ’1’,
    f_id: ’1’,
    e_name: ’name000001’,
    e_attr: ’’,
    f_name: ’name000001’,
    f_attr: ’’
},{
    _id: ’000001_000002’,
    e_id: ’1’,
    f_id: ’2’,
    e_name: ’name000001’,
    e_attr: ’’,
    f_name: ’name000002’,
    f_attr: ’’
}, ... ]
```

This is an example of MaxR schema realization. For MinR model the schema is the same but without the Denormalizations in R documents (*e_name*, *e_attr*, *f_name*, *f_attr*).

MongoDB offers an efficient secondary index implementation (BTree). We used it for indexing the relations documents with entity keys. This means that the $ekeys + findByKeys$ operations will be carried out directly from the database (cf. Section 6.5.1)

Cassandra Schema Mapping

Cassandra is a Column-based database (cf. Section 2.4.3). The entities and relations are stored in different column families (E, F and R). As in the example in Figure 5.3 we store each column of the Common Schema blocks (cf. 4.1.3) as a different column of Cassandra related Column Family. The highlighted columns in figure are the optional denormalization that we found only in MaxR.



E

1	name	attr
	name000001	
2	name	attr
	name000002	

F

1	name	attr
	name000001	
2	name	attr
	name000002	

R

000001_000001	e_id	f_id	e_name	f_name	e_attr	f_attr
	1	1	name000001	name000001		
000001_000002	e_id	f_id	e_name	f_name	e_attr	f_attr
	1	2	name000001	name000002		

Figure 5.3: Cassandra Schema Mapping

Cassandra provides a secondary indexing feature, but the operations that we can do on it are limited only at the equalities queries. Range queries are not supported because in the current implementation, they are using a not ordered hash indexing.

In addition, the DataStax documentation reports that this simple feature works well only with small number of unique values to index, i.e. for classification [25].

Since this limits too much the operations on the relations, we have decided to implement a custom secondary index with the One-to-Several Column

Families Indexes pattern recommended in DataStax documentation [26].

We report here the key points of this technique:

- One indexed value matches several row keys
- Each index is a single row with one super column per indexed value
- Index row key is the name of the index
- Index super column names are the values being indexed
- Index sub-column names are the row keys being indexed
- There is no sub-column value

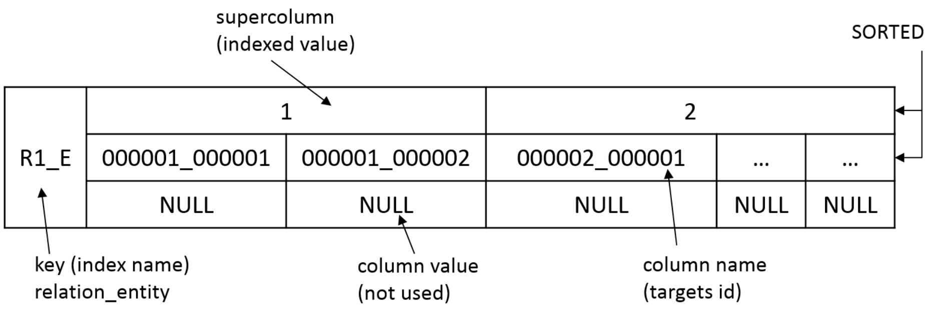
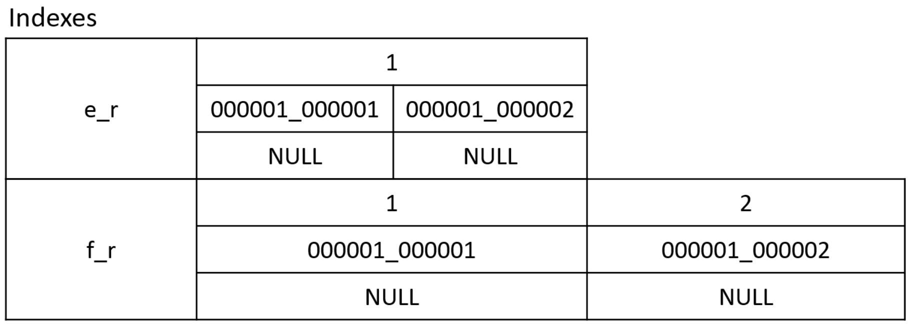


Figure 5.4: One-to-Several Column Families Indexes with Supercolumn



Indexes

e_r	1	
	000001_000001	000001_000002
	NULL	NULL
f_r	1	2
	000001_000001	000001_000002
	NULL	NULL

Figure 5.5: Cassandra Schema Mapping: Index

A general example of One-to-Several Column Families Index is shown in Figure 5.4 and the specific implementation for Database Schema used for Testing is shown in Figure 5.3.

This and other techniques are explained well also by Ed Anuff in his blog pages [9, 10].

5.4 Preliminary Tests

In this section we present the results of preliminary tests. This tests as we said in Section 5.1 are needed to confirm the theoretical assumptions that we have exposed in Section 4.7.

The results on MongoDB implementation are shown in Tables 5.2 and 5.1.

Both tables report the time costs of each CRUD operations in case of a sparse and a full relations. With "sparse" we mean that not all entity e are in relation with one or more entity f. Instead with "full" we mean that all entity e are in relation with at least one entity f.

CRUD	**Sparse relation**		**Full relation**	
$size(E) \sim 50\,byte$	$\#E = 1000$, $\#R = 100$		$\#E = 10$, $\#R = 10000$	
	MinR[ms]	**MaxR[ms]**	**MinR[ms]**	**MaxR[ms]**
createE	0,0670445	0,0620414	0,10061	0,60155
createR	0,0024016	0,00290194	0,29745565	0,25657643
findE	0,0480327	0,0490334	0,49964	0,39941
findR	0,1250821	0,0410271	0,20013	0,1001
selectE	1,9262514	1,9227638	0,6004	0,55125
selectR	**2,839279**	**0,1521017**	**46,90421**	**60,25334**
updateE	**36,6707633**	**37,4844611**	**33,57822**	**75,19826**
updateR	0,1512948	0,1541704	1,18493	0,39967
deleteE	0,0540424	0,0520344	0,50042	0,10006
deleteR	0,1581905	0,1581627	16,99203	18,88321

Table 5.1: Preliminary tests on MongoDB: CRUD operations, $size(E) \sim 50\,byte$

The only difference between two tables is the $size(E)$ parameter. With big entities the differences between two models grows.

Testing on Cassandra give similar results. Cassandra results are shown in Tables 5.4 and 5.3.

According to the assumptions, the operators that really differ largely when we change logical model are `selectR` and `updateE`.

Also, we can see the difference of the MinR multiple requests overhead in `findR` and the penalty of MaxR multi-update in `createR` when the entity size and relation cardinality are big. Nevertheless these are less important than the firsts two because the costs are however very small.

The detailed reports of these tests are shown in the next sections.

CRUD $size(E) \sim 10\,Kb$	**Sparse relation** $\#E = 1000,\ \#R = 100$ **MinR[ms]**	**MaxR[ms]**	**Full relation** $\#E = 10,\ \#R = 10000$ **MinR[ms]**	**MaxR[ms]**
createE	0,1150769	0,1200802	0,10007	0,10007
createR	0,00250164	0,00310265	0,29226131	0,45879466
findE	0,0710472	0,0720421	0,40027	0,4001
findR	0,146468	0,0410274	0,20013	0,10004
selectE	33,2085865	33,0015712	0,90066	0,70093
selectR	**33,3356201**	**0,174501**	**50,73389**	**524,22053**
updateE	**35,5814197**	**35,8271221**	**35,74434**	**317,81568**
updateR	0,189303	0,1701231	0,8006	0,70047
deleteE	0,0560438	0,0520357	0,30066	0,20013
deleteR	0,1644188	0,160107	19,74068	25,71719

Table 5.2: Preliminary tests on MongoDB: CRUD operations, $size(E) \sim 10.000\,byte$

CRUD $size(E) \sim 50\,byte$	**Sparse relation** $\#E = 1000,\ \#R = 100$ **MinR[ms]**	**MaxR[ms]**	**Full relation** $\#E = 10,\ \#R = 10000$ **MinR[ms]**	**MaxR[ms]**
createE	0,681453	0,510341	0,30005	0,29889
createR	0,2467189	0,199791	0,93145838	1,00767964
findE	0,450828	0,301788	0,80054	0,70026
findR	0,938364	0,270065	0,50027	0,2002
selectE	9,991591	8,338505	1,99307	1,60082
selectR	**18,639692**	**22,884967**	**4188,40966**	**11268,71447**
updateE	**0,961803**	**1,408406**	**2,24437**	**95,53648**
updateR	0,341824	0,281162	0,50036	0,70768
deleteE	0,78143	0,540361	0,8552	0,50085
deleteR	1,111182	0,862785	104,91701	97,59858

Table 5.3: Preliminary tests on Cassandra: CRUD operations, $size(E) \sim 50\,byte$

CRUD $size(E) \sim 5\,Kb$	**Sparse relation** $\#E = 100,\ \#R = 100$		**Full relation** $\#E = 10,\ \#R = 1000$	
	MinR[ms]	**MaxR[ms]**	**MinR[ms]**	**MaxR[ms]**
createE	0,320208	0,280181	0,3002	0,30026
createR	0,0980648	0,1100777	0,9512419	1,11054101
findE	0,220147	0,24937	0,80051	0,7002
findR	0,42028	0,170111	0,6004	0,40087
selectE	13,446265	13,033331	2,66169	1,90043
selectR	**25,762906**	**31,701302**	**4265,93176**	**17384,42828**
updateE	**1,0071**	**1,014188**	**2,5244**	**186,77612**
updateR	0,250161	0,18012	0,60037	0,7005
deleteE	0,38026	0,31021	0,80057	0,40026
deleteR	0,801854	0,550373	101,94431	230,04331

Table 5.4: Preliminary tests on Cassandra: CRUD operations, $size(E) \sim 5.000\,byte$

5.5 Detailed Tests Presentation Schema

To expose each detailed test we will follow this schema:

- In **Test Objectives** we will present the motivation of test.
- In **Test Scenario** we will present how the test will be actuated with figures, an algorithm of test and other information organized in a schematic form:
  - Setup: test steps to fill the database, following the schema presented in Section 5.3.2
  - Test: test steps to carry out the measurements
  - Test Parameters: a summary of what parameters we can configure for the test
  - Schema static properties: the set of properties of the schema that remain fixed for the entire test
  - Schema variable properties: the set of properties of the schema that change during the test execution
- In **Test Hypothesis** we will describe what we expect to find from the execution of test

- In **Test Results** we will present the results of the test summarized in tables and graphs. We will present only a portion of them, some combination of parameters that show all the behaviors that we want to describe. In graphs we report for MinR and MaxR cases the mean value of measures with the indication of standard deviation ($MinR-\sigma$, $MinR+\sigma$, $MaxR-\sigma$, $MaxR+\sigma$) that is the square root of calculated variance (cf. Section 5.2), to give an idea of performance variations.

- In **Test Discussion** we will expose the conclusions we have reached.

5.6 Testing Select Relation Operator (`selectR`)

As we have explained in Section 4.6.1 the `selectR` operator consists in reading of multiple tuples of relation and connected entities.

Theoretical investigations on the models led us to think that the first comparison to be done would be the one on the difference in cardinality between the primary entity and the relation, because the scan in MinR is done on the entity set and in MaxR is directly done on the R tuples.

The costs (simplified, as hypothesis, in a binary relation) of the algorithms for the two models are these below. [1]

MinR:

```
          scan(E) + proc_data(kE*size(E)) +
    ekeys(R, kE) + proc_data(kR*3*size(ID))) +
          kF*(find(F) + proc_data(size(F)))
```

MaxR:

```
    scan(R) + proc_data(kR*(size(E) + size(F)))
```

Analyzing these costs we can see that the heavier operations are the entity scan in MinR and the relation scan in MaxR (this is true for $kR \ll \#R$, that is the common case, instead if $kR \sim \#R$ the `ekeys` cost becomes comparable or also greater than the scan cost). The scan costs is proportional to the cardinality of the scanned set and then the first thing to check is the difference between #E and #R (see Test 1: #E vs #R).

The other big difference between the two algorithms is in the amount of data to be processed, prepared and transferred to central memory. In MinR

[1]The notations are shown in Section 4.2.2

are processed exactly the data we need, then kE entities E, kF entities F, kR ID pairs of relation. In contrast, being MaxR entities replicated on each tuple in R will be processed much more data, kR tuples with size equal to the sum of the size of the entities. In addition, when kR grows, the ekeys component in MinR takes importance, too. In the second test (see Test 2: $E_{replica}$ Impact), we investigate the impact of these two aspects.

The targets of these tests are two, choosing the best model for read operations on a relation and the identification of possible sharding point.

5.6.1 Test 1: #E vs #R

Test Objectives

With this test we would evaluate, in "Select Relation" operator, how much the `scan` cost changes in two algorithms when #R grows, with fixed #E.

To isolate the contribution of the scan from the total cost of the algorithm we have to cancel the costs of other components. This can be achieved by choosing a predicate on the entity E that does not return any results ($kE = 0$). If we have no E, we also do not search for associated relations, so the `ekeys` is not called. Of course also the data to be processed are zero.

Test Scenario

Setup:

1. Create *Rcount* entities in E
2. Create $2 * Rcount$ entities in F

Test:

1. Create new relation in R until $\#R = 2 * \#E$ is reached and measure Select Relation time for several value of #R.

Figures 5.6, 5.7 show the Test step for the MinR and MaxR models starting from a completed Setup. Algorithm 5.1 reports Setup and Test steps in pseudocode.

Test parameters:

Ecount:	fixed number of E entities
Esize:	fixed size of E entities
Rounds:	fixed number of repeats of the operations
steps:	list of #R values in which to take a log

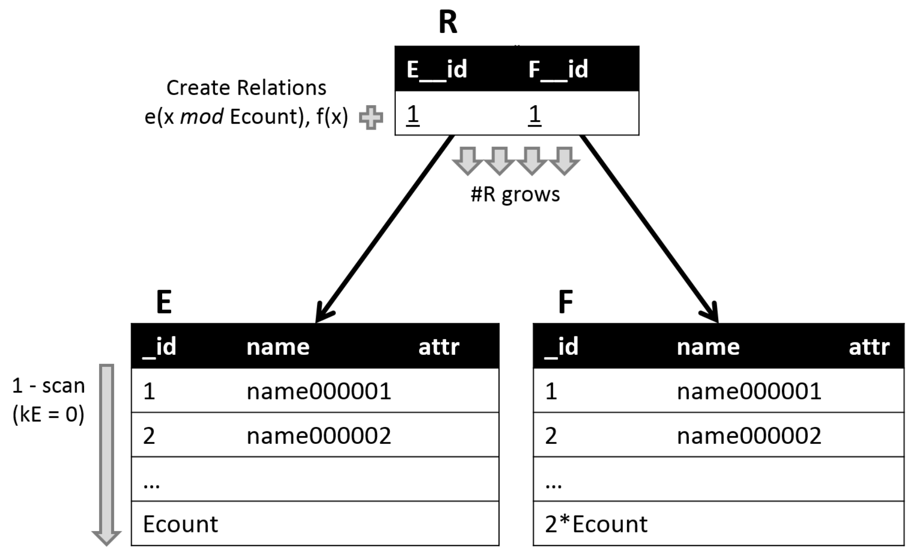


Figure 5.6: MinR, Select Relation Test 1: #E vs #R

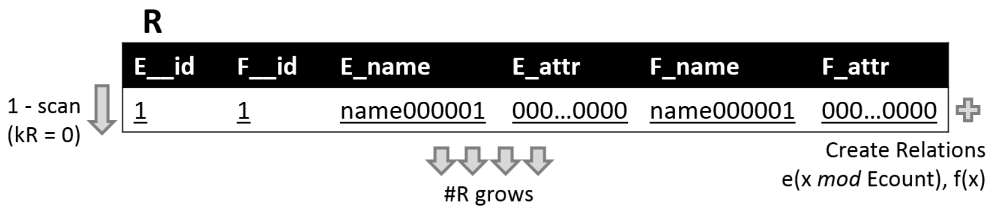

R
E__id F__id E_name E_attr F_name F_attr
1 1 name000001 000...0000 name000001 000...0000
1 - scan
(kR = 0)
#R grows
Create Relations
e(x mod Ecount), f(x)

Figure 5.7: MaxR, Select Relation Test 1: #E vs #R

Schema static properties:

$$\begin{cases} kE &= 0 \\ kR &= 0 \\ kF &= 0 \\ \#E &= Ecount \\ \#F &= 2 * Ecount \end{cases}$$

Schema variable properties:

$$\#R = from\ 1\ to\ 2 * \#E$$

Algorithm 5.1 Test 1: #E vs #R

```
procedure TEST_READR_MULTI_1
    // Setup
    Create $Ecount$ entity in $E$
    Create $2 * Ecount$ entities in $F$
    // Test
    for $i$ from 0 to $(2 * \#E - 1)$ do
        Create new relation in $R$ between $e_{i\ mod\ \#E}$ and $f_i$
        if log step is reached then
            Log selectR (for $kE = 0$, $kR = 0$)                ▷ LOG
        end if
    end for
end procedure
```

Test Hypothesis

What we expect varying #R maintaining the same value of #E is that in the MinR model nothing change, because the scan is on the entity table then on the fixed #E tuple. In MinR instead we expect a proportional linear increment to the #R value. We also expect that the others costs are irrelevant with respect to scan cost when the numbers are high enough.

The scan on E_{MinR} costs less with respect the one on R_{MaxR}, because of the bigger size of R_{MaxR} tuples that contain the copies of all the entities linked in relation and that correspond to more data to read from memory.

$$size(E_{MinR}) = size(E)$$
$$size(R_{MaxR}) = size(E) + size(F)$$

Having said this, we can assume that for $\#R << \#E$ use the MaxR model costs less but, when the #R grows, this cost increases until it reaches the constant cost of MinR model. The overtaking is presumably when the $\#R$ is close but slightly smaller than $\#E$, because of the size questions seen before.

An hypothetical chart might look like the one in Figure 5.8.

From this general behavior we can make some observations based on the specific type of relationship. A (1,1) relation will have the same cardinality as the linked entities, $\#R = \#E = \#F$. If the relation is non-mandatory [2]

[2]Non-mandatory relation means that can exist some entities that are not in relation with any others

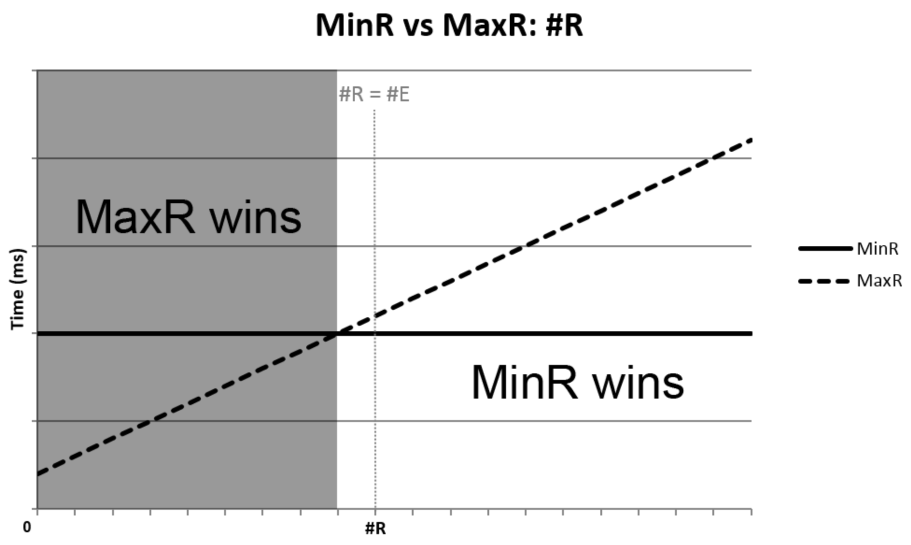


Figure 5.8: Hypothetical chart, Select Relation Test 1: #E vs #R

this constraint is relaxed in $\#R \leq min(\#E, \#F)$.

And yet, a (1,n) relation will have at maximum the same cardinality of the most populous linked entity, $\#R \leqslant max(\#E, \#F)$.

$$\begin{aligned} R(1,1): &\quad \#R &\leqslant min(\#E,\#F) \\ R(1,n): &\quad \#R &\leqslant max(\#E,\#F) \\ R(n,m): &\quad \#R &is\,free \end{aligned}$$

From these considerations we can draw the conclusion that for non-mandatory (1, 1) and (1, n) relations with $\#E \geqslant \#F$, MaxR model is the best choice, because as we can see in Figure 5.9 we are in the "MaxR usually wins" zone. Nevertheless, when $\#R \sim \#E$ MinR wins, so if the relation is mandatory the right choice is the MinR model.

Test Results

We have run this test on the two our implementations with several combinations of parameters.

Rcount = { 10, 100, 500, 1000, 2000, 3500, 5000, 6500, 8000, 9000, 10000, 11000, 12000, 20000, 30000, 40000, 50000, 100000 }

Esize = { 0, 10, 100, 1000, 10000, 100000 }

Rounds depends on Rcount parameters, from 10 to 1000 rounds

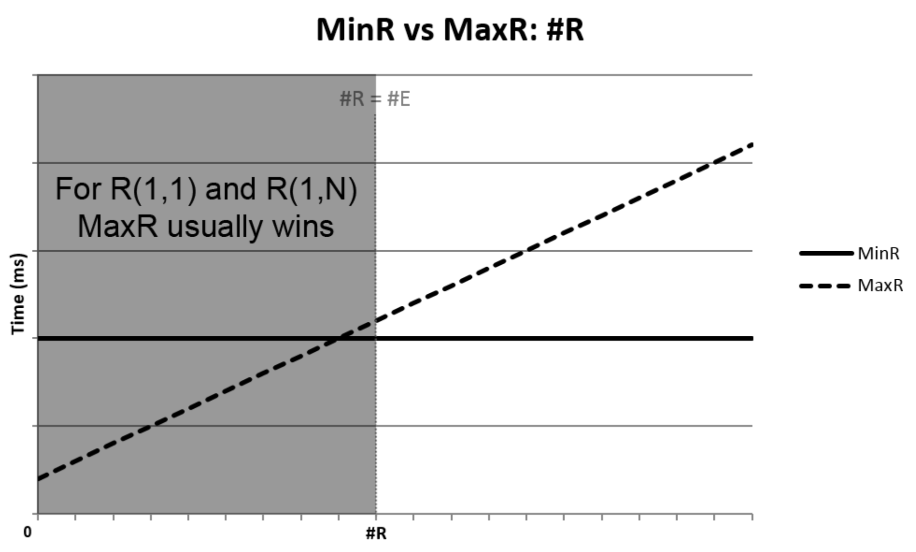


Figure 5.9: Hypothetical chart, R(1,1) and R(1,n) in Select Relation Test 1: #E vs #R

It was not possible to test all combinations on both databases. On Cassandra we have reached the limit of the central memory and some timeouts on connections with high value in Rcount or Esize parameters (Rcount ¿ 30000 with Esize = 10000). It could be a non-optimal configuration problem or a problem of drivers used (cf. 5.3).

We have reported here only some results that show all the necessary information to our test.

In Tables 5.5, 5.6 and Figures 5.10, 5.11, 5.12, 5.13 we can see MongoDB results for the Rcount values 10000 and 100000 and for two Esize values, 10 and 10000 and in Tables 5.7, 5.8 and Figures 5.14, 5.15, 5.16, 5.17 those of Cassandra for the Rcount values 1000 and 10000 and for two Esize values, 10 and 10000.

The difference in the chosen parameters for testing MongoDB and Cassandra is because of the impossibility with Cassandra to test a relation with 100000 tuples for the memory limits seen before and because MongoDB has very fluctuating behavior with small collections of documents. This could be caused by particular optimization in fetching data from memory and disk by Mongo, that use memory-mapped files (cf. 2.4.2).

The result is that loading 1 or 100 documents from Mongo consumes the same quantity of time. Probably because they are loaded together in the same chunk of memory and the scan loads always the entire chunk. But this is only our conjecture.

	#R = 10000 **size(E) = 10**			**#R = 10000** **size(E) = 10000**	
#E	**MinR[ms]**	**MaxR[ms]**	**#E**	**MinR[ms]**	**MaxR[ms]**
1000	3,122182	**0,480326**	1000	4,062694	**0,480338**
2000	3,202134	**0,760506**	2000	3,482322	**0,840566**
3000	3,06204	**1,000884**	3000	3,622414	**1,260858**
4000	3,002008	**1,32088**	4000	3,66244	**1,62131**
5000	2,901934	**1,561022**	5000	3,602408	**1,96153**
6000	2,982114	**1,921276**	6000	3,562374	**2,30154**
7000	3,102066	**2,121414**	7000	3,442292	**2,72182**
8000	3,082066	**2,44185**	8000	3,622412	**3,222242**
9000	2,921942	**2,801958**	9000	**3,482314**	3,622516
10000	**2,981994**	3,062264	10000	**3,802534**	4,082944
11000	**3,001996**	3,3022	11000	**3,502328**	4,543028
12000	**3,002008**	3,622414	12000	**3,442282**	5,203462
13000	**3,022024**	3,962646	13000	**3,522336**	5,563816
14000	**3,06204**	4,2028	14000	**3,502328**	5,704018
15000	**3,06204**	4,462968	15000	**3,68246**	6,184236
16000	**3,001886**	4,763282	16000	**3,542348**	7,084726
17000	**2,941852**	5,023348	17000	**3,422274**	7,204802
18000	**3,022094**	5,263508	18000	**3,6224**	7,324882
19000	**3,082054**	5,60383	19000	**3,48232**	8,185546
20000	**3,00212**	5,88392	20000	**3,522334**	8,545688

Table 5.5: selectR Test 1: **Mongo** #R = 10000

	#R = 100000 **size(E) = 10**			**#R = 100000** **size(E) = 10000**	
#E	**MinR[ms]**	**MaxR[ms]**	**#E**	**MinR[ms]**	**MaxR[ms]**
10000	25,91721	**3,10207**	10000	54,93658	**5,60371**
20000	25,9173	**5,90397**	20000	45,13059	**9,60692**
30000	26,21747	**8,80517**	30000	45,73039	**14,21064**
40000	25,71713	**11,70782**	40000	45,73047	**19,31284**
50000	25,81717	**14,8099**	50000	45,03001	**28,51903**
60000	25,81721	**17,61173**	60000	45,53025	**27,11862**
70000	25,91728	**20,31408**	70000	49,13277	**32,12141**
80000	25,8172	**23,31554**	80000	45,12953	**36,62441**
90000	**26,01734**	26,2175	90000	45,23017	**40,32569**
100000	**25,91724**	29,21948	100000	48,13204	**44,02933**
110000	**25,91781**	31,82233	110000	**44,52961**	48,93312
120000	**26,01733**	34,82209	120000	**53,53567**	58,23881
130000	**26,01737**	37,62443	130000	**53,43561**	59,43951
140000	**26,41766**	40,62704	140000	**44,72978**	71,04734
150000	**25,81775**	43,52964	150000	**45,43018**	76,14966
160000	**26,01727**	46,13074	160000	**44,82981**	74,14869
170000	**26,01734**	49,3329	170000	**44,32954**	78,75302
180000	**25,81717**	51,83454	180000	**44,52964**	94,46244
190000	**26,21796**	54,53634	190000	**53,93597**	101,86782
200000	**25,91724**	57,53834	200000	**53,53567**	111,47427

Table 5.6: selectR Test 1: **Mongo** #R = 100000

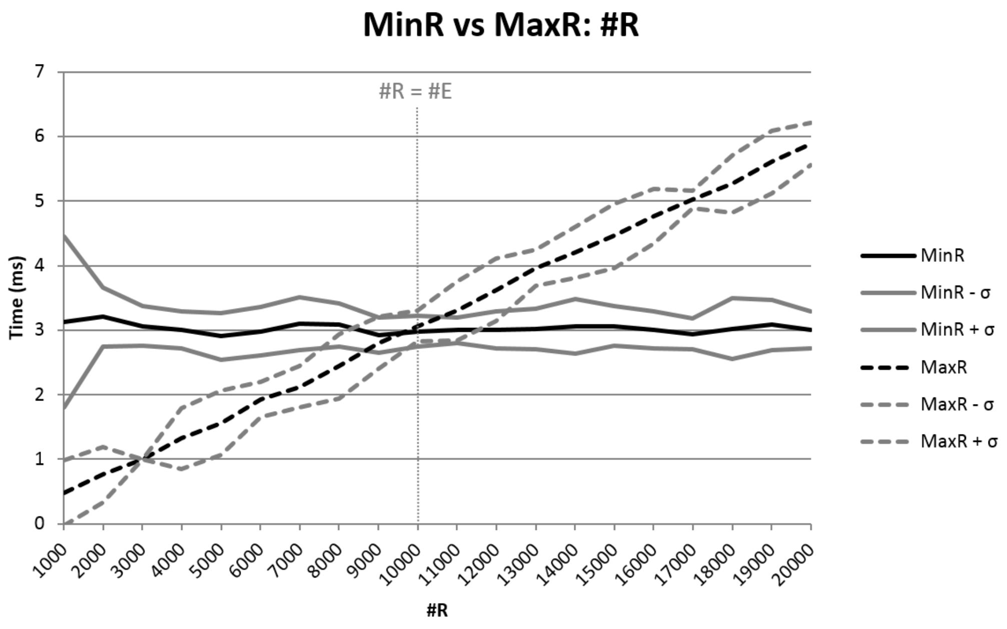


Figure 5.10: selectR Test 1: **Mongo** #R = 10000, size(E) = 10

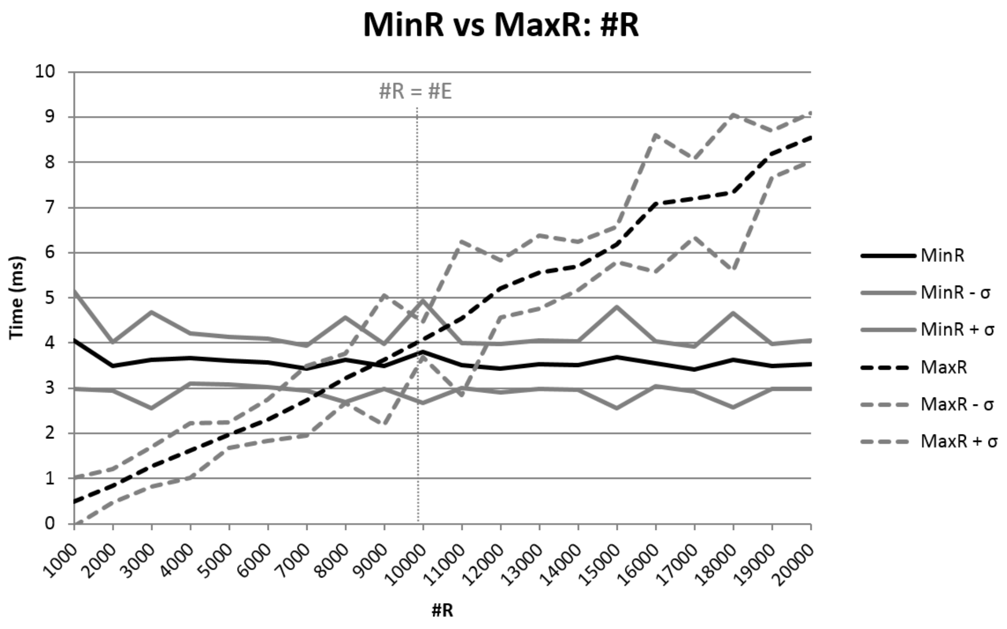


Figure 5.11: selectR Test 1: **Mongo** #R = 10000, size(E) = 10000

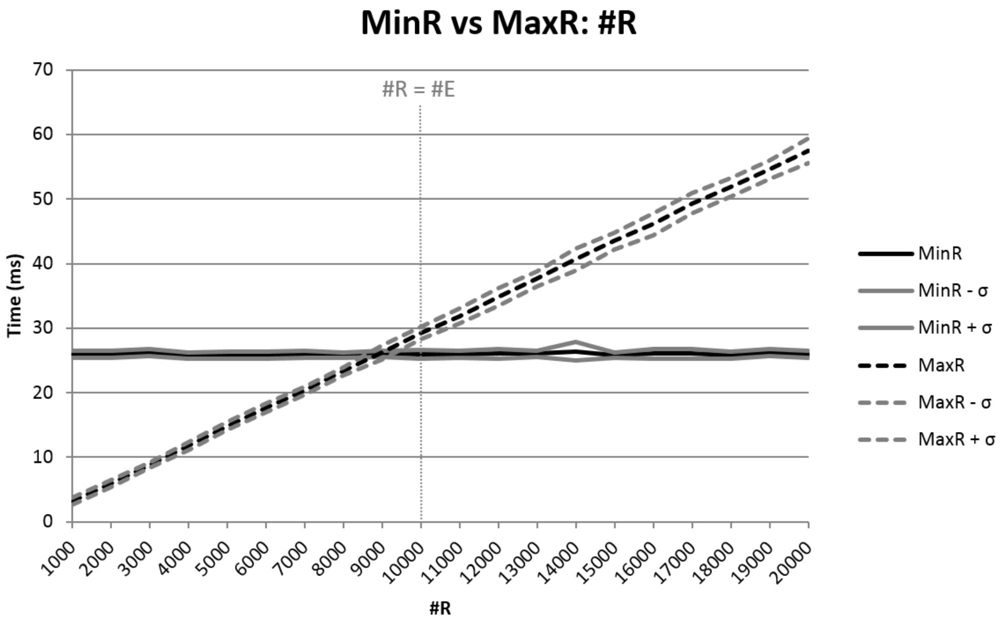


Figure 5.12: selectR Test 1: **Mongo** #R = 100000, size(E) = 10

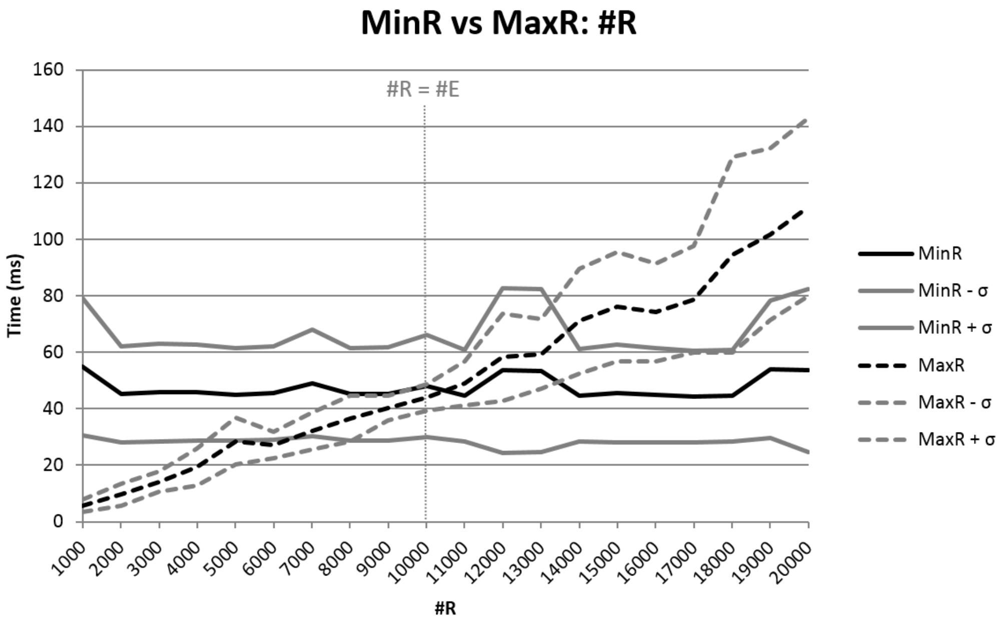


Figure 5.13: selectR Test 1: **Mongo** #R = 100000, size(E) = 10000

	#R = 1000			**#R = 1000**	
	size(E) = 10			**size(E) = 10000**	
#E	**MinR[ms]**	**MaxR[ms]**	**#E**	**MinR[ms]**	**MaxR[ms]**
100	32,12884	**5,45171**	100	97,58747	**12,10692**
200	31,67331	**11,5607**	200	97,89473	**23,3505**
300	30,51116	**16,82002**	300	96,08872	**39,62885**
400	30,38954	**22,62087**	400	97,44838	**47,83742**
500	32,25328	**28,7431**	500	94,12754	**62,51619**
600	**30,44418**	34,70468	600	96,29469	**79,58306**
700	**29,51908**	40,1489	700	94,96362	**94,36719**
800	**29,92373**	47,71436	800	**96,46734**	102,86674
900	**30,05114**	52,75347	900	**94,6196**	114,36232
1000	**30,27979**	60,21925	1000	**95,83696**	129,68995
1100	**29,31202**	66,94703	1100	**95,41987**	151,60817
1200	**29,79531**	72,65485	1200	**94,82551**	159,15263
1300	**29,67889**	78,16307	1300	**96,86264**	172,61836
1400	**29,49624**	87,59702	1400	**97,04485**	187,21899
1500	**30,10178**	93,49912	1500	**99,34833**	205,07223
1600	**29,82127**	101,22249	1600	**95,66641**	214,12544
1700	**30,44891**	108,40819	1700	**99,18801**	228,88999
1800	**30,17039**	115,37136	1800	**101,97237**	242,73074
1900	**30,64855**	123,06742	1900	**94,98239**	255,86764
2000	**30,13804**	131,67863	2000	**94,62157**	271,19528

Table 5.7: selectR Test 1: **Cassandra** #R = 1000

	#R = 10000 **size(E) = 10**			**#R = 10000** **size(E) = 10000**	
#E	**MinR[ms]**	**MaxR[ms]**	**#E**	**MinR[ms]**	**MaxR[ms]**
1000	697,00292	**61,13618**	1000	1478,2919	**128,23682**
2000	696,01348	**130,53162**	2000	1486,2511	**271,5584**
3000	700,354	**214,39338**	3000	1476,4686	**425,30868**
4000	700,21566	**311,3088**	4000	1475,273	**597,13788**
5000	698,07754	**412,68894**	5000	1470,4363	**769,74576**
6000	704,19404	**535,0684**	6000	1476,1747	**965,59778**
7000	703,1904	**657,83166**	7000	1477,8797	**1188,7811**
8000	**699,91642**	796,63704	8000	1530,6251	**1375,467**
9000	**705,41336**	937,93334	9000	**1488,5104**	1621,3456
10000	**701,20962**	1095,0463	10000	**1505,0875**	1842,8407
11000	**700,43148**	1240,4725	11000	**1487,22**	2077,8801
12000	**704,2145**	1428,0467	12000	**1489,0797**	2333,9451
13000	**693,37576**	1589,6737	13000	**1473,9916**	2654,4935
14000	**694,54654**	1805,4059	14000	**1518,8027**	2887,6091
15000	**698,55318**	1966,0778	15000	**1494,8272**	3225,0747
16000	**699,71012**	2155,5133	16000	**1490,8105**	3536,1286
17000	**769,39978**	2395,2753	17000	**1480,7146**	3877,9275
18000	**761,64838**	2624,4864	18000	**1490,9656**	4311,7697
19000	**749,7197**	2955,2693	19000	**1566,7167**	4567,773
20000	**745,63166**	3176,3304	20000	**1533,2202**	5185,6208

Table 5.8: selectR Test 1: **Cassandra** #R = 10000

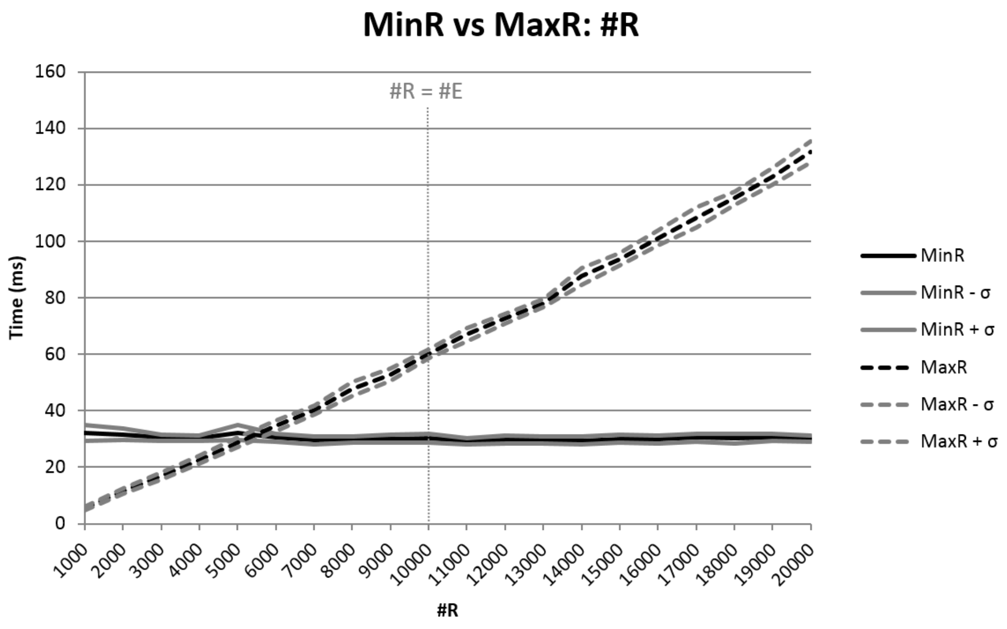


Figure 5.14: selectR Test 1: **Cassandra** #R = 1000, size(E) = 10

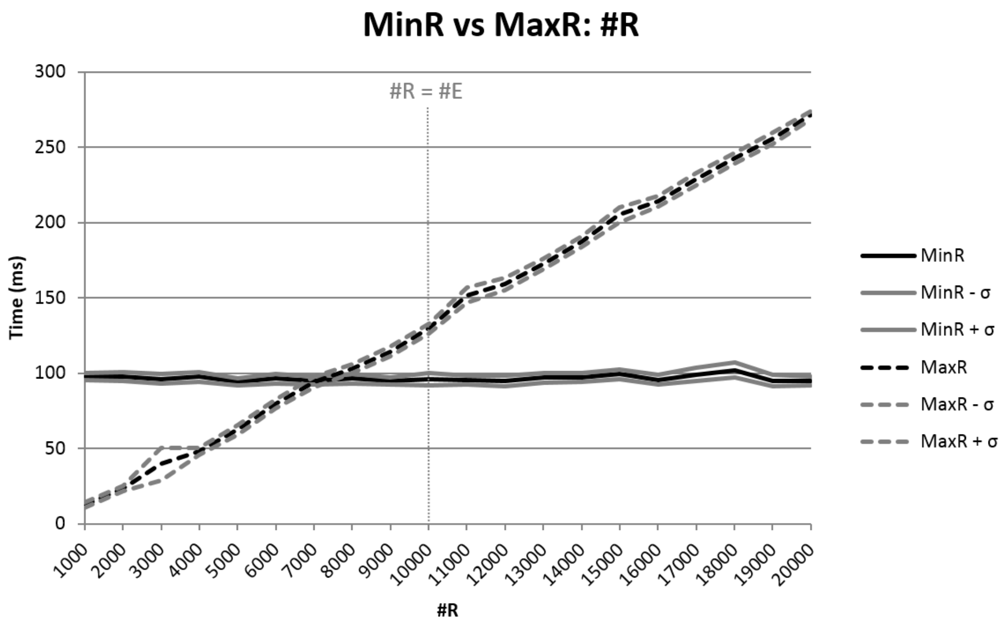


Figure 5.15: selectR Test 1: **Cassandra** #R = 1000, size(E) = 10000

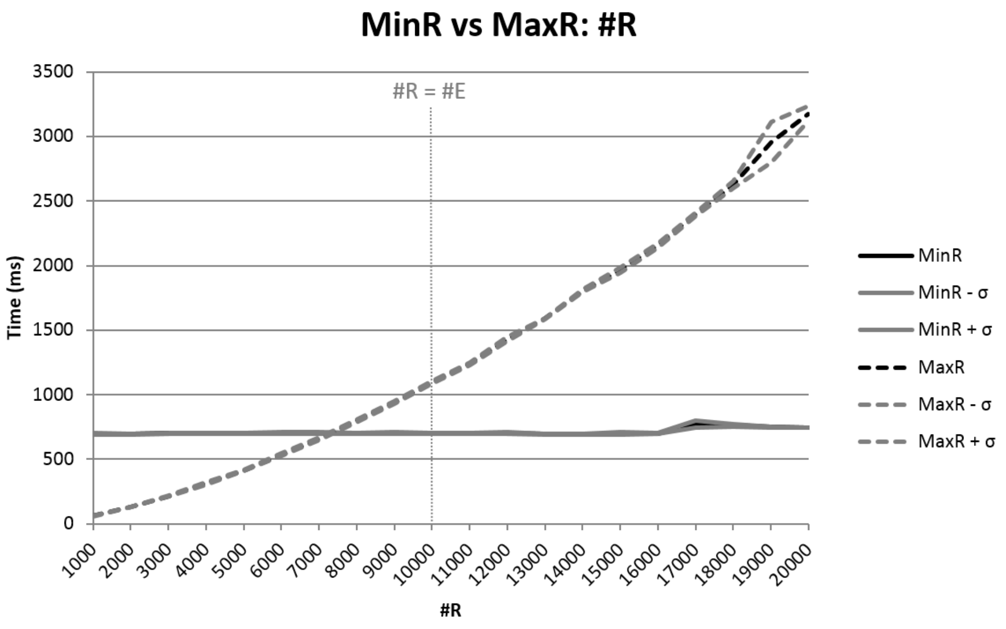


Figure 5.16: selectR Test 1: **Cassandra** #R = 10000, size(E) = 10

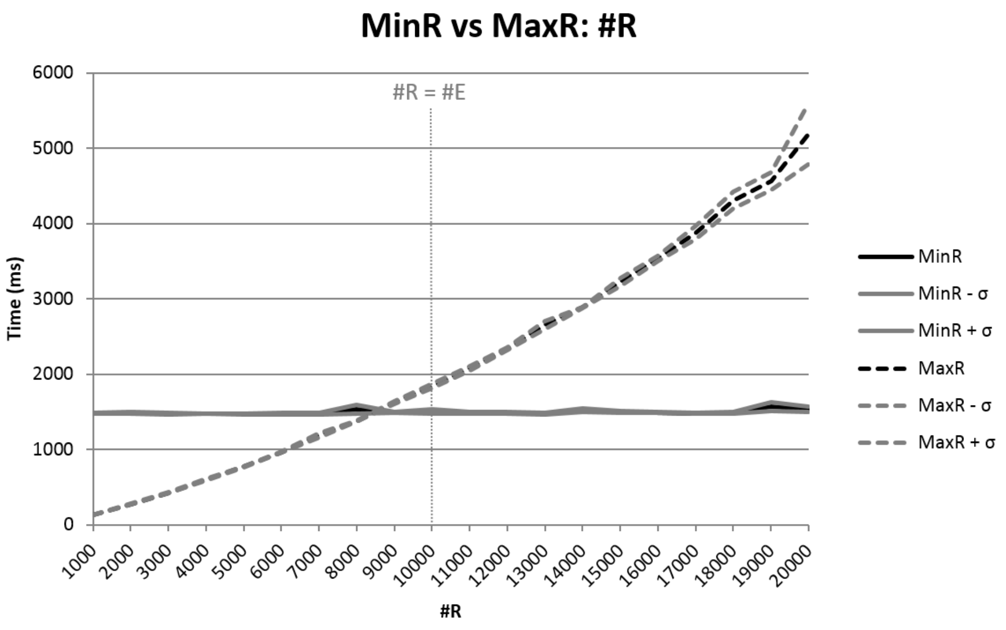


Figure 5.17: selectR Test 1: **Cassandra** #R = 10000, size(E) = 10000

From a comparison of the tables reported for this test (cf. Tables 5.5, 5.6, 5.7, 5.8), we can notice that MongoDB is faster than Cassandra in reading operations on single node with the same amount of data.

We could expect this since Cassandra belong to Column-oriented NoSQL class and, as we have seen in Section 5.1, it needs optimized data access structure to enhance reading performance.

The last information that we can get from the numbers is that when the #R become big enough the trend is not linear, it gets a non-linear component, but the conclusion don't change.

Test Discussion

MinR vs. MaxR From results we can see that our hypothesis were correct. In all tests the MinR chart trends is quite constant, and the MaxR one grows with the increase of #R.

MinR is always cheaper than MaxR after the threshold is reached ($\#R > \#E$).

Sharding To evaluate when to do sharding we can choose a time threshold beyond which we cannot go with the single operation and then with the parameters of our schema (#R e #E) launch the test and take the number of #R that corrisponds to the chosen time threshold. This is the sharding point.

For example suppose to have a database schema the statistically has $\#E = 10000$ with $Esize = 10000$ ($\sim 10KB$) and a (1,1) non optional relation R ($\#R = \#E$) and we want to deploy on Cassandra.

We run the test for $\#E = 10000$ and $Esize = 10000$ on the node and on the graph (suppose that is equal to the one on our test machine 5.17) we read $Time = \sim 1500ms$ the value correspondent to $\#R = 10000$.

But for the purpose of our application the single operation can take $Time = \sim 100ms$ at maximum. Then we repeat tests with the #R parameter divided by increasing number of nodes and for 10 nodes, $\#E = 10000/10 = 1000$ (the other our graph 5.15) we can read for $\#R = 10000/10 = 1000$ the value $Time = \sim 96ms$ for MinR.

In Cassandra we need 10 nodes with sharding point on R of 1000 tuples.

If we repeat the test on mongo what comes out is that one node is enough (the reading of the value from the graph in Figure 5.11 gives us a Time value less than 5ms, counting the variance). No sharding is needed.

5.6.2 Test 2: $E_{replica}$ Impact

Test Objectives

In this second test we try to measure the impact of entity replicas of MaxR (on `proc_data` component) versus the one of the `ekeys` component of MinR in "Select Relation" algorithm when the number of replicas in the MaxR model increases.

To do this we need to increase the number of $E_{replica}$ without changing other parameters.

So we can start with our database schema with one E entity that is in relation with *Rcount* F entities ($\#E = 1$, $\#R = Rcount$, $\#F = Rcount$) and run a query with a variable predicate on F that results in growing kR from 1 to *Rcount* ($F.name \leqslant$ '$nameXXXXXX$', with $XXXXXX$: $000001 \rightarrow Rcount$).

In this test the scan cost of both models is intentionally the same, because of $\#R = \#F$ (the predicate is on F, check the explanation in Section 5.6).

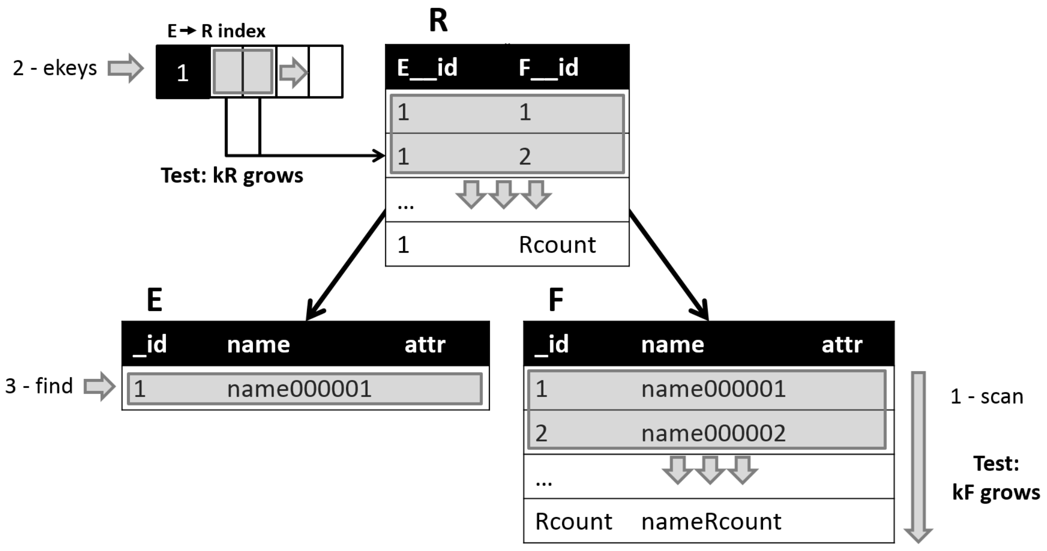


Figure 5.18: MinR, Select Relation Test 2: Replicas impact

Test Scenario

Setup:

1. Create #F entities in F
2. Create 1 entity in E

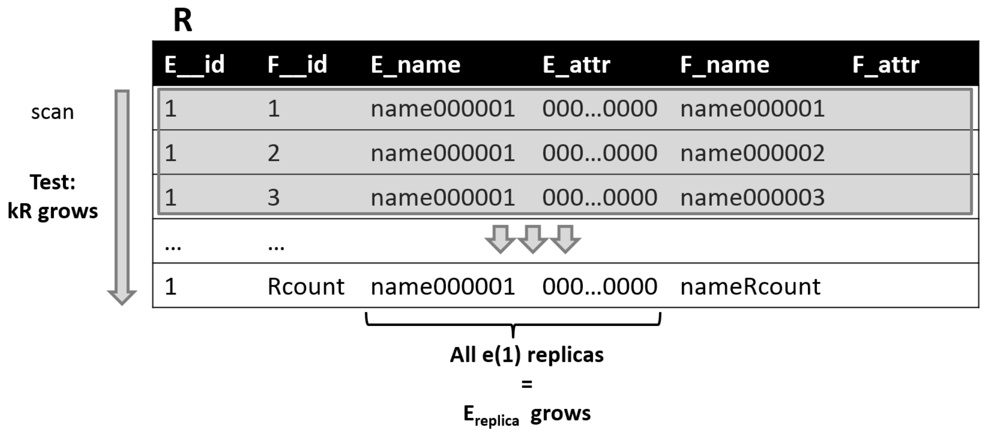

R
E__id
F__id
E_name
E_attr
F_name
F_attr
scan
Test:
kR grows
1 1 name000001 000...0000 name000001
1 2 name000001 000...0000 name000002
1 3 name000001 000...0000 name000003
... ...
1 Rcount name000001 000...0000 nameRcount
All e(1) replicas
=
E_replica grows

Figure 5.19: MaxR, Select Relation Test 2: Replicas impact

3. For each F entity, create a relation in R between this and the only E entity

Test:

1. Log queries with changing $E_{replica}$ from 1 to $Rcount$, like these:

   `SELECT * FROM R WITH F.name <= 'name000001'`
   : $kR = 1$, $E_{replica} = 1$
   `SELECT * FROM R WITH F.name <= 'name000100'`
   : $kR = 100$, $E_{replica} = 100$
   `SELECT * FROM R WITH F.name <= 'nameRcount'`
   : $kR = Rcount$, $E_{replica} = Rcount$

Figures 5.18, 5.19 show the Test step for the MinR and MaxR models starting from a completed Setup. Algorithm 5.2 reports Setup and Test steps in pseudocode.

Test parameters:

Rcount:	fixed number of relation
Esize:	fixed size of entities
Rounds:	fixed number of repeats of the operations
steps:	log steps

Schema static properties:

$$\begin{cases} kE & = 1 \\ \#E & = 1 \\ \#R & = Rcount \\ \#F & = Rcount \end{cases}$$

Schema variable properties:

$$\begin{cases} kF & = from\ 1 to\ Rcount \\ kR & = kF \\ E_{replica} & = kF \end{cases}$$

Algorithm 5.2 Test 2: $E_{replica}$ Impact

```
procedure TEST_READR_MULTI_2
    // Setup
    Create $Rcount$ entities in $F$
    Create 1 entity in $E$
    foreach $f$ in $F$ do
        Create a relation in R between this and the only E entity
    end for
    // Test
    for $i$ from 1 to $Rcount$ do
        if log step is reached then
            Log Select Relation $F.name <=$ '$name\{i\}_{padded}$'        ▷ LOG
        end if
    end for
end procedure
```

Test Hypothesis

What we expect is that with only one copy of E ($E_{replica} = 1$) the amount of data processed and returned are the same in both models, because of absence of multiple replica of the same entity.

When kR increases, by changing query, also the $E_{replica}$ grows, because all the R tuples contains the same E entity (with $e\_id = 1$). So MaxR implementation process $kR * (size(E) + size(F))$ data units, instead of MinR that process only $1 * size(E) + kR * (size(F) + 2 * ID)$ (because of $kF = kR$) with a clear advantage of MinR strategy.

But MinR has the `ekeys` component that increases with kR growing. Furthermore, MinR has the `findByKeys` component, but we assume that this cost is no relevant. This assumption will be confirmed by the measures.

We can suppose that the replicas impact on MaxR raises costs more than the `ekeys` component on MinR when the $Esize$ (and so the global data size of all replicas) is high enough, but not always.

An expected graph could be the one in Figure 5.20 where both MinR and MaxR grow with kR, but with different speed (given by that we said above).

MinR is always slower with small kR because of the more operations that it must do, but when kR grows the scenario could change. In details the query on MinR fetches only one replica of the E entity, so an augment of size(E) means a constant increase in all the graph. Instead, MaxR grows proportionally with the $E_{replica}$ and the $Esize$ parameters.

Finally if the size(E) is small the MaxR will grow more slowly than MinR, else if it is big the MaxR will overcome MinR.

In the Figure 5.21 there is another graph with the expected trends when the $Esize$ parameter changes. This can be useful for understanding what value of $Esize$ balances the replica weight in MaxR and the `ekeys` costs in MinR. Of course this will change according to the number of replicas and then to kR parameter.

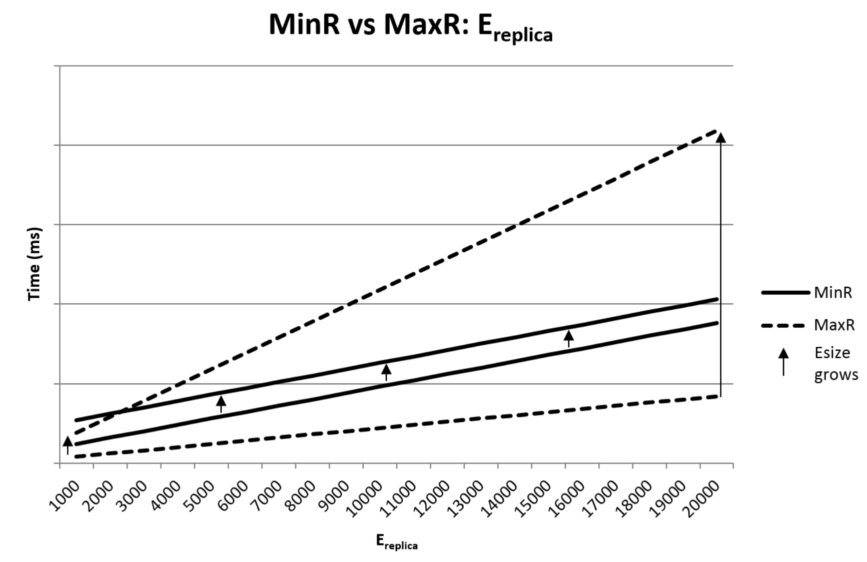


Figure 5.20: Varying $E_{replica}$ in Select Relation Test 2: Replicas Impact

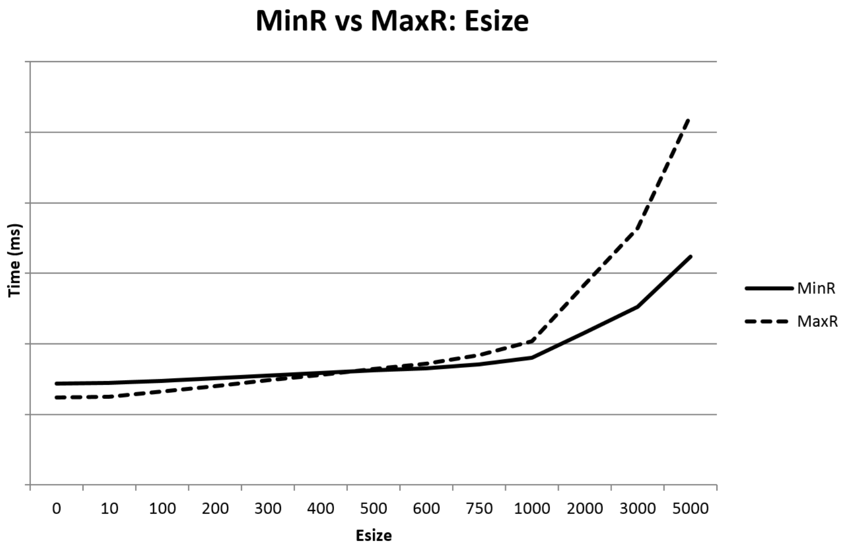


Figure 5.21: Varying *ESize* in Select Relation Test 2: Replicas Impact

Test Results

We have computed test for some combinations of *Rcount* and *Esize*.

Rcount = { 1000, 2000, 3000, 4000, 5000, 10000 }

Esize = { 0, 10, 100, 200, 300, 400, 500, 600, 750, 1000, 2000, 3000, 5000 }

Rounds depends on Rcount parameters, from 10 to 100 rounds

In Tables 5.9, 5.10 and Figures 5.22, 5.23, 5.24, 5.25 we can see MongoDB results for the *Rcount* = 1000 when we change *Esize* or $E_{replica}$ parameters.

In Tables 5.11, 5.12 and Figures 5.26, 5.27, 5.28, 5.29 we can see instead the Cassandra results with the same characteristics.

Test Discussion

The numbers of MongoDB are aligned with our assumptions.

Cassandra has a strange behavior. MaxR has an higher costs than MinR when there are no secondary replicas. In addition, MaxR grows slowly than MinR with the increasing of replica number. This is the opposite to what we expected. The reason could be found in the column-oriented data organization. Column-oriented organization is efficient when only a small subset of columns are needed. To retrieve multiple columns it is necessary to perform

	#R = 1000 *Esize* = **0**			**#R = 1000** *Esize* = **5000**	
$E_{replica}$	**MinR[ms]**	**MaxR[ms]**	$E_{replica}$	**MinR[ms]**	**MaxR[ms]**
1	0,641025	**0,60043**	1	0,70042	**0,60014**
100	1,671785	**0,900615**	100	**1,451**	2,601705
200	3,15262	**1,45094**	200	**2,598825**	4,95331
300	4,26135	**1,951335**	300	**3,251885**	7,50501
400	4,10276	**2,085785**	400	**3,9532**	11,66161
500	4,903815	**2,757605**	500	**4,70319**	15,36627
600	6,203525	**3,00173**	600	**5,67972**	19,3281
700	6,514305	**3,753065**	700	**6,35432**	24,59521
800	7,30983	**4,10274**	800	**7,055285**	27,83905
900	8,156005	**4,75372**	900	**7,883455**	31,5649
1000	8,80588	**5,103395**	1000	**8,905945**	36,98915

Table 5.9: selectR Test 2: **Mongo** #R = 1000, changing $E_{replica}$

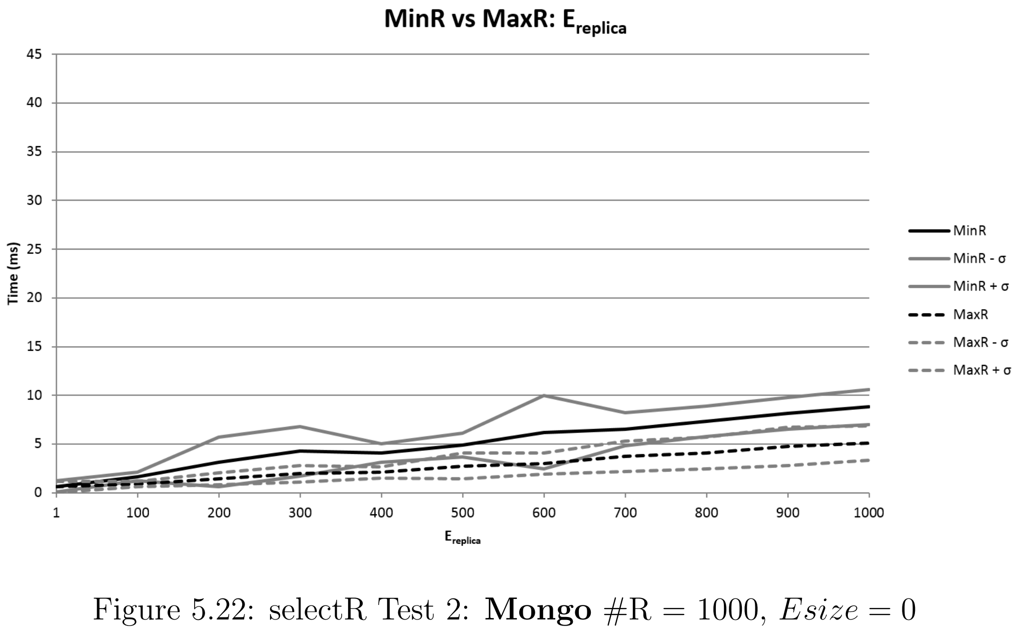


Figure 5.22: selectR Test 2: **Mongo** #R = 1000, $Esize = 0$

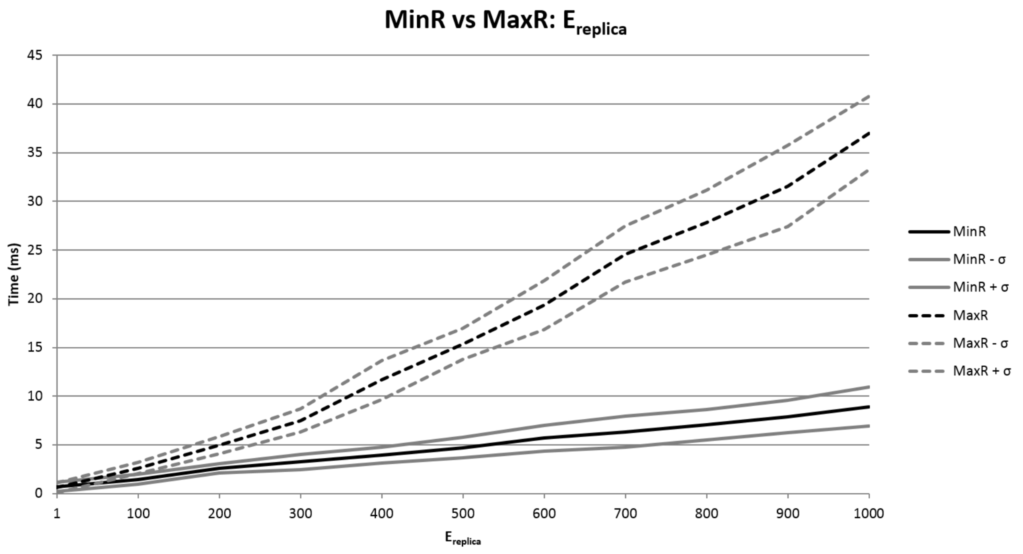


Figure 5.23: selectR Test 2: **Mongo** #R = 1000, $Esize$ = 5000

	#R = 1000 $E_{replica}$ **= 1**			**#R = 1000** $E_{replica}$ **= 1000**	
$Esize$	**MinR[ms]**	**MaxR[ms]**	$Esize$	**MinR[ms]**	**MaxR[ms]**
0	0,641025	**0,60043**	0	8,80588	**5,103395**
10	0,650255	**0,50032**	10	8,75585	**5,25354**
100	0,650255	**0,484035**	100	8,85537	**5,80388**
200	0,65072	**0,600915**	200	8,97755	**6,054605**
300	0,65096	**0,5504**	300	8,760965	**6,50433**
400	0,650465	**0,500365**	400	8,355565	**6,854545**
500	0,650695	**0,550365**	500	8,855915	**7,35491**
600	0,60063	**0,500575**	600	8,96051	**7,70513**
750	0,70077	**0,55038**	750	8,95595	**8,50588**
1000	0,700485	**0,60039**	1000	**8,55567**	9,65669
2000	0,65012	**0,500335**	2000	**8,555685**	16,07271
3000	0,65045	**0,62497**	3000	**8,755285**	22,6943
5000	0,70042	**0,60014**	5000	**8,905945**	36,98915

Table 5.10: selectR Test 2: **Mongo** #R = 1000, changing $Esize$

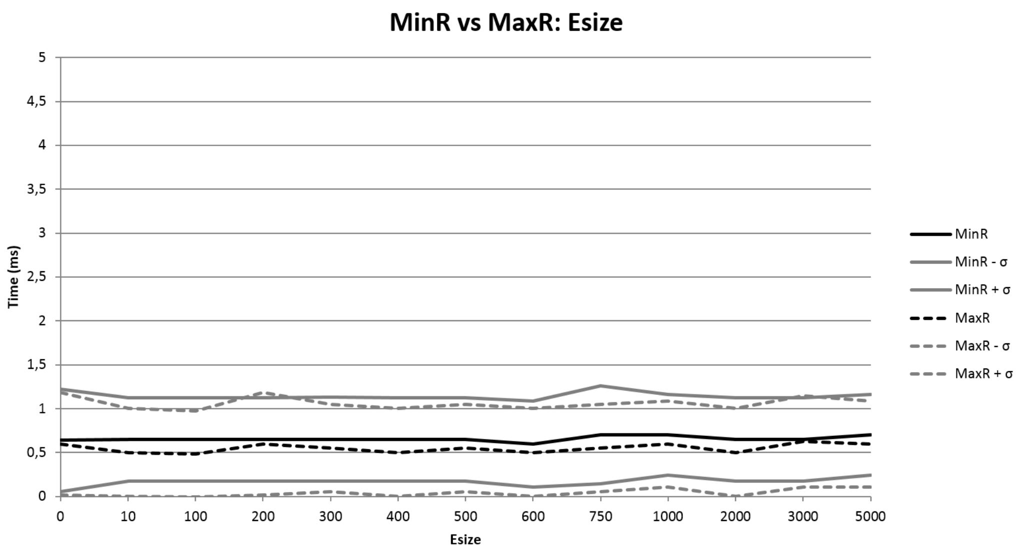


Figure 5.24: selectR Test 2: **Mongo** #R = 1000, $E_{replica} = 1$

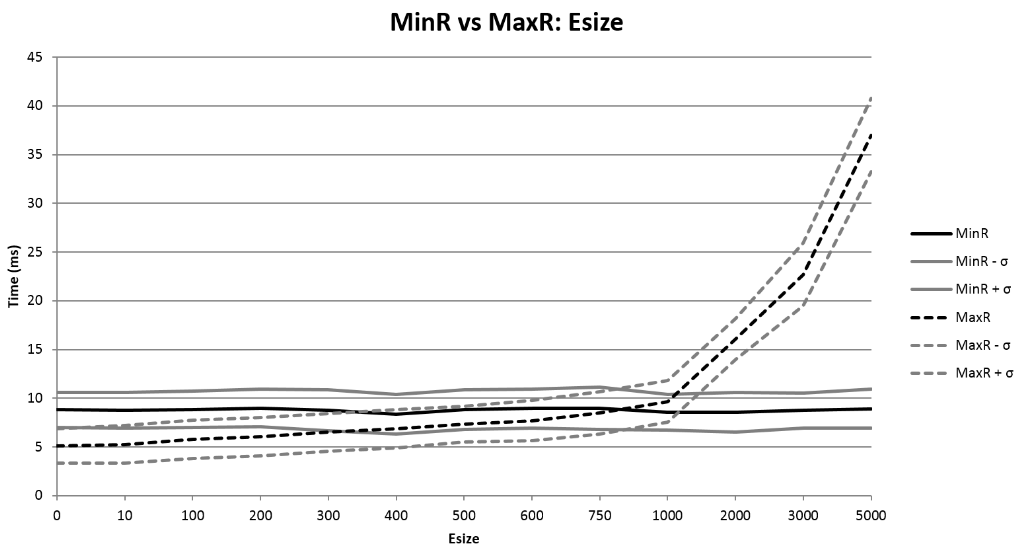


Figure 5.25: selectR Test 2: **Mongo** #R = 1000, $E_{replica} = 1000$

	#R = 1000			**#R = 1000**	
	$Esize$ = **0**			$Esize$ = **5000**	
$E_{replica}$	**MinR[ms]**	**MaxR[ms]**	$E_{replica}$	**MinR[ms]**	**MaxR[ms]**
1	**35,17193**	66,84498	1	**31,61055**	105,8703
100	**36,609**	68,67072	100	**35,38876**	102,0968
200	**42,82912**	69,94852	200	**41,16409**	104,0276
300	**48,06896**	71,57529	300	**46,62017**	106,641
400	**53,24517**	72,35409	400	**53,02661**	109,5197
500	**58,42327**	74,48272	500	**58,82645**	111,4072
600	**65,15845**	74,86541	600	**65,27574**	110,9693
700	**70,78959**	76,78127	700	**71,13109**	114,8372
800	**76,78656**	78,50948	800	**78,75024**	116,62
900	83,22419	**80,14967**	900	**84,31552**	117,7293
1000	88,99807	**80,84971**	1000	**89,42283**	118,2179

Table 5.11: selectR Test 2: **Cassandra** #R = 1000, changing $E_{replica}$

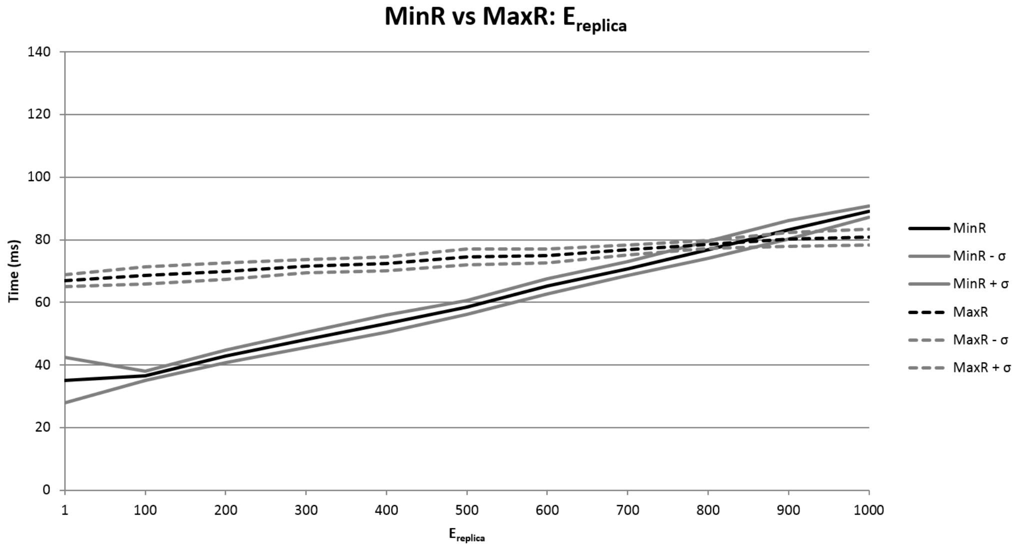


Figure 5.26: selectR Test 2: **Cassandra** #R = 1000, $Esize = 0$

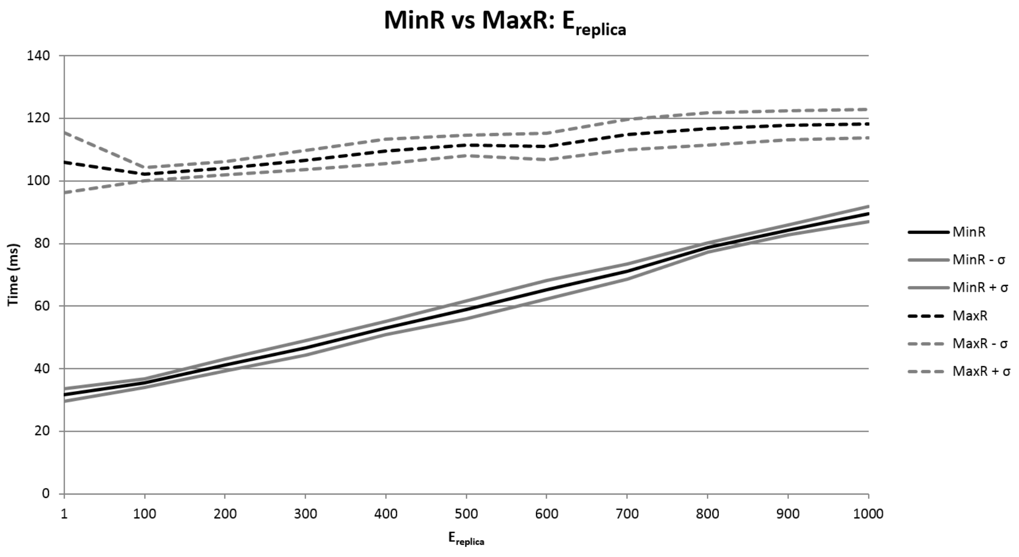


Figure 5.27: selectR Test 2: **Cassandra** #R = 1000, $Esize$ = 5000

	#R = 1000 $E_{replica}$ **= 1**			**#R = 1000** $E_{replica}$ **= 1000**	
$Esize$	**MinR[ms]**	**MaxR[ms]**	$Esize$	**MinR[ms]**	**MaxR[ms]**
0	**35,17193**	66,84498	0	88,99807	**80,84971**
10	**31,75729**	66,91108	10	88,52652	**80,60569**
100	**30,66036**	68,2432	100	91,56112	**82,39226**
200	**30,81841**	68,40364	200	89,03774	**82,67459**
300	**31,3144**	68,80326	300	90,51434	**84,51141**
400	**31,19999**	69,08822	400	88,05873	**85,7217**
500	**30,51166**	70,33656	500	89,83986	**85,38014**
600	**30,56986**	71,0129	600	**88,9346**	91,15357
750	**31,40501**	81,68767	750	**89,58988**	94,06008
1000	**30,80501**	74,16554	1000	**90,40885**	91,19213
2000	**32,3627**	87,12384	2000	**90,82746**	99,23354
3000	**31,3909**	87,02209	3000	**90,61795**	105,5642
5000	**31,61055**	105,8703	5000	**89,42283**	118,2179

Table 5.12: selectR Test 2: **Cassandra** #R = 1000, changing $Esize$

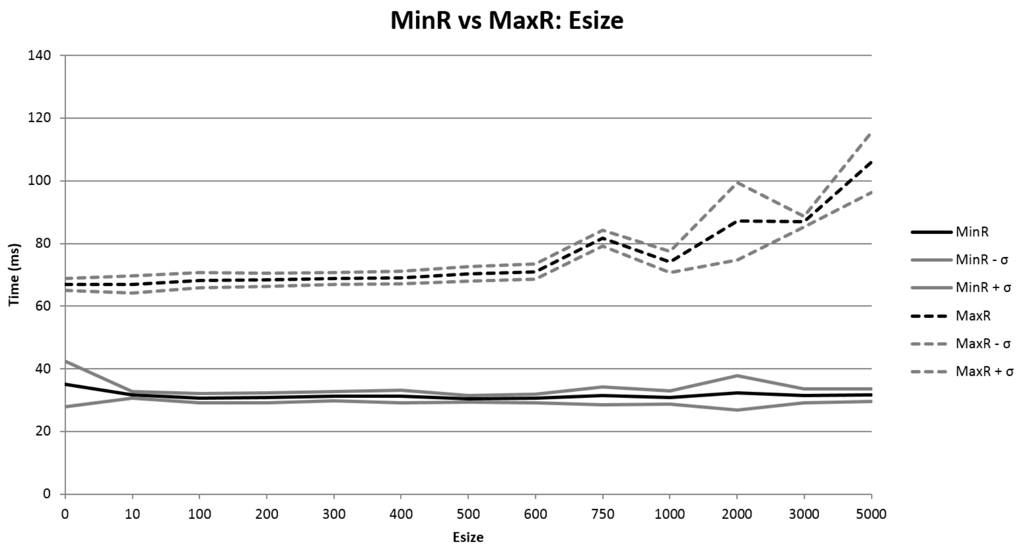


Figure 5.28: selectR Test 2: **Cassandra** #R = 1000, $E_{replica} = 1$

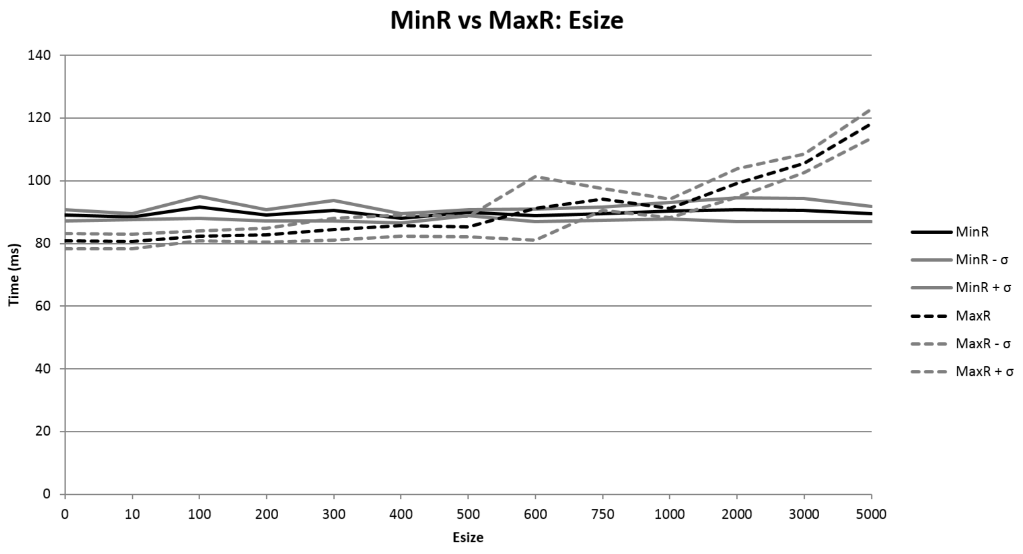


Figure 5.29: selectR Test 2: **Cassandra** #R = 1000, $E_{replica} = 1000$

many reads. More requests with a small number of columns is better than one request with many columns. MaxR relation table has more columns than MinR one. This is because MaxR fetching is slow.

However, when the number of rows to fetch grows, MinR has to read a lot of data from the index (`ekeys`). MaxR instead maintain the same overhead of multiple column fetch but it is spread over several data. This is because the MinR growing is higher than the MaxR one.

5.7 Testing Update Entity Operator (`updateE`)

Updating an entity is the other important operation to compare between MinR and MaxR strategies.

The usefulness of this test is not to know when one is better than the other, because as we have seen (cf. Section 4.4) the MinR is always better, but it is to see how much time has to be elapsed to be sure that all replicas of the entity involved will be updated, that is how long it takes the data set to be strictly consistent.

Remember the concept of Eventual Consistency (cf. 2.3.2), or that the data set may not be consistent during a certain time interval, but at the end of this must return to be strictly consistent.

We did two tests on this topic, the first just to confirm our hypothesis that update a tuple does not depend on the cardinality of the set to which it belongs (Test 1: #R grows). The second is the real measurement of what we have call the "Time to Consistency" where is the replica number of the entity that we have to update, the parameter that changes (Test 2: #Replica grows).

5.7.1 Test 1: #R grows

Test Objectives

Like we have said in the introduction, in this test we change the $\#R$ value, creating new relations not related with the entity that must be updated.

At every predefined step of $\#R$ we log the "Update Entity" operation on the E entity with $key = 2$, named here e_2.

For MinR this is translated into only one update in the E set, but for MaxR this means that in addition to the primary entity update, all the replicas in the R set are to be updated.

In this first test there are any replica of the e_2 entity in the R set, because all the relation are related to e_1.

That's why we expect a similar behavior between two models and that it is to be constant in the entire test (changing #R).

We also draw a graph that show the impact of growing size of entity on the update time (of the only copy) just to compare the sensibility of different implementations. This is only a secondary test, useful for clarifying ideas about how some behaviors differ in different databases.

Test Scenario

Setup:

1. Create *Rcount* entities in F
2. Create 2 entities in E
3. Create only one relation in R between e_1 and f_1 entities.

Test:

1. Create new relations in R between e_1 all the F entities f_i with i from 2 to $\#F$ and log `updateE('e', 2)` for several value of $\#R$.

Figures 5.30, 5.31 show the Test step for the MinR and MaxR models starting from a completed Setup. Algorithm 5.3 reports Setup and Test steps in pseudocode.

Test parameters:

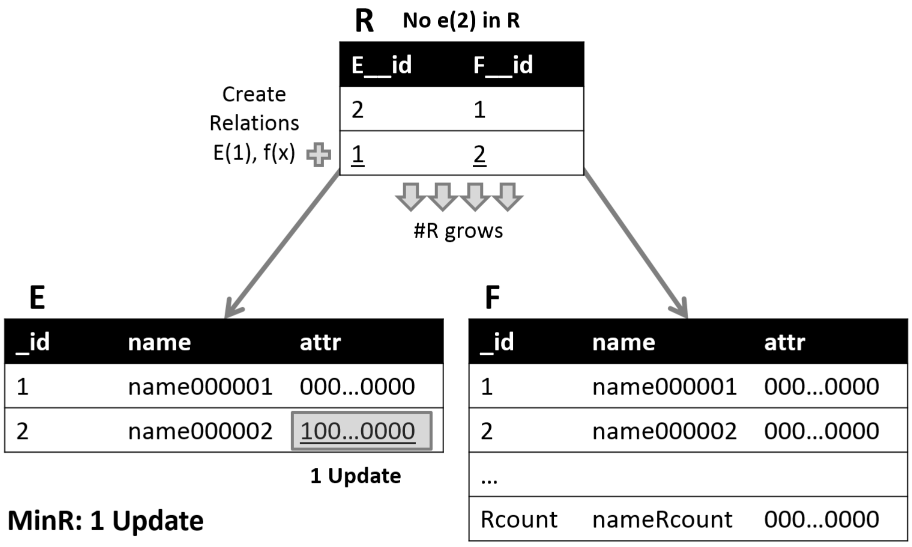
Rcount:	fixed number of relation
Esize:	fixed size of entities
Rounds:	fixed number of repeats of the operations
steps:	log steps

Schema static properties:

$$\begin{cases} kE &= 1 \\ kR &= 0 \\ kF &= 0 \\ \#E &= 2 \\ \#F &= Rcount \end{cases}$$

Schema variable properties:

$$\#R = from\ 1to\ \#F$$



R

E__id	F__id
2	1
1	2

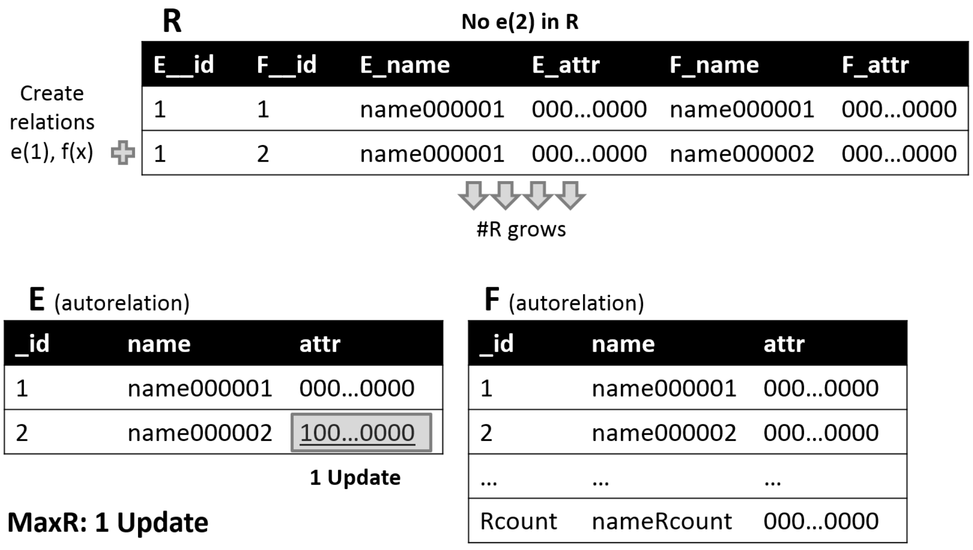
E

_id	name	attr
1	name000001	000...0000
2	name000002	100...0000

F

_id	name	attr
1	name000001	000...0000
2	name000002	000...0000
...		
Rcount	nameRcount	000...0000

Figure 5.30: MinR, Update Entity Test 1: #R grows



R

E__id	F__id	E_name	E_attr	F_name	F_attr
1	1	name000001	000...0000	name000001	000...0000
1	2	name000001	000...0000	name000002	000...0000

E (autorelation)

_id	name	attr
1	name000001	000...0000
2	name000002	100...0000

F (autorelation)

_id	name	attr
1	name000001	000...0000
2	name000002	000...0000
...	...	...
Rcount	nameRcount	000...0000

Figure 5.31: MaxR, Update Entity Test 1: #R grows

Algorithm 5.3 Test 1: #R grows

```
procedure TEST_UPDATEE_1
    // Setup
    Create $Rcount$ entities in $F$
    Create entity 1 in $E$
    Create entity 2 in $E$
    Create a relation in $R$ between $e_1$ and $f_1$
    // Test
    for $i = 2$; $i <= Rcount$; $i++$ do
        Create a relation in $R$ between $e_1$ and $f_i$ entity
        if log step is reached then
            Log Update Entity $e_2$                          ▷ LOG
        end if
    end for
end procedure
```

Test Hypothesis

What we expect is that both graphs (of MinR and MaxR models) are very similar, because both need to update only one copy of the entity.

In addition we expect also that the trends are constants when $\#R$ changes, because the update should use the hash map to find the entity to update by key and update it. These operations should be not related with the cardinality change.

Test Results

We have computed tests for some combinations of *Rcount* and *Esize*.

Rcount = { 1000, 2000, 3000, 4000, 5000, 10000 }

Esize = { 10, 100, 1000, 2000, 5000, 10000, 50000 }

Rounds depends on Rcount parameters, from 10 to 100 rounds

Primary Test: Changing #R In Table 5.13 and Figures 5.36, 5.37 we can see MongoDB results for the $Rcount = 1000$ (maximum $\#R$) when we change $\#R$ parameter, but maintaining $E_{replica} = 1$.

In Table 5.14 and Figures 5.38, 5.39 we can see instead the Cassandra results with the same characteristics.

	$Esize$ = **10**			$Esize$ = **50000**	
$\#R$	**MinR[ms]**	**MaxR[ms]**	$\#R$	**MinR[ms]**	**MaxR[ms]**
1	**31,8223**	30,7151	1	**30,2583**	33,8341
100	35,0266	**34,4002**	100	**31,5904**	34,798
200	35,1162	**34,5868**	200	**30,4405**	37,6955
300	**34,6775**	35,7103	300	**33,2853**	36,1506
400	34,0873	**33,8325**	400	34,7739	**34,2236**
500	35,5768	**34,5058**	500	**31,4736**	38,8067
600	36,1522	**35,1458**	600	**30,2975**	35,4004
700	33,7718	**32,8884**	700	**29,1943**	38,7675
800	**34,852**	35,8465	800	**30,8552**	37,1202
900	35,51	**32,4941**	900	**29,2598**	36,8247
1000	**29,4852**	32,6846	1000	**34,2582**	38,6645

Table 5.13: updateE Test 1: **Mongo** changing $\#R$

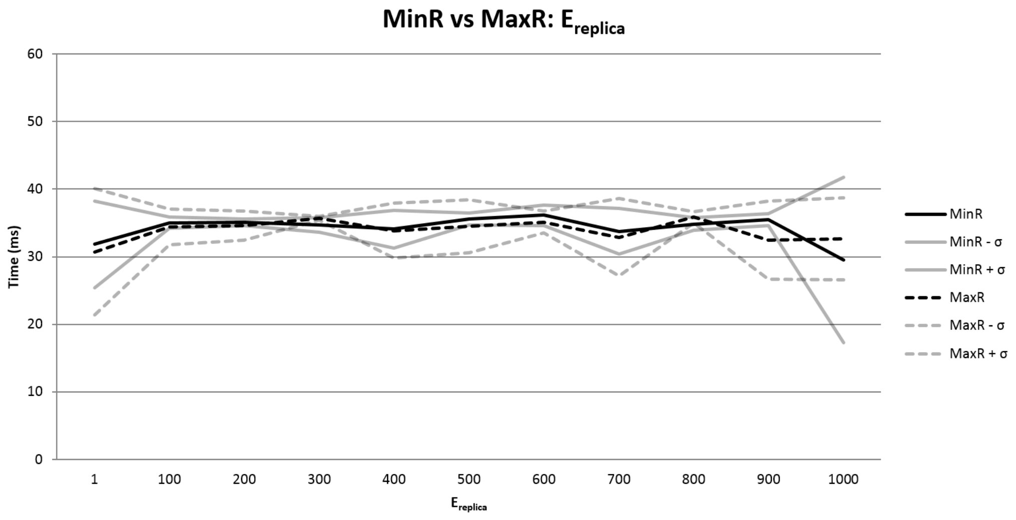


Figure 5.32: updateE Test 1: **Mongo** $\#R = 1000$, $Esize = 10$

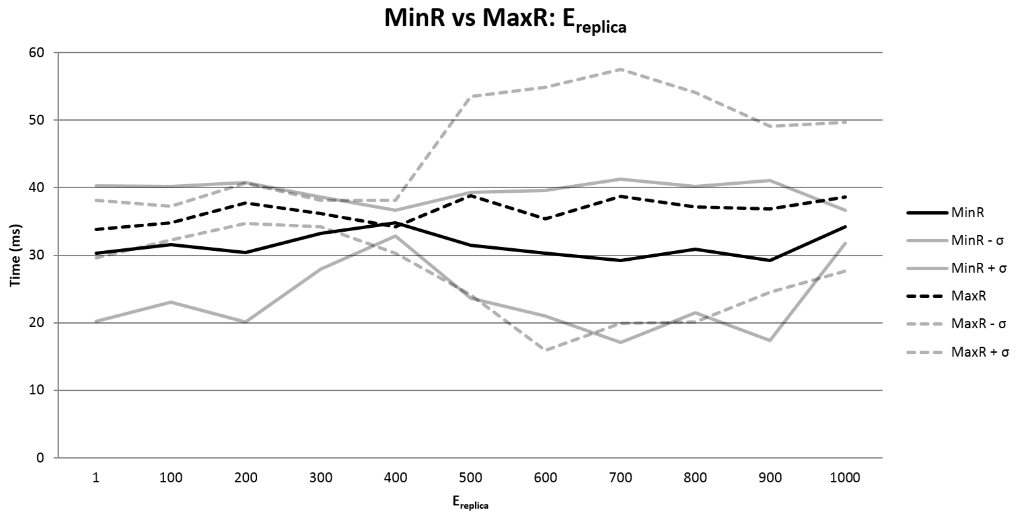


Figure 5.33: updateE Test 1: **Mongo** $\#R = 1000$, $Esize = 50000$

	$Esize$ = **10**			$Esize$ = **50000**	
$\#R$	**MinR[ms]**	**MaxR[ms]**	$\#R$	**MinR[ms]**	**MaxR[ms]**
1	**0,35022**	0,85054	1	**1,5987**	2,40636
100	**0,4003**	0,70046	100	2,00128	**1,20076**
200	**0,34995**	0,60039	200	**1,20492**	2,4016
300	**0,35024**	0,60065	300	**1,20096**	1,60118
400	**0,35028**	0,70044	400	**1,6009**	2,02494
500	**0,30016**	0,60039	500	**1,20078**	2,00274
600	**0,40025**	0,80251	600	**1,22964**	2,20148
700	**0,35024**	0,74981	700	**1,4008**	1,99898
800	**0,4003**	0,64987	800	**1,20504**	2,00344
900	**0,40023**	0,65046	900	**1,40268**	1,99868
1000	**0,35026**	0,65066	1000	**1,40298**	1,80114

Table 5.14: updateE Test 1: **Cassandra** changing $\#R$

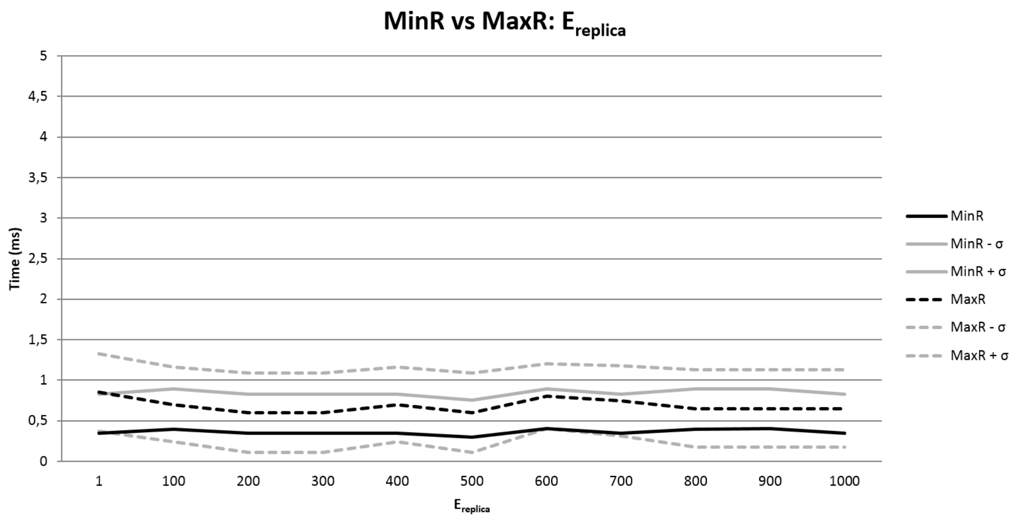


Figure 5.34: updateE Test 1: **Cassandra** $\#R = 1000$, $Esize = 10$

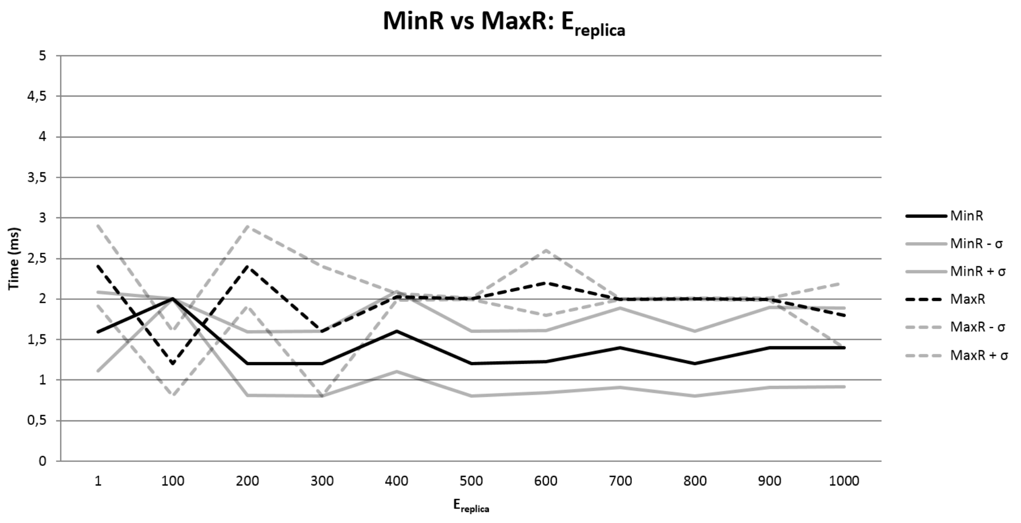


Figure 5.35: updateE Test 1: **Cassandra** $\#R = 1000$, $Esize = 50000$

Secondary Test: Changing Esize In Table 5.15 and Figures 5.32, 5.33 we can see MongoDB results for the $Rcount = 1000$ (maximum $\#R$) when we change $Esize$ parameter.

In Table 5.16 and Figures 5.34, 5.35 we can see instead the Cassandra results with the same characteristics.

#R = 1			**#R = 1000**		
Esize	**MinR**[ms]	**MaxR**[ms]	*Esize*	**MinR**[ms]	**MaxR**[ms]
10	31,8223	**30,7151**	10	**29,4852**	32,6846
100	**30,7604**	31,2476	100	36,677	**34,9913**
500	30,8749	**30,8089**	500	**34,0687**	36,4841
1000	**30,7136**	30,9987	1000	**33,1797**	34,4785
2000	33,5032	**30,9931**	2000	**32,7803**	33,8424
5000	30,9055	**30,7235**	5000	**34,9712**	35,2493
10000	30,8938	**30,2607**	10000	**33,3378**	36,9916
50000	**30,2583**	33,8341	50000	**34,2582**	38,6645

Table 5.15: updateE Test 1: **Mongo** changing $Esize$

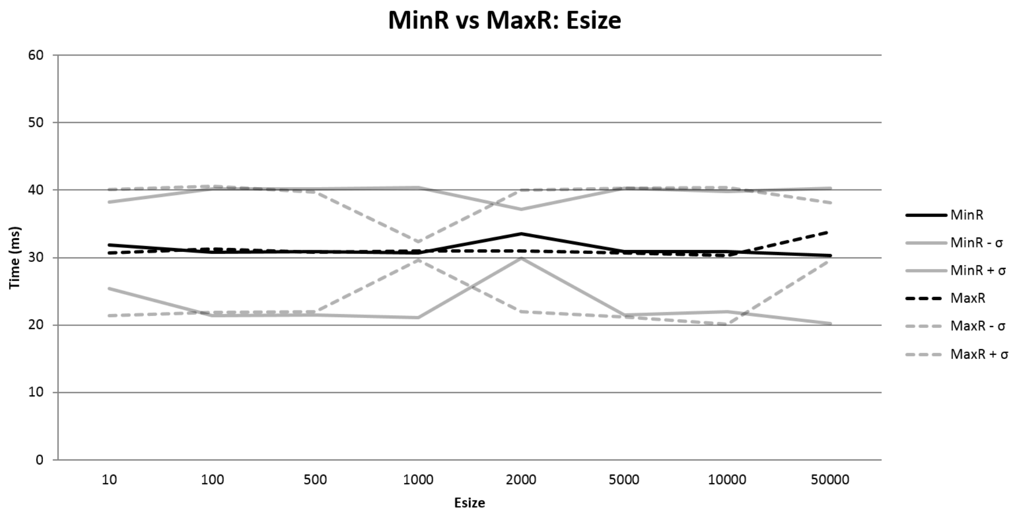


Figure 5.36: updateE Test 1: **Mongo** $kR = 1$

Test Discussion

From results we can see that our hypothesis were correct. In all tests where we change the $\#R$ parameter the graphs of the two models are very similar

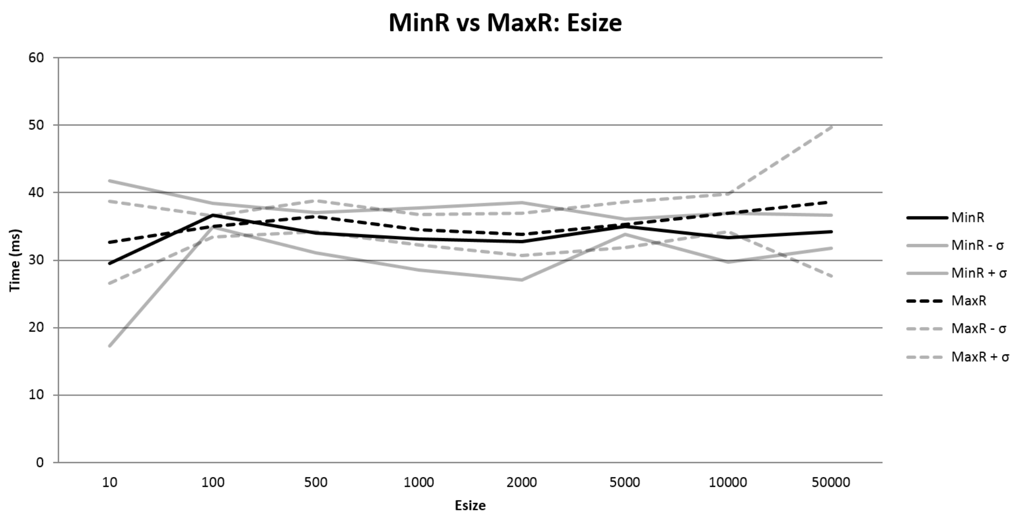


Figure 5.37: updateE Test 1: **Mongo** $kR = 1000$

	#R = 1			**#R = 1000**	
Esize	**MinR[ms]**	**MaxR[ms]**	*Esize*	**MinR[ms]**	**MaxR[ms]**
10	**0,35022**	0,85054	10	**0,35026**	0,65066
100	**0,3175**	0,65058	100	**0,50049**	0,60034
500	**0,3002**	0,6004	500	**0,45006**	0,75046
1000	**0,3502**	0,65037	1000	**0,15013**	0,70047
2000	**0,35978**	0,70046	2000	**0,35021**	0,60039
5000	**0,50031**	0,70016	5000	**0,44984**	0,70038
10000	**0,90101**	1,40087	10000	**0,60043**	1,09968
50000	**1,5987**	2,40636	50000	**1,40298**	1,80114

Table 5.16: updateE Test 1: **Cassandra** changing *Esize*

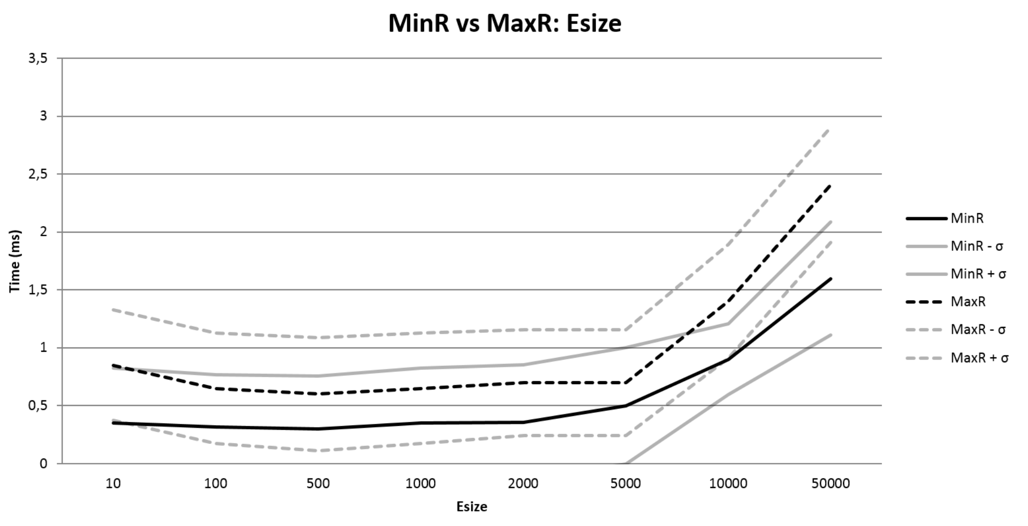


Figure 5.38: updateE Test 1: **Cassandra** $\#R = 1$

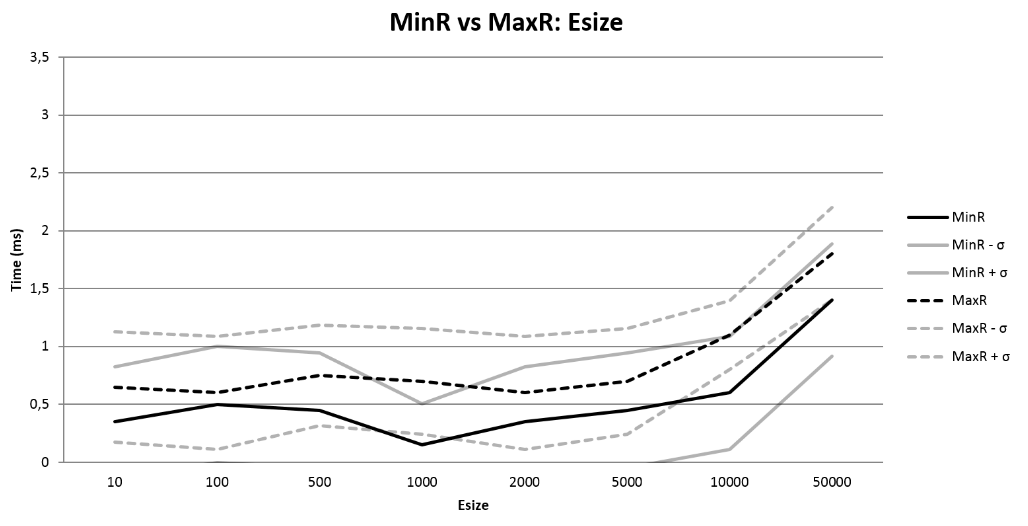


Figure 5.39: updateE Test 1: **Cassandra** $\#R = 1000$

and with a constant behavior. Then the $\#R$ value is not related to the update cost and, also, the two models have no difference if there is no other entity copy in addition to the primary one.

We can also see that Cassandra is really faster than Mongo in write operations. This was predictable because Cassandra is optimized for writing and MongoDB excels in reading (cf. Sections 2.4.2 and 2.4.3)

In the secondary tests, where we change the $Esize$ maintaining a fixed $\#R$ the numbers show that Cassandra is more sensitive to the changes in tuple size than MongoDB, in fact the Mongo graphs (5.36 and 5.37) are constants, instead ones of Cassandra (5.38 and 5.39) rise with a big $Esize$.

5.7.2 Test 2: #Replica grows

Test Objectives

In this test, we really investigate the burden of updating many replicas of the same entity.

We start from the schema produced by the running of the previous test (cf. Test 1: #R grows), with *Rcount* F entities all in relation with the first of the two E entities $e(1)$, and changing step by step all the relations of the R set, from $e_1 \to f_i$ to $e_2 \to f_i$, maintaining constant the R cardinality (this is unnecessary given the previous test results, but it is introduced to make the test independent from others).

Doing this each new relation contains a replica of the entity e_2 and they must be updated by “Update Entity” operation.

In the predefined steps we launch and log the update operation.

With these graphs we can evaluate the time that is needed to reach the strict consistency.

This can be an indication of when to do sharding, bacause if we have a time constraint on consistency we can read on the graph the maximum number of replicas that is compatible with this. Then we can know the maximum cardinality of R table on the single node before to do sharding, because the worst-case is when we have replicas of the same entity on each tuple.

Test Scenario

Setup: (we can start from the output database of Test 1: #R grows or do these tasks)

1. Create *Rcount* entities in F
2. Create 2 entities in E

3. Create only one relation in R between e_1 and f_1 entities.

Test:

1. Create new relations in R between e_1 all the F entities f_i with i from 2 to $\#F$ and log `updateE('e', 2)` for several value of $\#R$.

Figures 5.40, 5.41 show the Test step for the MinR and MaxR models starting from a completed Setup. Algorithm 5.4 reports Setup and Test steps in pseudocode.

Test parameters:

Rcount:	fixed number of relation
Esize:	fixed size of entities
Rounds:	fixed number of repeats of the operations
steps:	log steps

Schema static properties:

$$\begin{cases} kE & = 1 \\ kF & = 0 \\ \#E & = 2 \\ \#F & = Rcount \end{cases}$$

Schema variable properties:

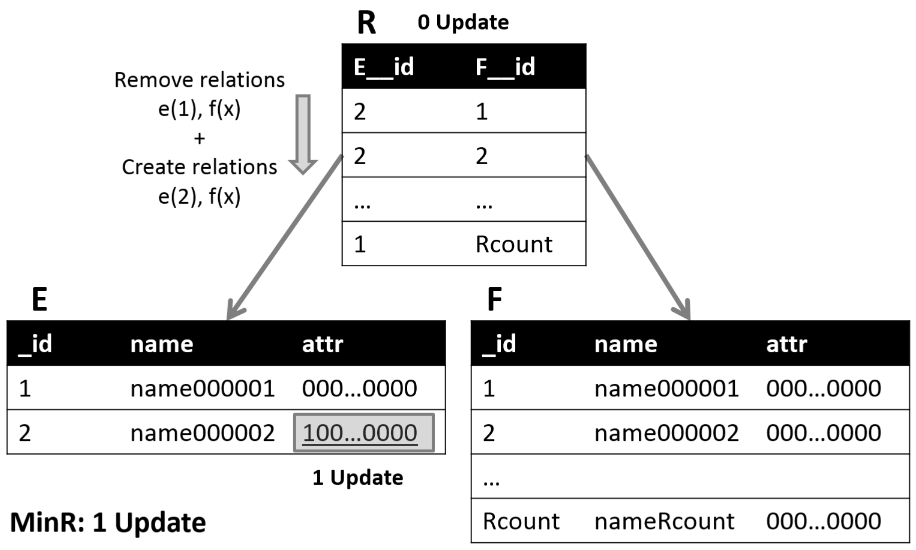
$$\begin{cases} \#R & = from\ 1 to\ \#F \\ kR & = \#R \end{cases}$$

Test Hypothesis

What we expect is that the inclusion of new relations with the entity $e(2)$ does not cause any cost change in the MinR model. In MaxR instead we expect a linear and proportional increase of the update time, depending to the number of entity replicas, which is equal to the number of relations created.

A hypotetical chart that summarize these conditions is reported in Figure 5.42.

We also expect that if we have more than one relation that contains replicas of E entities we can take the sum of maximum estimated cardinality

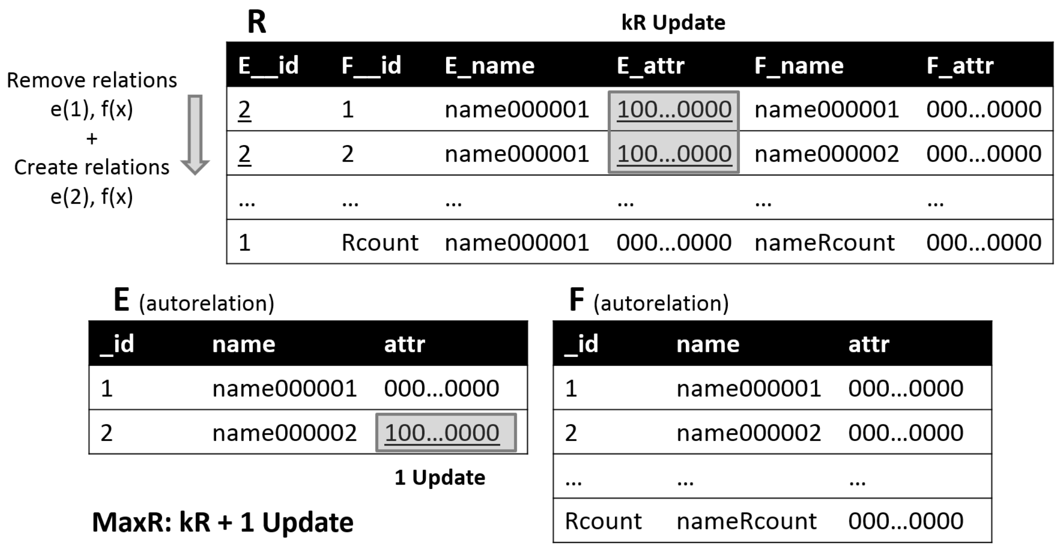


E__id	F__id
2	1
2	2
...	...
1	Rcount

_id	name	attr
1	name000001	000...0000
2	name000002	100...0000

_id	name	attr
1	name000001	000...0000
2	name000002	000...0000
...		
Rcount	nameRcount	000...0000

Figure 5.40: MinR, Update Entity Test 2: #Replica grows



E__id	F__id	E_name	E_attr	F_name	F_attr
2	1	name000001	100...0000	name000001	000...0000
2	2	name000001	100...0000	name000002	000...0000
...	...	...	...	...	...
1	Rcount	name000001	000...0000	nameRcount	000...0000

_id	name	attr
1	name000001	000...0000
2	name000002	100...0000

_id	name	attr
1	name000001	000...0000
2	name000002	000...0000
...	...	...
Rcount	nameRcount	000...0000

Figure 5.41: MaxR, Update Entity Test 2: #Replica grows

Algorithm 5.4 Test 2: #Replica grows

```
procedure TEST_UPDATEE_2
    // Setup
    Create $Rcount$ entities in $F$
    Create entity 1 in $E$
    Create entity 2 in $E$
    Create relations in $R$ between $e_1$ and all the $F$ entities
    // Test
    for $i = 2; i <= Rcount; i++$ do
        Delete a relation in $R$ between $e_1$ and $f_i$
        Create a relation in $R$ between $e_2$ and $f_i$ entity
        if log step is reached then
            Log Update Entity $e_2$                                ▷ LOG
        end if
    end for
end procedure
```

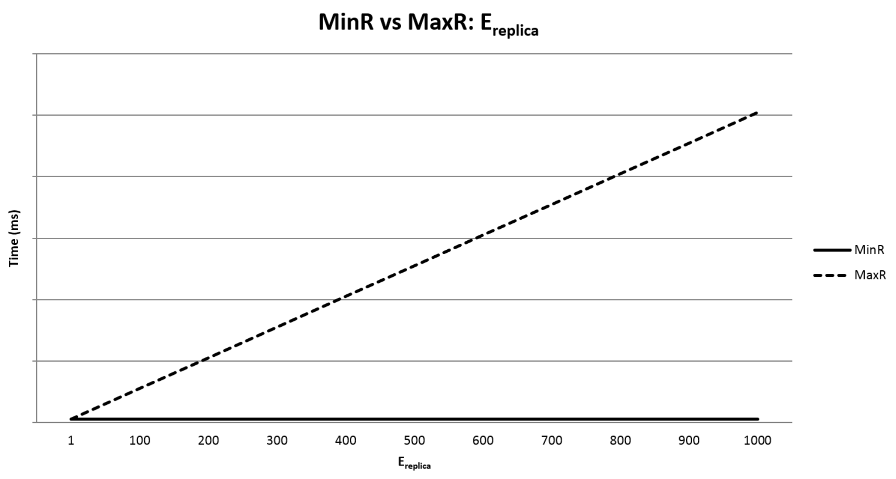


Figure 5.42: Hypotetical chart, Update Entity Test 2: #Replica grows

and suppose to have only one relation with this maximum number of tuples. To check this we'll use the costs measured for two different relations, R1 and R2, with $\#R1 = 2 * \#R2$, and we'll compare R1 costs with two times R2 costs. This because in single node the operations should be serialized.

Test Results

We have computed test for some combinations of *Rcount* and *Esize*.

Rcount = { 1000, 2000, 3000, 4000, 5000, 10000 }

Esize = { 10, 100, 1000, 2000, 5000, 10000, 50000 }

Rounds depends on Rcount parameters, from 10 to 100 rounds

Primary Test: Changing kR In Table 5.17 and Figures 5.47, 5.48 we can see MongoDB results for the $Rcount = 1000$ when we change kR (and then $E_{replica}$) parameter.

In Table 5.18 and Figures 5.49, 5.50 we can see instead the Cassandra results with the same characteristics.

	#R = 1000			**#R = 1000**	
	$Esize$ = **10**			$Esize$ = **50000**	
kR	**MinR[ms]**	**MaxR[ms]**	kR	**MinR[ms]**	**MaxR[ms]**
1	**31,9404**	33,917	1	**33,745**	48,3451
100	35,9996	**32,4833**	100	**33,2208**	55,6878
200	33,5835	**29,9019**	200	**36,3131**	62,5308
300	**33,7695**	36,6307	300	**36,16**	60,5604
400	37,4394	**36,2438**	400	**31,438**	71,3857
500	**33,6957**	39,1019	500	**33,1189**	75,6601
600	**31,2809**	34,2199	600	**34,4792**	79,9427
700	33,8348	**32,3456**	700	**35,8115**	81,2379
800	**31,237**	34,0274	800	**35,4948**	108,297
900	38,2122	**36,2134**	900	**38,3002**	105,923
1000	**33,1966**	45,2967	1000	**32,0932**	123,238

Table 5.17: updateE Test 2: **Mongo** #R = 1000, changing kR

Secondary Test: Changing Esize In Table 5.19 and Figures 5.43, 5.44 we can see MongoDB results for the $Rcount = 1000$ when we change $Esize$ parameter.

In Table 5.20 and Figures 5.45, 5.46 we can see instead the Cassandra results with the same characteristics.

Secondary Test: $R = R1 + R2$

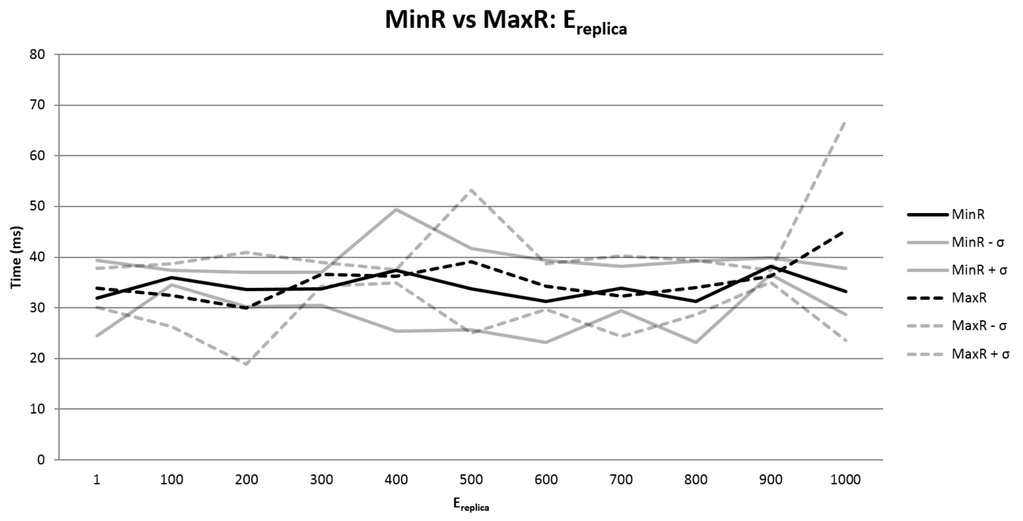


Figure 5.43: updateE Test 2: **Mongo** #R = 1000, $Esize = 10$

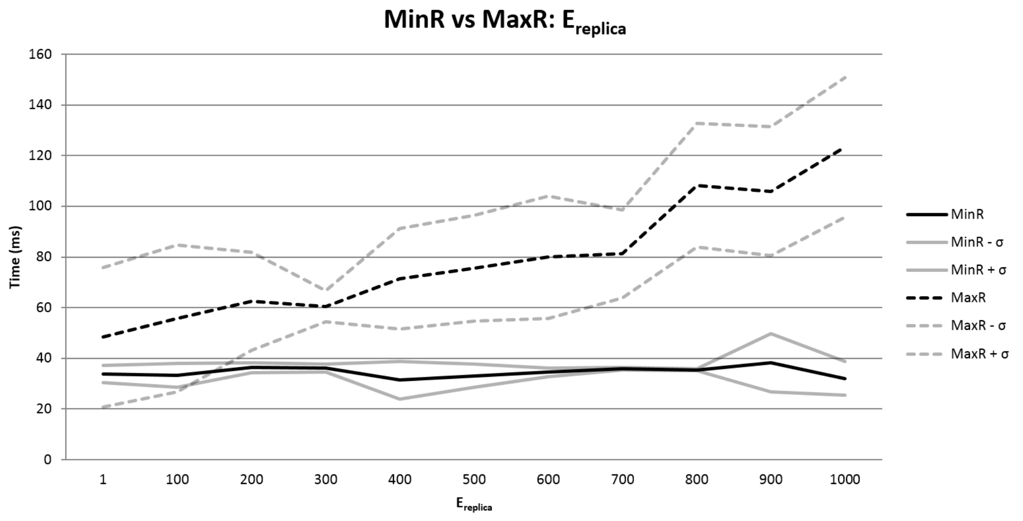


Figure 5.44: updateE Test 2: **Mongo** #R = 1000, $Esize = 50000$

	#R = 1000			**#R = 1000**	
	Esize = **10**			*Esize* = **50000**	
kR	**MinR[ms]**	**MaxR[ms]**	*kR*	**MinR[ms]**	**MaxR[ms]**
1	**0,4503**	1,15112	1	**1,59886**	4,0026
100	**0,40021**	7,70475	100	**1,60108**	76,78
200	**0,40026**	13,9648	200	**1,00066**	148,124
300	**0,45029**	20,5998	300	**1,20072**	230,36
400	**0,45032**	27,716	400	**1,20088**	299,332
500	**0,30018**	34,8797	500	**1,00314**	392,229
600	**0,40028**	41,6618	600	**1,19862**	543,322
700	**0,3002**	50,1999	700	**1,00194**	607,614
800	**0,35022**	56,8618	800	**1,20072**	685,147
900	**0,35021**	65,2452	900	**1,00068**	725,928
1000	**0,44977**	72,5408	1000	**1,20104**	820,332

Table 5.18: updateE Test 2: **Cassandra** #R = 1000, changing *kR*

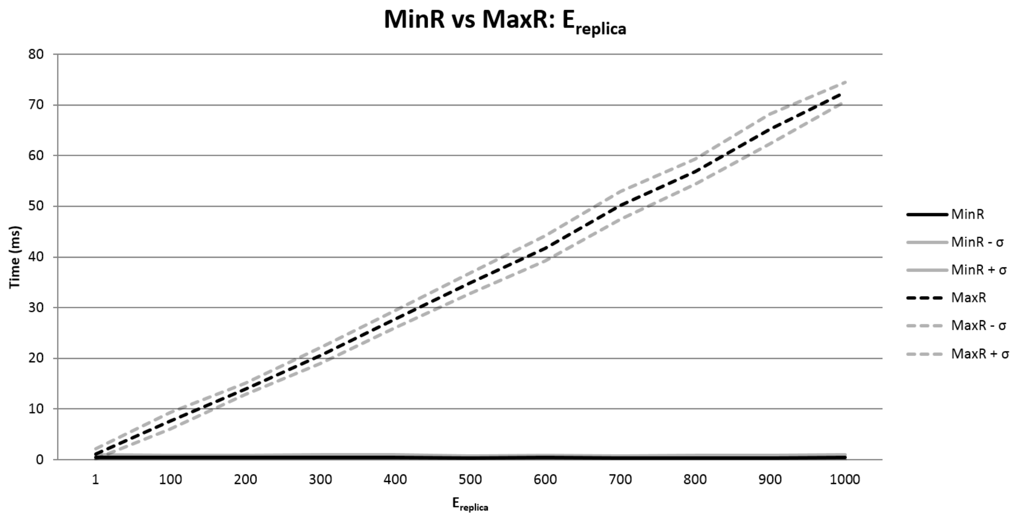


Figure 5.45: updateE Test 2: **Cassandra** #R = 1000, *Esize* = 10

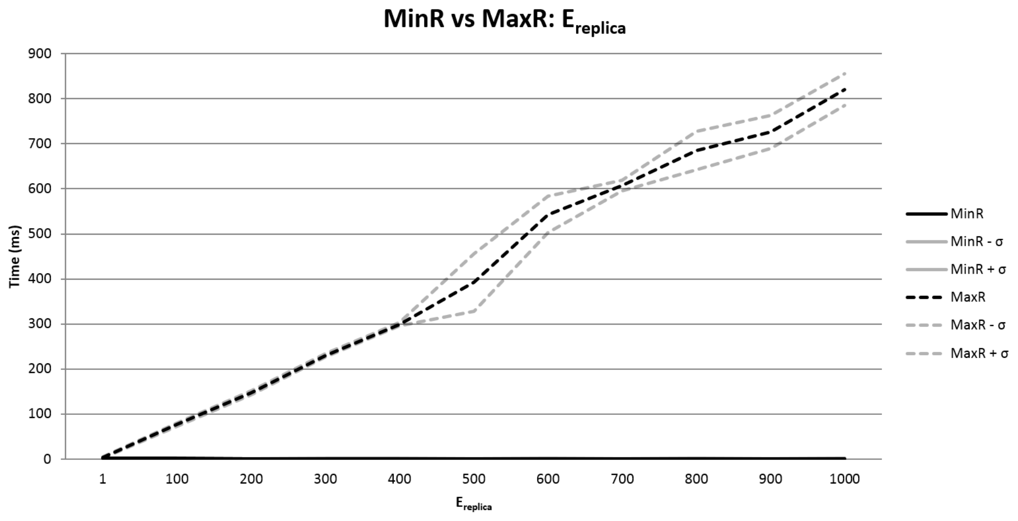


Figure 5.46: updateE Test 2: **Cassandra** #R = 1000, $Esize$ = 50000

	#R = 1000 kR **= 1**			**#R = 1000** kR **= 1000**	
$Esize$	**MinR[ms]**	**MaxR[ms]**	$Esize$	**MinR[ms]**	**MaxR[ms]**
10	**31,9404**	33,917	10	**33,1966**	45,2967
100	35,8403	**33,5723**	100	35,886	**34,5681**
500	35,1713	**33,9348**	500	**33,0506**	33,9461
1000	33,1724	**32,5183**	1000	38,6555	**37,9264**
2000	**32,0577**	33,3793	2000	**32,7037**	42,6057
5000	**31,0045**	32,9033	5000	**35,3309**	42,6326
10000	**34,0442**	34,4822	10000	**32,6591**	49,2828
50000	**33,745**	48,3451	50000	**32,0932**	123,238

Table 5.19: updateE Test 2: **Mongo** #R = 1000, changing $Esize$

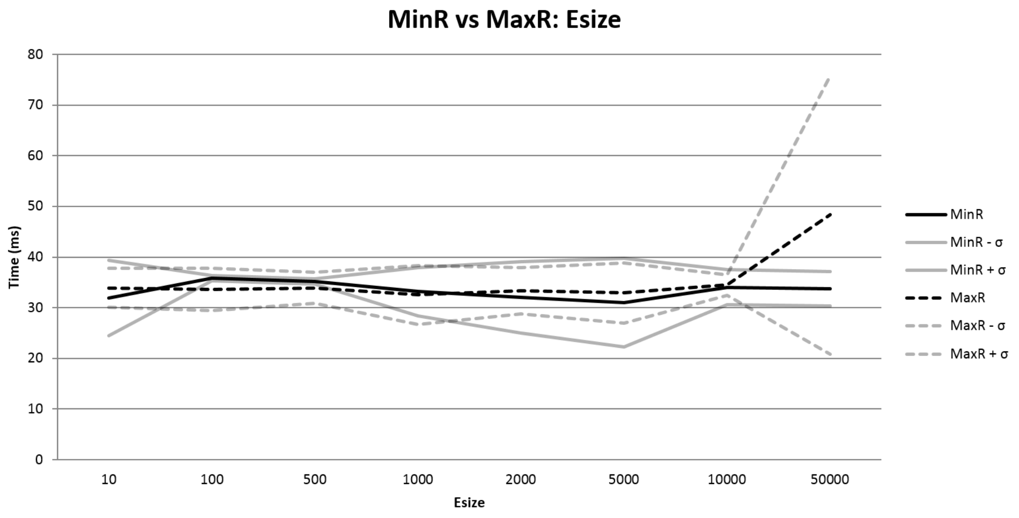


Figure 5.47: updateE Test 2: **Mongo** #R = 1000, $kR = 1$

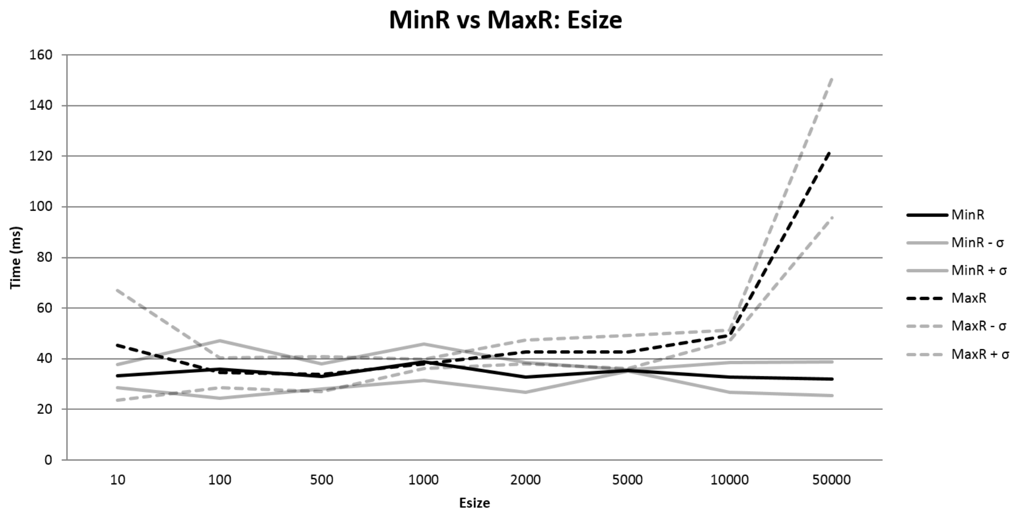


Figure 5.48: updateE Test 2: **Mongo** #R = 1000, $kR = 1000$

	#R = 1000	
	***kR* = 1**	
Esize	**MinR**[ms]	**MaxR**[ms]
10	**0,4503**	1,15112
100	**0,35822**	0,90062
500	**0,40027**	0,95305
1000	**0,39947**	0,95065
2000	**0,40029**	1,40139
5000	**0,50093**	1,25086
10000	**0,80057**	1,90625
50000	**1,59886**	4,0026

	#R = 1000	
	***kR* = 1000**	
Esize	**MinR**[ms]	**MaxR**[ms]
10	**0,44977**	72,5408
100	**0,20014**	73,608
500	**0,50141**	80,9899
1000	**0,30017**	100,276
2000	**0,35024**	107,369
5000	**0,55103**	151,771
10000	**0,50027**	244,929
50000	**1,20104**	820,332

Table 5.20: updateE Test 2: **Cassandra** #R = 1000, changing *Esize*

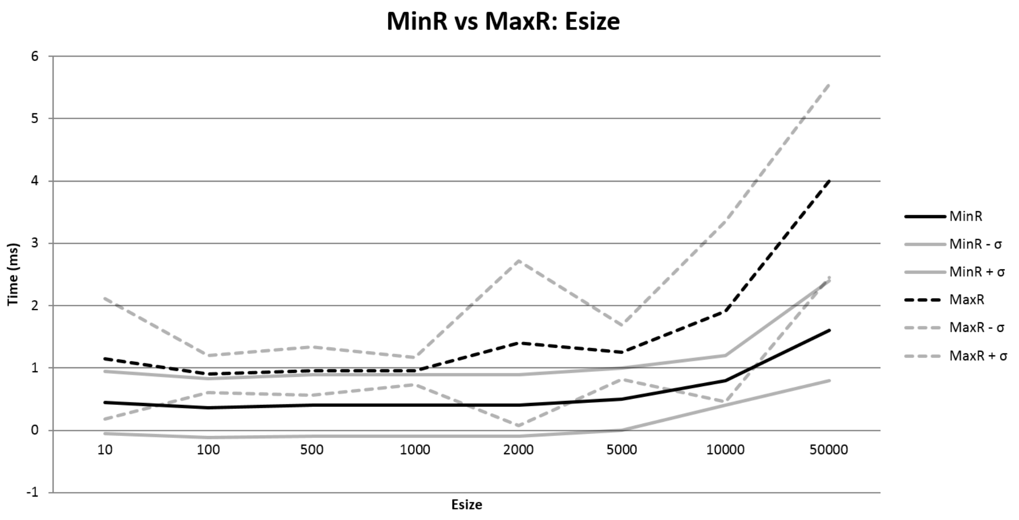


Figure 5.49: updateE Test 2: **Cassandra** #R = 1000, $kR = 1$

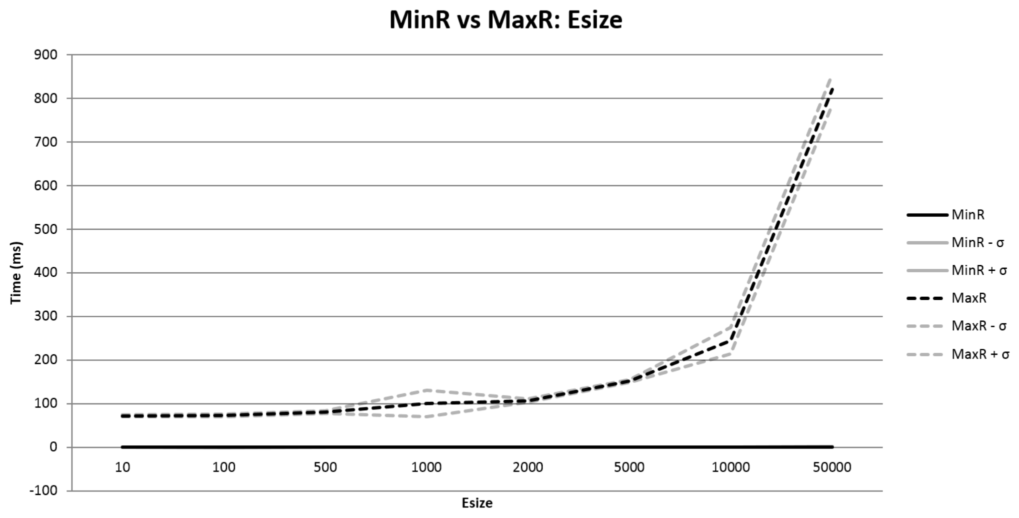


Figure 5.50: updateE Test 2: **Cassandra** #R = 1000, $kR = 1000$

#R1 = 1000, #R2 = 500

Esize	**R1[ms]**	**R2*2[ms]**	**R1-R2[ms]**	**R1/R2**
10	45,2967	68,26288	-22,96618	0,663562686
100	34,5681	66,34228	-31,77418	0,521056858
500	33,9461	68,01642	-34,07032	0,499086838
1000	37,9264	68,27252	-30,34612	0,555514869
2000	42,6057	70,71884	-28,11314	0,602466047
5000	42,6326	75,09746	-32,46486	0,567696963
10000	49,2828	87,8041	-38,5213	0,561281307
50000	123,238	153,71652	-30,47852	0,801722547

Table 5.21: updateE Test 2: **Mongo** #R = #R1 + #R2 = 1000, changing *Esize*

	#R1 = 1000, #R2 = 500			
$Esize$	**R1**[ms]	**R2*2**[ms]	**R1-R2**[ms]	**R1/R2**
10	72,5408	66,63457333	5,906226667	1,088636069
100	73,608	76,97807333	-3,370073333	0,956220347
500	80,9899	76,67286	4,31704	1,056304669
1000	100,276	83,70699333	16,56900667	1,19794053
2000	107,369	97,98050667	9,388493333	1,095820012
5000	151,771	142,0740667	9,696933333	1,068252663
10000	244,929	223,95484	20,97416	1,093653524
50000	820,332	835,91526	-15,58326	0,981357847

Table 5.22: updateE Test 2: **Cassandra** #R = #R1 + #R2 = 1000, changing $Esize$

Test Discussion

From results we can see that the MaxR graphs in Cassandra grow linearly in proportion to the number of $E_{replica}$ and MinR is always constant. This is equal to what we supposed. The good is that Cassandra behavior is predictable.

MongoDB instead has a strange behavior with small numbers. When the $Esize$ is less then about 2000 the graph, with increasing $E_{replica}$, is constant and it is similar with the MinR one. An explanation could be given by the fact that MongoDB performs the write operations to main memory (and write to a log file the operations to ensure durability) and only when a predefined chunk of memory is full it performs a flush to disk.

Below to a certain threshold, probably, for both MinR and MaxR the size of updated data does not fill more than one chunk and then flush to disk takes place only when the connection is closed at the and of test, giving the same time in both tests. For bigger $Esize$ MaxR use more chunks in memory and then the linear increment of cost is visible.

Another effect that probably derives from this difference in writing is that MongoDB is much less sensitive to increasing $Esize$ than Cassandra. In fact, Mongo is clearly faster than Cassandra with large objects, while Cassandra wins with small ones.

In the second test we can check this behavior: in Mongo graph (5.48 the two trends is very similar until the $Esize$ reach the 2000 point and then the MaxR grows faster than MinR, instead for Cassandra (5.50) the MaxR growing factor is always bigger, from the start ($Esize = 10$).

We have also found that in Cassandra we can take the sum of maximum cardinality of all the relations that contains E entities replica and use this number to launch our tests like if we have only one big relation. Check the Table 5.22. There are small difference between update on one R (R1) and on two R with half cardinality of R1 (we have used the double of the costs logged for R2, with $\#R1 = 2 * \#R2$).

Instead, for MongoDB there is a difference, but it is fixed around 30ms which is the time of minimal write operation for mongodb (cf Table 5.21).

With these graphs if we have some time constraint on consistency we can launch the test with supposed $Esize$ and with *Rcount* equal to the maximum number of estimated $E_{replica}$ and find the maximum $\#R$ that is compatible with our constraint.

If we have more than one relation, we can divide this number on all of them following the same proportion of the maximum estimated cardinalities. If we are using MongoDB before evaluating results we have only to subtract from them the contributes of additional operations (30ms for each relations in addition to the first on our machine).

In conclusion we can say that if you want to use Cassandra in a write-heavy scenario and if you need a fast consistency convergence, MinR model is definitely the best, as expected.

In MongoDB instead you can also evaluate the strategy MaxR if the relation cardinality is not too high ($\#R < 2000$ on test machine), for the particular behavior that it has in write operations.

5.8 Summary

In this chapter we have presented the tests that we have done and the results on MongoDB and Cassandra implementations. Some results are not in line with what we have concluded with our theoretical reasoning. In Table 5.23 we summarize test results. For each test we have reported if the results of MongoDB and Cassandra are coherent with theoretical conclusions. In particular Cassandra has a strange behavior in `selectR` Test 1: #E vs #R and Mongo in `updateE` Test 2: #Replica grows.

Running these tests on a single node with the estimated parameters of a relation (*Rcount*, $Esize$, $E_{replica}$, …) give us graphs that can be read and interpreted to take decisions on which logical model to adopt (MinR vs MaxR objective) and on maximum size (cardinality) of the tested relation (Sharding objective).

Changing database means getting different results. In most cases the differences are only in numbers but sometimes the conclusions can change a lot. For example from our tests it is evident that for Cassandra the MinR

Operators	**Tests**	**MongoDB**	**Cassandra**
selectR	Test 1	Yes	Yes
	Test 2	Yes	No
updateE	Test 1	Yes	Yes
	Test 2	No	Yes

Table 5.23: Test Results Summary

model is usually the best, also when the MaxR should show its advantages. This happens because the column based data model is not suited to the MaxR model, that has many columns to be fetched together. In MongoDB instead MinR advantages are those that fall because the writes cannot be individually completed. So MaxR, that updates many replicas at the same time, is not so penalized in Mongo context.

In general we have to run the tests on the node before taking some decisions.

Chapter 6

Design of Evaluation Framework and its Implementation

6.1 Introduction

In this chapter we will describe the Evaluation Framework Architecture, how it works and how it was developed. Then we will expose what we have implemented to perform tests described in this thesis. From the results of these we compiled the tables presented in Chapter 5. We will also present some interesting points of our database provider implementations for MongoDB and Cassandra.

Finally we will plan some features to implement in the future releases.

6.2 Evaluation Framework Architecture

The Evaluation Framework is a system to run tests on a single Marijà node. It consist of some piece:

- A single Marijà Worker, that is responsible of database setup and which implements CRUD and Relational Algebra basic operators.
- A Tester, the benchmark platform responsible to run tests and store the results
- A set of Tests, that are the implemented tests to run

The Marijà Worker will be reusable in the Marijà project as single node basic operation performer. It is divided in two interface: the `ISetupManager` used for setup database schema and the `ICRUD` that implement all basic CRUD and Relational Algebra basic operators. The interfaces are to be implemented for the specific database to test. The implementation is called Database Provider. A schema of Marijà worker is shown in Figure 6.1.

A copy of the Tester is deployed directly on a Marijà node and use the Worker functionality to execute tests as shown in the Figure 6.2.

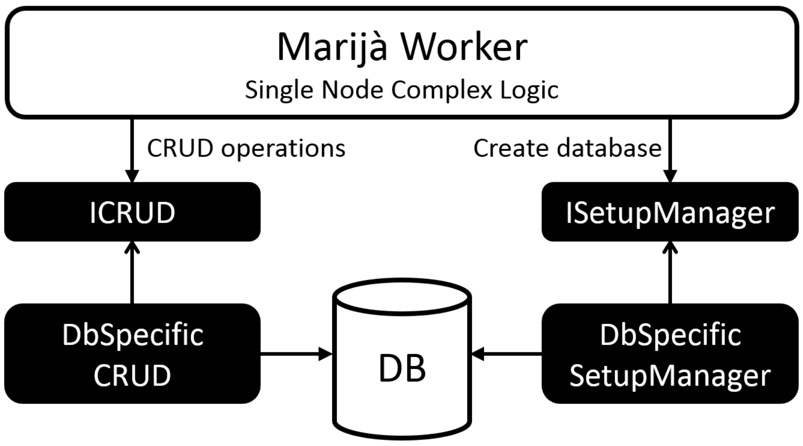


Figure 6.1: Marijà Worker

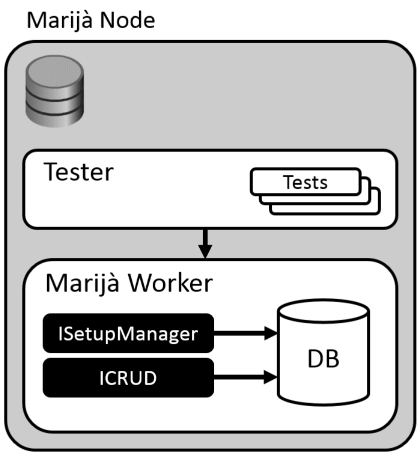


Figure 6.2: Marijà Node Deploy

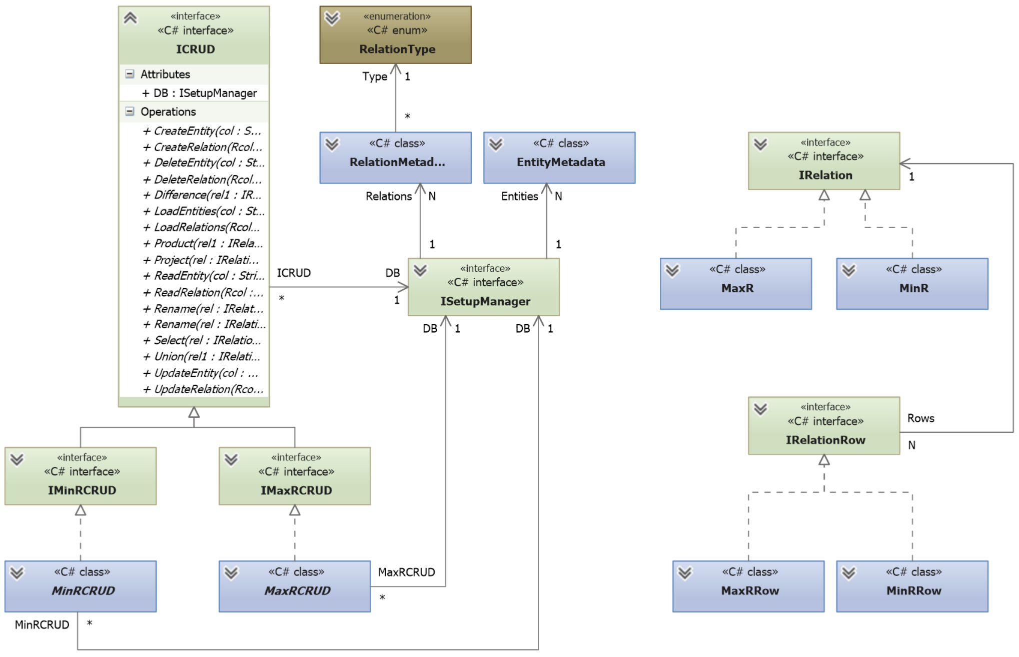


Figure 6.3: Marijà Worker Architecture

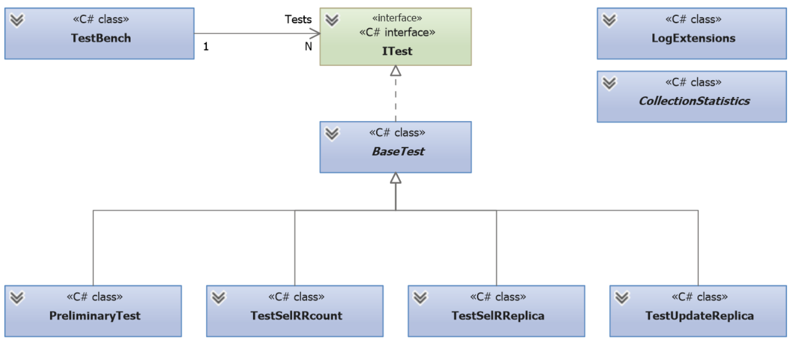


Figure 6.4: Tester Architecture

6.3 Abstract Model of Marijà Single Node

To design the Marijà node we used the classical object-oriented programming paradigm. Then we have implemented the entire architecture with the Microsoft .NET technology.

To present the architecture we present the main classes and interfaces. The UML Class Diagram of Marijà Worker and the Tester are shown respectively in Figures 6.3 and 6.4.

Marijà Worker classes and interfaces description

- `ISetupManager` is responsible to setup database. It contains a list of all relations and entities metadata
- `EntityMetadata` contains the properties of an entity
- `RelationMetadata` contains the properties of a relation, like name, type and connected entities
- `ICRUD` contains methods to execute the basic CRUD and Relational Algebra functionality on a database
- `IRelation` represents an in-memory structure of a query result. They are composed of a IRelationRow list
- `IRelationRow` represents a single relation, a single tuple of the results
- `MaxR, MinR` implements IRelation for the two logical models
- `MaxRRow, MinRRow` implements IRelationRow for the two logical models
- `IMaxRCRUD, IMinRCRUD` identify an ICRUD as implementation of MaxR or MinR model
- `MaxRCRUD, MinRCRUD` are abstract implementations of IMaxRCRUD and IMinRCRUD with the implementations of Relational Algebra operators that works on IMinR and IMaxR in memory structure and then they are not database dependent

Tester related classes and interfaces description

- `ITest` represent a test to run
- `BaseTest` is an abstract implementation of ITest with the base structure used to run my tests. It include a workflow to setup and destroy database, run tests and save logs
- `TestBench` is the test platform that is responsible to load database provider, choose the logical model to use and run tests
- `LogExtensions` is a utility class that contains the static methods to take logs
- `CollectionStatistics` is an abstract class that include methods to retrieve statistics on database collection, like cardinality, average size of objects and more. It is used in log phase. It is to extend with specific database implementation or with custom logic.

6.4 Tests Implementation

The Tester platform can run a list of tests that are classes which implement `ITest` interface.

We created a general abstract class `BaseTest`, that contains the common testing workflow of all our tests.

`BaseTest` is responsible to create database before test starts, to run tests, to save logs and to destroy database after the test. It contains a list of value for each parameter. The `BaseTest` parameter are:

- `Rcount`, that is the maximal cardinality of relation set R
- `Ecount`, that is the maximal cardinality of entity set E
- `Esize`, that is the size in bytes of the *attr* attribute of all entities. This is useful to change the size of each entity.

Tests are repeated for all combinations of configured parameters.

There is also a common test configuration class, `TestBench`. It contains the value of Log parameters:

- `Rounds`, number of repeated execution of fine grained logged operation
- `FullOperatorRounds`, number of repeated execution of coarse grained logged operation

- `VarianceRounds`, number of execution used to check variance. Check the `LogExtension` code in Appendice A

`TestBench` also contains three dictionary to store the expected collection statistics (objects count, objects average size and number of relevant objects) for the operation that is in execution. They must be updated in the specific test implementation. We have extended ***`CollectionStatistics`*** abstract class for handle them. The name of this class is `ExpectedCollection-Statistics`.

We wrote four test classes:

- `PreliminaryTest` extends `BaseTest` to run preliminary tests

- `TestSelRRcount` extends `BaseTest` to run the Select Relation Test 1: #E vs #R

- `TestSelRRcount` extends `BaseTest` to run the Select Relation Test 2: $E_{replica}$ Impact

- `TestSelRRcount` extends `BaseTest` to run the Update Entity Test 1: #R grows and Test 2: #Replica grows

6.5 Providers Implementation

The part of code commented in this section are reported in Section A.

We implemented two full database provider. One for MongoDB and the other for Apache Cassandra.

Creating a provider means implementing only three interfaces:

- `ISetupManager`
- `IMinRCRUD`
- `IMaxRCRUD`

6.5.1 MongoDB Implementation

We used the official 10gen-supported C# / .NET driver for MongoDB version 1.7 [1]. The driver is very stable and mature and allows to use all the features of the database, i.e. secondary indexing.

MongoSetupManager implements `ISetupManager` using the driver class `MongoClient` to configure the client parameters and `MongoServer` to establish connection. The disposable object is one returned by the `MongoServer RequestStart` method.

MongoDB collections need to be sometimes compacted. So the `CompactDB` method is implemented to run the “compact” command on all of them.

MongoMinRCRUD, MongoMaxRCRUD implements `IMinRCRUD` and `IMaxRCRUD` in a simple way. There are only some things to highlight: the use of the secondary indexing that MongoDB provides out-of-the-box, how we execute queries and how we have integrated logging.

The use of built-in secondary indexes has imposed a simplification of LoadRelations algorithm (which is equivalent to `selectR`). The `ekeys` operation is no longer to be called manually to retrieve the relation keys and then use them to fetch relation tuples. This mechanism is responsibility of MongoDB. The algorithm provide only the query directly on “X__id” attribute of relation collection (X is the related entity).

Querying of MongoDB is very simple. All operators that we need are provided by the driver. So we have only translated our query object in a IMongoQuery derivative object and we have used it to query database.

In Mongo implementation also the Update operators algorithms are slightly different. This is because MongoDB provides the update in-place feature: fetching the object is not needed to perform update. Only the attributes to be updated are sent to database.

Finally, to integrate log capability in Mongo query operations without making a dirty implementation, we have inserted it in the middle of MongoDB functions and LINQ ToList and ToDictionary methods. These methods are used in the Mongo CRUDs classes to execute the prepared query. We have created some Extension Methods [1] more specific than the standards LINQ ones, so our methods are called from Mongo classes. Then to log update there was no other way to do it without calling the log function manually. So we have extended (with another extension methods) the MongoCollection class with a LoggedUpdate method that internally call the base Update method but logging it.

[1]Extension Methods, an extension technique for C# explained in the Microsoft MSDN official documentation [13]

6.5.2 Apache Cassandra Implementation

To implement Cassandra provider we have chosen the FluentCassandra 1.2.2 after evaluating other libraries, as we exposed in Section 5.3.

FluentCassandra is not so stable and we had to overcome some problems during the run of tests, like an abnormal use of memory for example. We had to stop and restart tests sometimes to refresh the proper functioning. Another big problem was that using the same session to run multiple operations have the effect of slowing down incrementally each new task. This was not acceptable for our tests. So between each repetition of operation to log we have inserted a callingto th `Refresh` method that is responsible to destroy and recreate connection and reconfigure all that is needed.

CassandraSetupManager implements `ISetupManager` using the driver class `CassandraContext` to configure the client parameters and establish connection. The disposable object is the `CassandraContext` itself.

The `Refresh` method is implemented and it is necessary to log correct values with Cassandra like we said before.

CassandraMinRCRUD, CassandraMaxRCRUD implements respectively `IMinRCRUD` and `IMaxRCRUD` in a way that is very close to the theory. The `ekeys` operation is provided by our `CassandraSetupManager` in the `EKeys` method and, at lower lever, in `GetCentralIndex` method. In these we have implemented the One-to-Several Column Families Indexes pattern recommended in DataStax documentation explained in Section 5.3.2.

6.6 Summary

In this chapter we have presented the Evaluation Framework Architecture and the database provider implementations for MongoDB and Cassandra.

We have also described the most interesting parts of code with that can be checked in Appendice A.

Chapter 7

Discussion and Future Work

7.1 Summary of Work Done

NoSQL databases are certainly interesting, but choosing which one to use is a challenge for the programmers that decide to develop a scalable application. There are many differences between the available products. Some products are mature, others constantly updated. Several different low level data structures and indexes must be studied well before using them. There is a need for improving the programmability and manageability of NoSQLs. Definitely, the Marijà project would be an important introduction as a common interface for this world.

Starting from the Marijà assumption of single node independence, we focused on the modeling of relations between entities in a NoSQL.

We have proposed two opposite models for serializing application level entities connected through relations, MinR and MaxR. As we have extensively explained the difference between these two models is the fact that MinR keeps entities and relations between them in separate tables thus introducing an overhead whenever a relation has to be navigated from the entity. On the contrary, MaxR flattens in a single tuple all entities connected through some relations thus improving performance of query execution in spite of an increased overhead of a updated operations. From database schema perspective the only change between the two presented models is the presence of the denormalization of entities in relation tables. This has allowed us to define a Common Schema between the two models.

As mentioned above, the two serialization models offer different advantages and drawbacks. To support a user in selecting one of them, we have presented a method for comparing them in a complete way from the perspective of the typical CRUD (Create, Read, Update, Delete) operations a user may want to perform. More specifically, we had to identify for each operation what were the possible variants. Create, Update and Delete operations work only on one entity/relation at a time and in only one way. The Read operation instead has several variants as it can work on one or more objects, it can require the check of various predicates. Thus, we have distinguished between simple Read represented by the basic find by key operator (defined in two flavors for entities, `findE`, and relations, `findR`) and complex Read exploiting the expression of Relational Algebra. By studying in detail all possible ways of formulating a Read operation in Relational Algebra, we have seen that the comparison between MinR and MaxR can be reduced to the analysis of two simpler operation, that are `selectR` for relation reading and `selectE` for entity reading.

Having identified the main aspects to be compared, we have developed a framework that has allowed us to execute comparison tests on two different NoSQL databases, MongoDB and Cassandra. The analysis of the results of the tests has allowed us to gather conclusions not only on the relative advantages and disadvantages of MinR and MaxR, but also on the identification of what we call Sharding Point. Both aspects are briefly summarized.

MinR vs MaxR Our main objective was to comparing MinR and MaxR strategies with respect the properties of relations to store.

As regards the choice between MaxR and MinR, we have reached some conclusions:

- By theoretically analyzing MaxR we have seen that it is the best suited to read small amounts of data frequently. MinR instead is good to write often maintaining consistency.

- Preliminary tests confirm this and show that MaxR is preferable for sparse relations. MinR instead is the best when relations are full.

- Generally speaking, if we expect a lot of fluctuation in the relation cardinality we have to choose MinR, that always remains constant.

- In case of non-mandatory (1,1) or (1,n) relations, MaxR usually wins. Although if the relation cardinality become similar to the entity cardinalities, the performance of the two models are similar and for full relations MinR is slightly better. For strongly related objects instead MinR surely is the best, because in the MaxR there will be a too many replicas.

- MaxR is recommended with small entities because the MinR additional operations overhead weighs too much. With large entities instead MinR is preferable.
- We have verified that the increase in the cost of updating an entity is linear and proportional to the number of replicas. Furthermore it is not influenced by the cardinality of the entity set.

Sharding Point The secondary objective of the comparison was to find useful guidelines to choose when to perform the sharding of a node. This could be used for defining a partitioning strategy at design time and actuating it at runtime.

What we have concluded is that imposing some constraint on the single operations latency or on the time to reach consistency after an update, we can obtain the maximum cardinality of any relations in a single node. So, we have a sharding point. If we have to ensure a maximum latency constraint on single read operation, we can launch the `selectR` Test 1: #E vs #R with several values of #E (that imply several values of #R) and check when the constraint is satisfied. Furthermore, if we have to ensure a time constraint on consistency we can launch the `updateE` Test 2: #Replica grows with the expected schema parameters and check (from the resulting graph) the maximum number of replicas that is compatible with the constraint. Then we know the maximum cardinality of R table on the single node that satisfy us prerequisites. This is the Sharding Point.

7.2 Discussion

Findings when moving from theoretical analysis to practical testing on NoSQL The results on the two implementations show that in some cases the theoretical conclusions can be slightly different than the actual ones. Our experience with MongoDB and Cassandra has allowed us to find these deviations from the theory.

MongoDB has better overall performance than Cassandra, it looses only in small entities write concerns. This is because Mongo uses an in-memory data structure for reading and manipulation. However, to ensure durability of a single update, Mongo must wait the flushing of memory and this cancels the MinR advantage in updating an entity. So MaxR is a good choice also with a frequent update of entities.

Cassandra instead has a big penalty in reading many columns. This is because of the column-based organization. This penalize MaxR models.

MinR is usually the best choice for Cassandra.

Cassandra, likely all Column-oriented NoSQLs, could be a good choice only in a fully designed system where only the pre-engineered queries can be performed.

Selecting the right NoSQL for Marijà On what concerns the Marijà single node we have some suggestion on what NoSQL to use.

Initially we thought that the column-based is the data-model best suited for Marijà. This is because Marijà starts from a set of queries to optimize, and then we can build the correct access structures to optimize the readings. Furthermore, the column-based are meant to be distributed across multiple nodes.

However given the independence of individual nodes and given our practical experience, we can conclude that it is better to maximize performance on nodes and manually manage the internode communication. From this point of view the document-based NoSQL are the best option. They are scalable and provide some automatic sharding techniques, but are designed to work at their full potential on a single machine.

Experience in NoSQL programming We report here some of the NoSQL usage experiences that we have obtained during this thesis.

For general-purpose use, we recommend the use of Document-based NoSQL. They are more rich of features and easy to use. If you have to manage a very high amount of data, then the Column-oriented NoSQL are more suited than others. However, the structures for optimized data access must be designed carefully, analyzing a priori how the database will be queried, otherwise the performance decreases significantly. The key value are appropriate only for simple scenarios, such as for a distributed cache or when there is no need to use a particular data structure.

Before choosing what product to use you must also assess the infrastructure for which the database is designed. For example, MongoDB is designed to maximize performance on single node. Cassandra instead presupposes to be deployed over a multi-node network. Other products like Amazon DynamoDB or Google BigTable are tied to their cloud infrastructure.

Furthermore, the documentation and API availability are quite heterogeneous.

We can also trace a comparison of MongoDB and Cassandra for what concerns the developer vision:

Mongo

- It is very simple to install and does not need any critical configuration.
- It is very simple to use. There are many concept similar to what we can found in RDBMS or in Object-Oriented Programming Paradigm.
- It is very simple to design database schema, because of it does not need any particular data structure to enhance performance.
- The documentation is very complete, with a complete reference and several theoretical topics on architecture design.
- It is stable, during our experiments it has never crashed.
- Use of memory is very optimized. Even with several concurrent operations the central memory remain at stable level.
- Official driver for Microsoft .NET is very stable and complete.
- There are some third party administration tools.
- From API we can obtain statistical information on the document collections (Object Count, Avarage Object Size, etc.).

Cassandra

- It is more complicated to install and configure.
- To start using Cassandra a deep analysis of related architectural design patterns is needed.
- The documentation exist but it is not well organized, we have found more information in related site than in the official one.
- It need indexing to ensure good reading performance, but the pattern to use is well explained and they are simple to implement.
- It is Big Data oriented, there is more administration tools to manage cluster then to manage data on single node. Cassandra CQL Shell has some problematic with mixed case names and with Dynamic Column Families (without fixed definition of columns).
- The project is under upgrade. Some new mechanism were introduced and some features will be marked as deprecated.

- An official driver for Microsoft .NET not exists, only the Thrift interface and CQL binary protocol. The third party project are good but with some lacks for what concern the complete implementation of Cassandra mechanisms. FluentCassandra, that we have used, is aligned with the last version of Cassandra, but it has some problem with memory and connection management.

7.3 Future Work

As future works we plan to replicate our tests on an in-memory Key-Value (Redis) on an machine with a large quantity of RAM, because it can be maximize the single node performance.

Also we have to test a write-oriented Document-based database (CouchDB) to see if write performance are comparable with the ones of Cassandra.

Then we plan to repeat the tests on different machines and find some mathematical functions that describe the relationship between the characteristics of used machines and the outcomes of the tests. This should be useful for theoretical estimation, before real testing.

Finally, we have to enlarge tests on the Marijà query distributed system to check if the assumption of single node independence is valid and if the final performance are comparable with a specific database distributed solution.

Appendices

Appendix A

Evaluation Framework Code Fragments

A.1 Introduction

A.2 Marijà Worker Interfaces

ISetupManager

```
public interface ISetupManager
{
    bool IsMaxR { get; set; }

    IDictionary<string, EntityMetadata> Entities { get; set; }
    IDictionary<string, RelationMetadata> Relations { get; set; }

    object GetRelId(string rel, IDictionary<string, object> ids);
    object CreateObject(string colName, object id,
        IDictionary<string, object> attrs);
    void UpdateObjects(string colName, IQuery query,
        IDictionary<string, object> toUpdate);
    void DeleteObjects(string colName, IQuery query);
    object GetCollection(string name);
    bool DatabaseExist(string name, string url = null);
    void DropCollection(string name);
    void DropDatabase();

    void Setup(IEnumerable<RelationMetadata> relations,
        params string[] entities);
    void Setup(IEnumerable<RelationMetadata> relations,
        IDictionary<string, IEnumerable<string>> entities);
    void SetupIndex(string colName, params string[] keyNames);
    void SetupEntity(string entity, IEnumerable<string> attrs = null,
        bool? drop = null);
    void SetupRelation(RelationMetadata rel,
        IDictionary<string, IEnumerable<string>> entitiesAttrs = null);

    object Start(string name, string url = null);
    object Refresh(string name = null, string url = null);
    void Init();
    void Stop();

    void CompactDB();
}
```

ICRUD, IMinRCRUD, IMaxRCRUD

```
public interface IMinRCRUD : ICRUD { }
public interface IMaxRCRUD : ICRUD { }

public interface ICRUD
{
    ISetupManager DB { get; set; }

    object CreateEntity(string col, object id,
        IDictionary<string, object> attrs);
    object DeleteEntity(string col, object id);
    IRelation ReadEntity(string col, object id);
    object UpdateEntity(string col, object id,
        IDictionary<string, object> toUpdate, bool consistent = false);

    object CreateRelation(string Rcol, IDictionary<string, object> ids);
    object DeleteRelation(string Rcol, IDictionary<string, object> ids);
    IRelation ReadRelation(string Rcol, IDictionary<string, object> ids);
    object UpdateRelation(string Rcol, IDictionary<string, object> ids,
        IDictionary<string, object> toUpdate);

    IRelation LoadEntities(string col, IQuery query = null);
    IRelation LoadRelations(string Rcol, IQuery query = null);

    IRelation Union(IRelation rel1, IRelation rel2);
    IRelation Difference(IRelation rel1, IRelation rel2);
    IRelation Product(IRelation rel1, IRelation rel2);
    IRelation Project(IRelation rel, IEnumerable<string> attrs);
    IRelation Rename(IRelation rel, string asName);
    IRelation Rename(IRelation rel, IDictionary<string, string> attrs);
    IRelation Select(IRelation rel, IQuery query);
}
```

IRelation, IRelationRow

```
public interface IRelation
{
    IList<IRelationRow> Rows { get; set; }

    void Add(IDictionary<string, object> attrs);
    IRelationRow First();
}

public interface IRelationRow : IEquatable<IRelationRow>
{
    object this[string attr] { get; }
    IDictionary<string, object> Attributes { get; set; }
    IDictionary<string, object> Keys { get; set; }
}
```

A.3 MongoDB Implementation

MongoSetupManager

```
public class MongoSetupManager : MarijaSyntethicTester.ISetupManager
```

```
{
public bool ReinitDatabase { get; set; }
public IDisposable ToDispose { get; private set; }
public MongoDatabase DB { get; private set; }

public bool IsMaxR { get; set; }

public IDictionary<string, EntityMetadata> Entities { get; set; }
public IDictionary<string, RelationMetadata> Relations { get; set; }

...

public object Start(string name, string url = null)
{
    Entities = new Dictionary<string, EntityMetadata>();
    Relations = new Dictionary<string, RelationMetadata>();

    MongoClient client;
    if (url != null) client = new MongoClient(url);
    else
    {
      client = new MongoClient(new MongoClientSettings()
      {
        MaxConnectionPoolSize = 1,
        WaitQueueSize = 1
      });
    }

    MongoServer mongo = client.GetServer();
    mongo.Connect();

    DB = mongo.GetDatabase(name);

    ToDispose = mongo.RequestStart(DB);

    return DB;
}

...

public void CompactDB()
{
    foreach (var entity in Entities.Keys)
    {
        if (DB.CollectionExists(entity))
            DB.RunCommand(new CommandDocument("compact", entity));
    }
}

...

public void SetupEntity(string entity, IEnumerable<string> attrs = null,
    bool? drop = null)
{
    if ((drop ?? ReinitDatabase) && DB.CollectionExists(entity))
    {
        DB.DropCollection(entity);
    }

    if (!DB.CollectionExists(entity))
    {
        DB.CreateCollection(entity,
            MongoDB.Driver.Builders.CollectionOptions.SetAutoIndexId(true));
    }
    else
    {
        DB.RunCommand(new CommandDocument("compact", entity));
    }

    if (!Entities.ContainsKey(entity))
    {
        Entities.Add(entity, new EntityMetadata(entity)
        {
            Collection = DB.GetCollection(entity)
        });
    }
}

public void SetupRelation(RelationMetadata rel,
    IDictionary<string, IEnumerable<string>> entitiesAttrs = null)
{
    if (!Relations.ContainsKey(rel.Name))
    {
        var col = DB.GetCollection(rel.Name);

        SetupEntity(rel.Name);

        foreach (var e in rel.Entities)
        {
            col.EnsureIndex(e + "__id");
            if (!Entities[e].Relations.Contains(rel.Name))
                Entities[e].Relations.Add(rel.Name);
        }

        rel.Collection = col;
        Relations.Add(rel.Name, rel);
    }
}

...

}
```

MongoMinRCRUD

```
public class MongoMinRCRUD : MinRCRUD, IMongoCRUD, IMinRCRUD
{
public MongoMinRCRUD(ISetupManager db)
{
    base.DB = db;
}

#region Write

/// <summary>
/// Create(A)
///
/// find(A)
/// insert(A)
/// transfer
///
/// </summary>
/// <param name="col">Collection name</param>
/// <param name="id">Entity id</param>
/// <param name="attrs">Entity attributes</param>
/// <returns>Created entity</returns>
public override object CreateEntity(string col, object id,
        IDictionary<string, object> attrs)
{
    return getE(col).AsMongoCol().Insert(DB.CreateObject(col, id, attrs));
}

/// <summary>
/// Delete(A):
///
/// a = find(A)
/// transfer
/// delete(a)
/// foreach (R in {A}) {
///    R_a = select(R, R.Aid = @Aid)
///    foreach (r in R_a) {
///         delete(r)
///    }
/// }
///
/// </summary>
/// <param name="col">Collection name</param>
/// <param name="id">Entity id</param>
/// <returns>Operation result: true</returns>
public override object DeleteEntity(string col, object id)
{
    var E = getE(col);
    foreach (var R in E.Relations)
    {
        getR(R).AsMongoCol().Remove(Query.EQ(col + "__id", id.ToString()));
    }

    return E.AsMongoCol().Remove(Query.EQ("_id", id.ToString()));
}

/// <summary>
/// Update(A):
///
/// a = find(A)
/// updateIndex(IX_A)
/// update(a)
/// transfer
///
/// </summary>
/// <param name="col">Collection name</param>
/// <param name="id">Entity id</param>
/// <param name="toUpdate">Attributes to update</param>
/// <param name="consistent">Consistent writes</param>
/// <returns>Operation result: true</returns>
public override object UpdateEntity(string col, object id,
        IDictionary<string, object> toUpdate, bool consistent = true)
{
    var E = getE(col);
    var Ecol = E.AsMongoCol();

    var writeConcern = new WriteConcern()
    {
       W = 1,
        Journal = false,
        FSync = true
    };

    return E.AsMongoCol()
        .LoggedUpdate(Query.EQ("_id", id.ToString()), Update.Combine(
                toUpdate.Select(a => Update.Set(a.Key, a.Value.ToString()))),
                writeConcern);
}

/// <summary>
/// Create(R):
```

```
///
/// foreach (X in {R}) {
///         insert(X)
/// }
/// select(R, p)
/// transfer
/// insert
/// transfer
///
/// </summary>
/// <param name="Rcol">Relation name</param>
/// <param name="ids">Relation related entities keys</param>
/// <returns>Created relation</returns>
public override object CreateRelation(string Rcol,
        IDictionary<string, object> ids)
{
    var R = getR(Rcol);
    if (ids.Count != 2 && R.Type != RelationType.Multi)
        throw new InvalidOperationException(
                "only binary relations can be created with constraints");

    var relId = DB.GetRelId(Rcol, ids);

    var e1 = ids.First();
    var e2 = ids.Last();
    switch (R.Type)
    {
        case RelationType.R_11:
            if (R.AsMongoCol()
                .Find(Query.Or(Query.EQ(e1.Key + "__id", e1.Value.ToString()),
                        Query.EQ(e2.Key + "__id", e2.Value.ToString()))).Any()
                )
            {
                throw new InvalidOperationException(
                        "Relation type contraint violation");
            }
            break;
        case RelationType.R_1N:
            if (R.AsMongoCol()
                .Find(Query.EQ(e2.Key + "__id", e2.Value.ToString())).Any())
            {
                throw new InvalidOperationException(
                        "Relation type contraint violation");
            }
            break;
        case RelationType.R_NM:
            if (R.AsMongoCol()
                .Find(Query.And(Query.EQ(e1.Key + "__id", e1.Value.ToString()),
                        Query.EQ(e2.Key + "__id", e2.Value.ToString()))
                ).Any())
            {
                throw new InvalidOperationException(
                        "Relation type contraint violation");
            }
            break;
        default:
            break;
    }

    var relation = new BsonDocument().Add("_id", relId.ToString());
    foreach (var e in R.Entities)
    {
        string id = ids.ContainsKey(e) ? ids[e].ToString() : string.Empty;
        if (string.IsNullOrEmpty(id))
            throw new System.ArgumentException(
                "ids: entity not found in relation metadata");

        if (!getE(e).AsMongoCol().Find(Query.EQ("_id", id)).Any())
            throw new InvalidOperationException(
                string.Format("Entity {0} with key {1} not found.", e, id));

        relation = relation.Add(e + "__id", id);
    }
    return R.AsMongoCol().Insert(relation as BsonDocument);
}

/// <summary>
/// Delete(R):
///
/// r = find(R)
/// transfer
/// updateIndex(IX_R)
/// delete(r)
///
/// </summary>
/// <param name="Rcol">Relation name</param>
/// <param name="ids">Relation related entities keys</param>
/// <returns>Operation result: true</returns>
public override object DeleteRelation(string Rcol,
        IDictionary<string, object> ids)
{
    object relId = base.DB.GetRelId(Rcol, ids);
    return getR(Rcol).AsMongoCol().Remove(Query.EQ("_id", relId.ToString()));
}

/// <summary>
/// Update(R):
///
/// find(R)
/// updateIndex(IX_R)
/// update(R)
/// transfer
/// </summary>
/// <param name="Rcol">Relation name</param>
/// <param name="ids">Relation related entities keys</param>
/// <param name="toUpdate">Relation attributes to update</param>
/// <returns>Operation result: true</returns>
public override object UpdateRelation(string Rcol,
        IDictionary<string, object> ids, IDictionary<string, object> toUpdate)
{
    var relId = DB.GetRelId(Rcol, ids);

    var toU = toUpdate.Where(a => !a.Key.EndsWith("_id"));
    if (toU.Count() == 0)
        return null;

    return getR(Rcol).AsMongoCol()
        .LoggedUpdate(Query.EQ("_id", relId.ToString()),
                Update.Combine(toU.Select(a => Update.Set(a.Key, a.Value.ToString()))));
}

#endregion

#region Read

/// <summary>
/// Read(A): find by key
///
/// find(A, Aid)
///
```

```
/// </summary>
/// <param name="col">Entity name</param>
/// <param name="id">Entity key</param>
/// <returns>MaxR containing the single entity (autorelation)</returns>
public override IRelation ReadEntity(string col, object id)
{
    var minr = new MinR();
    var E = getE(col);
    var doc = E.AsMongoCol().FindOneById(id.ToString());
    if (doc == null) return new MinR();
    var idString = id.ToString();

    if (DB.Relations.ContainsKey(col))
    {
        minr.Add(doc.ToRow());
    }
    else
    {
        minr.Entities.Add(E.Name, new Dictionary<string,
                IDictionary<string, object>> { { idString, doc.ToRow() } });
        minr.Add(new Dictionary<string, object> { { E.Name + "__id", idString } });
    }
    return minr;
}

/// <summary>
/// Read(R): find by key
///
/// foreach (var x in Rids) {
///     find(x.entity, x.id)
///     transfer
/// }
/// find(R, Rids)
/// transfer
///
/// </summary>
/// <param name="col">Entity name</param>
/// <param name="id">Entity key</param>
/// <returns>MaxR containing the single relation</returns>
public override IRelation ReadRelation(string Rcol, IDictionary<string, object> ids)
{
    var R = getR(Rcol);
    var entities = new Dictionary<string, IDictionary<string,
        IDictionary<string, object>>>();

    var entityKeys = new Dictionary<string, object>();
    foreach (var ename in R.Entities)
    {
        var id = ids.ContainsKey(ename) ? ids[ename].ToString() : string.Empty;
        if (!string.IsNullOrEmpty(id))
        {
            entities.Add(ename, new Dictionary<string,
                IDictionary<string, object>>
                {
                        { id, getE(ename).AsMongoCol().FindOneById(id).ToRow() }
                });
            entityKeys.Add(ename, id);
        }
    }

    var rel = R.AsMongoCol().FindOneById(DB.GetRelId(Rcol, ids).ToString())
        as BsonDocument;

    if (rel == null)
        return null;
    return new MinR
    {
        Rows = new List<IRelationRow> { new MinRRow(entities)
        {
                Attributes = rel.ToRow(), Keys = entityKeys }
        },
        Entities = entities
    };
}

/// <summary>
/// Read(A): query
///
/// select(A, p)
/// transfer
///
/// </summary>
/// <param name="col">Entity name</param>
/// <param name="query">Read predicate</param>
/// <returns>MaxR containing the entities</returns>
public override IRelation LoadEntities(string col, IQuery query = null)
{
    var minr = new MinR();
    var E = getE(col);
    minr.Entities.Add(col, new Dictionary<string,
        IDictionary<string, object>>());
    foreach (var e in E.AsMongoCol().Find(E.BuildMongoQuery(query)).ToList())
    {
        var id = e["_id"].AsString;
        minr.Entities[E.Name].Add(id, e.ToRow());
        minr.Add(new Dictionary<string, object> { { E.Name + "__id", id } });
    }
    return minr;
}

/// <summary>
/// Read(R): query
///
/// rA = select(A, p)
/// transfer
/// select(R, R.Aid IN rA.id)
/// transfer
/// foreach (var X in ({R}-A)) {
///     ids = distinct(kR)
///     foreach (id in ids) {
///             find(X, id)
///     }
///     transfer
/// }
/// join()
///
/// </summary>
/// <param name="col">Entity name</param>
/// <param name="id">Entity key</param>
/// <returns>MaxR containing the relations</returns>
public override IRelation LoadRelations(string Rcol, IQuery query = null)
{
    List<BsonDocument> Rs;
    IDictionary<string, IDictionary<string, object>> As = null;
    var entities = new Dictionary<string, IDictionary<string,
        IDictionary<string, object>>>();

    var R = getR(Rcol);
    var simple = query as SimpleQuery;
    if (simple == null)
        Rs = R.AsMongoCol().FindAll().ToList();
    else if (Rcol.Equals(simple.Entity))
        Rs = R.AsMongoCol().Find(R.BuildMongoMinRQuery(simple)).ToList();
    else
    {
        var A = getE(simple.Entity);
        As = A.AsMongoCol().Find(A.BuildMongoQuery(simple))
                .ToDictionary(r => (r as BsonDocument)["_id"].AsString,
                        r => (r as BsonDocument).ToRow());
        Rs = R.AsMongoCol().Find(Query.In(A.Name + "__id",
                        As.Keys.Select(a => BsonValue.Create(a)).ToArray()))
                .ToList<BsonDocument>();

        entities.Add(A.Name, As);
    }

    foreach (var Ecol in R.Entities)
    {
        // if it is already loaded (it is the filtered entity)
        if (simple != null && (Ecol == simple.Entity && As != null))
            continue;

        var ids = Rs.Select(r => r[Ecol + "__id"]).Distinct();
        var Es = getE(Ecol).AsMongoCol().Find(Query.In("_id", ids))
                .ToDictionary(r => (r as BsonDocument)["_id"].AsString,
                        r => (r as BsonDocument).ToRow());

        entities.Add(Ecol, Es);
    }

    var relations = new List<IRelationRow>();
    foreach (var rel in Rs)
    {
        var ents = R.Entities.ToDictionary(e => e, e => rel[e + "__id"].AsString as object);
        relations.Add(new MinRRow(entities) { Attributes = rel.ToRow(), Keys = ents });
    }
    return new MinR { Rows = relations, Entities = entities };
}

#endregion
}
```

MongoMaxRCRUD

```
public class MongoMaxRCRUD : MaxRCRUD, IMongoCRUD, IMaxRCRUD
{
public MongoMaxRCRUD(ISetupManager db)
{
    base.DB = db;
}

#region Write

/// <summary>
/// Create(A)
///
/// find(A)
/// insert(A)
/// transfer
///
/// </summary>
/// <param name="col">Collection name</param>
/// <param name="id">Entity id</param>
/// <param name="attrs">Entity attributes</param>
/// <returns>Created entity</returns>
public override object CreateEntity(string col, object id,
        IDictionary<string, object> attrs)
{
    return getE(col).AsMongoCol().Insert<object>(DB.CreateObject(col, id, attrs));
}

/// <summary>
/// Delete(A):
///
/// a = find(A)
/// transfer
/// delete(a)
/// foreach (R in {A}) {
///     R_a = select(R, R.Aid = @Aid)
///     foreach (r in R_a) {
///         delete(r)
///     }
/// }
///
/// </summary>
/// <param name="col">Collection name</param>
/// <param name="id">Entity id</param>
/// <returns>Operation result: true</returns>
public override object DeleteEntity(string col, object id)
{
    var e = getE(col);
    foreach (string current in e.Relations)
    {
        getR(current).AsMongoCol()
                .Remove(Query.EQ(col + "__id", id.ToString()));
    }
    return e.AsMongoCol().Remove(Query.EQ("_id", id.ToString()));
}

/// <summary>
/// Update(A):
///
/// a = find(A)
/// updateIndex(IX_A)
/// update(a)
/// transfer
/// R_a = select(R, R.Aid = @Aid)
/// foreach (R in R_a) {
///         update(R)
///         transfer
/// }
///
/// </summary>
/// <param name="col">Collection name</param>
/// <param name="id">Entity id</param>
/// <param name="toUpdate">Attributes to update</param>
/// <param name="consistent">Consistent writes</param>
/// <returns>Operation result: true</returns>
public override object UpdateEntity(string col, object id,
        IDictionary<string, object> toUpdate, bool consistent = true)
{
    var writeConcern = new WriteConcern()
    {
        W = 1,
        Journal = false,
        FSync = true
    };

    var E = getE(col);
    var result = E.AsMongoCol()
        .LoggedUpdate(Query.EQ("_id", id.ToString()), Update.Combine(
                toUpdate.Select(a => Update.Set(a.Key, a.Value.ToString()))),
                writeConcern);

    if (!consistent)
        writeConcern = WriteConcern.Acknowledged;

    foreach (string R in E.Relations)
    {
        getR(R).AsMongoCol()
                .LoggedUpdate(Query.EQ(col + "__id", id.ToString()),
                        Update.Combine(toUpdate.Select(a =>
                                Update.Set(col + "_" + a.Key, a.Value.ToString())
                        )), UpdateFlags.Multi);
    }
    return result;
}

/// <summary>
/// Create(R):
///
/// foreach (X in {R}) {
///         insert(X)
/// }
/// select(R, p)
/// transfer
/// insert
/// transfer
///
/// </summary>
/// <param name="Rcol">Relation name</param>
/// <param name="ids">Relation related entities keys</param>
/// <returns>Created relation</returns>
public override object CreateRelation(string Rcol,
        IDictionary<string, object> ids)
{
    var R = getR(Rcol);
    if (ids.Count != 2 && R.Type != RelationType.Multi)
        throw new InvalidOperationException(
                "only_binary_relations_can_be_created_with_constraints");

    var relId = DB.GetRelId(Rcol, ids);

    var e1 = ids.First();
    var e2 = ids.Last();
    switch (R.Type)
    {
        case RelationType.R_11:
            if (R.AsMongoCol()
                .Find(Query.Or(Query.EQ(e1.Key + "__id", e1.Value.ToString()),
                        Query.EQ(e2.Key + "__id", e2.Value.ToString()))).Any()
                )
            {
                throw new InvalidOperationException(
                        "Relation_type_contraint_violation");
            }
            break;
        case RelationType.R_1N:
            if (R.AsMongoCol()
                .Find(Query.EQ(e2.Key + "__id", e2.Value.ToString())).Any())
            {
                throw new InvalidOperationException(
                        "Relation_type_contraint_violation");
            }
```

```
                break;
            case RelationType.R_NM:
                if (R.AsMongoCol()
                    .Find(Query.And(Query.EQ(e1.Key + "__id", e1.Value.ToString()),
                            Query.EQ(e2.Key + "__id", e2.Value.ToString()))
                    ).Any())
                {
                    throw new InvalidOperationException(
                            "Relation type contraint violation");
                }
                break;
            default:
                break;
        }

        var relation = new BsonDocument().Add("_id", relId.ToString());
        foreach (var e in R.Entities)
        {
            string id = ids.ContainsKey(e) ? ids[e].ToString() : string.Empty;
            if (string.IsNullOrEmpty(id))
                throw new System.ArgumentException(
                    "ids: entity not found in relation metadata");

            var doc = getE(e).AsMongoCol().FindOneById(id);
            if (doc == null)
                throw new InvalidOperationException(
                    string.Format("Entity {0} with key {1} not found.", e, id));

            foreach (var attr in doc)
            {
                relation = relation.Add(e + "_" + attr.Name, attr.Value);
            }
        }
        return R.AsMongoCol().Insert(relation as BsonDocument);
}

/// <summary>
/// Delete(R):
///
/// r = find(R)
/// transfer
/// updateIndex(IX_R)
/// delete(r)
///
/// </summary>
/// <param name="Rcol">Relation name</param>
/// <param name="ids">Relation related entities keys</param>
/// <returns>Operation result: true</returns>
public override object DeleteRelation(string Rcol,
        IDictionary<string, object> ids)
{
    var relId = DB.GetRelId(Rcol, ids);
    return getR(Rcol).AsMongoCol().Remove(Query.EQ("_id", relId.ToString()));
}

/// <summary>
/// Update(R):
///
/// find(R)
/// updateIndex(IX_R)
/// update(R)
/// transfer
/// </summary>
/// <param name="Rcol">Relation name</param>
/// <param name="ids">Relation related entities keys</param>
/// <param name="toUpdate">Relation attributes to update</param>
/// <returns>Operation result: true</returns>
public override object UpdateRelation(string Rcol,
        IDictionary<string, object> ids, IDictionary<string, object> toUpdate)
{
    var relId = DB.GetRelId(Rcol, ids);

    var toU = toUpdate.Where(a => !a.Key.EndsWith("_id"));
    if (toU.Count() == 0)
        return null;

    return getR(Rcol).AsMongoCol()
        .LoggedUpdate(Query.EQ("_id", relId.ToString()), Update.Combine(
                toU.Select(a => Update.Set(a.Key, a.Value.ToString()))));
}

#endregion

#region Read

/// <summary>
/// Read(A): find by key
///
/// find(A, Aid)
///
/// </summary>
/// <param name="col">Entity name</param>
/// <param name="id">Entity key</param>
/// <returns>MaxR containing the single entity (autorelation)</returns>
public override IRelation ReadEntity(string col, object id)
{
    var doc = getE(col).AsMongoCol().FindOneById(id.ToString());
    if (doc == null) return new MaxR();
    var maxr = new MaxR(col);
    var prefix = string.Empty;
    if (!DB.Relations.ContainsKey(col)) prefix = col + "_";
    maxr.Add(doc.ToRow(prefix));
    return maxr;
}

/// <summary>
/// Read(R): find by key
///
/// find(R, Rids)
/// transfer
///
/// </summary>
/// <param name="col">Entity name</param>
/// <param name="id">Entity key</param>
/// <returns>MaxR containing the single relation</returns>
public override IRelation ReadRelation(string Rcol,
        IDictionary<string, object> ids)
{
    var R = getR(Rcol);
    var entityKeys = new Dictionary<string, object>();
    var rel = R.AsMongoCol()
        .FindOneById(DB.GetRelId(Rcol, ids).ToString()) as BsonDocument;

    if (rel == null)
        return null;

    foreach (var ename in R.Entities)
    {
        var id = ids.ContainsKey(ename) ? ids[ename].ToString() : string.Empty;
        entityKeys.Add(ename, id);
    }
    return new MaxR(R.Entities) { Rows = new List<IRelationRow> {
        new MaxRRow { Attributes = rel.ToRow(), Keys = entityKeys } } };
}

/// <summary>
/// Read(A): query
///
/// select(A, p)
/// transfer
///
/// </summary>
/// <param name="col">Entity name</param>
/// <param name="query">Read predicate</param>
/// <returns>MaxR containing the entities</returns>
public override IRelation LoadEntities(string col, IQuery query = null)
{
    var maxr = new MaxR(col);
    var E = getE(col);
    foreach (var e in E.AsMongoCol().Find(E.BuildMongoQuery(query)).ToList())
    {
        maxr.Add(e.ToRow(E.Name + "_"));
    }
    return maxr;
}

/// <summary>
/// Read(R): query
///
/// rA = select(A, p)
/// transfer
/// select(R, R.Aid IN rA.id)
/// transfer
/// foreach (var X in ({R}-A)) {
///     ids = distinct(kR)
///     foreach (id in ids) {
///             find(X, id)
///     }
///     transfer
/// }
/// join()
///
/// </summary>
/// <param name="col">Entity name</param>
/// <param name="id">Entity key</param>
/// <returns>MaxR containing the relations</returns>
public override IRelation LoadRelations(string Rcol, IQuery query = null)
{
    List<BsonDocument> Rs;

    var R = getR(Rcol);
    if (query == null)
        Rs = R.AsMongoCol().FindAll().ToList();
    else
        Rs = R.AsMongoCol().Find(R.BuildMongoMaxRQuery(query)).ToList();

    var relations = new List<IRelationRow>();
    foreach (var rel in Rs)
    {
        var ents = R.Entities
                .ToDictionary(e => e, e => rel[e + "__id"].AsString as object);

        relations.Add(new MaxRRow { Attributes = rel.ToRow(), Keys = ents });
    }
    return new MaxR(R.Entities) { Rows = relations };
```

```
}

#endregion
}
```

A.4 Apache Cassandra Implementation

CassandraSetupManager

```
public class CassandraSetupManager : MarijaSyntethicTester.ISetupManager
{
internal string CentralIndexName = CentralIndexName;

public string Keyspace { get; set; }

public bool ReinitDatabase { get; set; }
public IDisposable ToDispose { get; private set; }
public CassandraContext DB { get; private set; }

public bool IsMaxR { get; set; }

public IDictionary<string, EntityMetadata> Entities { get; set; }
public IDictionary<string, RelationMetadata> Relations { get; set; }

...

public object Start(string name, string url = null)
{
  Keyspace = name.ToLower();
  Entities = new Dictionary<string, EntityMetadata>();
  Relations = new Dictionary<string, RelationMetadata>();

  return Refresh(name, url);
}

public object Refresh(string name = null, string url = null)
{
  try
  {
    if (DB != null && !DB.WasDisposed)
      Stop();
  }
  catch { }

  DB = new CassandraContext(new ConnectionBuilder(
    keyspace: (name ?? Keyspace).ToLower(),
    host: url ?? "localhost",
    connectionTimeout: 24*60*60,
    pooling: true,
    serverPollingInterval: 100,
    connectionLifetime: 24*60*60,
    maxPoolSize: 1000000));

  foreach (var e in Entities.Values)
    e.Collection = DB.GetColumnFamily(e.Name);

  foreach (var r in Relations.Values)
    r.Collection = DB.GetColumnFamily(r.Name);

  ToDispose = DB;

  return DB;
}
```

```
...
private void setupCentralIndexTable()
{
  if (ReinitDatabase && DB.ColumnFamilyExists(CentralIndexName))
    DB.DropColumnFamily(CentralIndexName);

  if (!DB.ColumnFamilyExists(CentralIndexName))
  {
    var cf = new CassandraColumnFamilySchema()
    {
      FamilyName = CentralIndexName,
      FamilyType = ColumnType.Super,
      KeyValueType = CassandraType.AsciiType,
      SuperColumnNameType = CassandraType.AsciiType,
      ColumnNameType = CassandraType.AsciiType,
      DefaultColumnValueType = CassandraType.AsciiType
    };

    DB.Keyspace.TryCreateColumnFamily(cf);
  }
}

public void SetupEntity(string entity, IEnumerable<string> attrs = null,
  bool? drop = null)
{
  if ((drop ?? ReinitDatabase) && DB.ColumnFamilyExists(entity))
    DB.DropColumnFamily(entity);

  if (!DB.ColumnFamilyExists(entity))
  {
    var cf = new CfDef()
    {
      Keyspace = Keyspace,
      Name = entity,
      Column_type = "Standard",
      Comparator_type = "AsciiType",
      Default_validation_class = "AsciiType",
    };

    cf.Column_metadata = new List<ColumnDef>()
    {
      new ColumnDef() { Name = ASCIIEncoding.ASCII.GetBytes("idxall"),
        Validation_class = "AsciiType", Index_type = IndexType.KEYS }
    };
    if (attrs != null)
    {
      foreach (var attr in attrs)
      {
        cf.Column_metadata.Add(new ColumnDef() {
          Name = ASCIIEncoding.ASCII.GetBytes(attr),
          Validation_class = "AsciiType"
        });
      }
    }

    DB.AddColumnFamily(cf);
    //DB.Keyspace.TryCreateColumnFamily(cf);
  }
  //else DB.RunCommand(new CommandDocument("compact", entity));

  if (!Entities.ContainsKey(entity))
#if LogCollection
    Entities.Add(entity, new EntityMetadata(entity) {
      Collection = new LogMongoCollection(DB,
        DB.GetColumnFamily(entity).Settings
          as MongoCollectionSettings<BsonDocument>)
    });
#else
    Entities.Add(entity, new EntityMetadata(entity) {
      Collection = DB.GetColumnFamily(entity)
    });
#endif
}

public void SetupRelation(RelationMetadata rel,
  IDictionary<string, IEnumerable<string>> entitiesAttrs = null)
{
  if (!Relations.ContainsKey(rel.Name))
  {
#if LogCollection
    var col = new LogMongoCollection(DB,
      DB.GetCollection(rel.Name).Settings
        as MongoCollectionSettings<BsonDocument>);
#else
    var col = DB.GetColumnFamily(rel.Name);
#endif

    var attrs = new List<string>();

    foreach (var e in rel.Entities)
    {
      attrs.Add(e + "__id");
      if (IsMaxR)
      {
        if (entitiesAttrs != null && entitiesAttrs.ContainsKey(e))
          attrs.AddRange(entitiesAttrs[e].Select(a => e + "_" + a));
      }
      if (!Entities[e].Relations.Contains(rel.Name))
        Entities[e].Relations.Add(rel.Name);
    }

    SetupEntity(rel.Name, attrs);

    rel.Collection = col;
    Relations.Add(rel.Name, rel);
  }
}

...

public IEnumerable<CassandraObject> EKeys(string entity, string relation,
  params CassandraObject[] entityKeys)
{
  if (entityKeys.Length == 0)
    return new CassandraObject[0];

  try
  {
    var r = GetCentralIndex()
      .Get(entity + "_" + relation)
      .FetchColumns(entityKeys);
    return r.SelectMany(s => s.Columns
      .SelectMany(k => k.Select(c => c.ColumnName))
    ).ToList().Distinct();
  }
  catch
  {
    return new CassandraObject[0];
  }
}

public void AddToEKeys(string entity, string relation,
  CassandraObject entityId, CassandraObject relationId)
{
  GetCentralIndex()
    .InsertColumn(entity + "_" + relation, entityId,relationId, new Byte());
}

public void RemoveFromEKeys(string entity, string relation,
  CassandraObject entityId, CassandraObject relationId)
{
  try
  {
    GetCentralIndex()
      .RemoveColumn(entity + "_" + relation, entityId, relationId);
  }
  catch
  {
  }
}

...

public IEnumerable<object> ExecuteCassandraQuery(CassandraColumnFamily cf,
  IQuery query)
{
  var simple = query as SimpleQuery;
  if (simple != null)
  {
    var attr = simple.Attr;
    if (attr == "_id")
    {
      // Primary keys predicate

      switch (simple.Type)
      {
        case QueryType.EQ:
          return cf.Get(simple.Value).ToList();

        case QueryType.NE:
          return cf.Get().ToList().Where(r =>
            r.Key.GetValue<AsciiType>().ToString() != simple.Value);

        case QueryType.GT:
          return cf.Get().StartWithKey(simple.Value).ToList()
            .SkipWhile(r =>
              r.Key.GetValue<AsciiType>().ToString() == simple.Value);

        case QueryType.GTE:
          return cf.Get().StartWithKey(simple.Value).ToList();

        case QueryType.LT:
          return cf.Get().TakeUntilKey(simple.Value).ToList();

        case QueryType.LTE:
          return cf.Get().TakeUntilKey(simple.Value).ToList()
            .TakeWhile(r =>
              r.Key.GetValue<AsciiType>().ToString() != simple.Value);

        default: return new FluentColumnFamily[0];
      }
    }
    else if (attr == simple.Entity + "__id")
```

```
{
  // Entity Ids case: we have to query EKeys

  IEnumerable<CassandraObject> keys;
  switch (simple.Type)
  {
    case QueryType.EQ:
      keys = EKeys(simple.Entity, cf.FamilyName, simple.Value);
      break;

    case QueryType.NE:
      keys = GetCentralIndex()
        .Get(simple.Entity + "_" + cf.FamilyName)
        .ToList().First()
        .Columns.Where(s => s.ColumnName != simple.Value)
        .SelectMany(s => s.Columns.Select(c => c.ColumnName))
        .Distinct();
      break;

    case QueryType.GT:
      keys = GetCentralIndex()
        .Get(simple.Entity + "_" + cf.FamilyName)
        .TakeUntilColumn(simple.Value).ToList()
        .SelectMany(s => s.Columns.Select(c => c.ColumnName))
        .Distinct();
      break;

    case QueryType.GTE:
      keys = GetCentralIndex()
        .Get(simple.Entity + "_" + cf.FamilyName)
        .TakeUntilColumn(simple.Value).ToList()
        .SelectMany(s => s.Columns.Select(c => c.ColumnName))
        .Distinct();
      break;

    case QueryType.LT:
      keys = GetCentralIndex()
        .Get(simple.Entity + "_" + cf.FamilyName)
        .TakeUntilColumn(simple.Value).ToList()
        .SelectMany(s => s.Columns.Select(c => c.ColumnName))
        .Distinct();
      break;

    case QueryType.LTE:
      keys = GetCentralIndex()
        .Get(simple.Entity + "_" + cf.FamilyName)
        .TakeUntilColumn(simple.Value).ToList()
        .SelectMany(s => s.Columns.Select(c => c.ColumnName))
        .Distinct();
      break;

    default: return new FluentColumnFamily[0];
  }
  if (keys.Count() == 0) return new FluentColumnFamily[0];
  return cf.Get(keys.ToArray()).ToList();
}

// Simple attributes case: Scan

Func<FluentColumnFamily, bool> expr = null;

switch (simple.Type)
{
  case QueryType.EQ:
    expr = f => f.Columns.Count != 0 && f[attr] != null &&
            f[attr].GetValue<AsciiType>().ToString()
              .CompareTo(simple.Value) == 0;
          break;

        case QueryType.NE:
          expr = f => f.Columns.Count != 0 && f[attr] != null &&
            f[attr].GetValue<AsciiType>().ToString()
              .CompareTo(simple.Value) != 0;
          break;

        case QueryType.GT:
          expr = f => f.Columns.Count != 0 && f[attr] != null &&
            f[attr].GetValue<AsciiType>().ToString()
              .CompareTo(simple.Value) > 0;
          break;

        case QueryType.GTE:
          expr = f => f.Columns.Count != 0 && f[attr] != null &&
            f[attr].GetValue<AsciiType>().ToString()
              .CompareTo(simple.Value) >= 0;
          break;

        case QueryType.LT:
          expr = f => f.Columns.Count != 0 && f[attr] != null &&
            f[attr].GetValue<AsciiType>().ToString()
              .CompareTo(simple.Value) < 0;
          break;

        case QueryType.LTE:
          expr = f => f.Columns.Count != 0 && f[attr] != null &&
            f[attr].GetValue<AsciiType>().ToString()
              .CompareTo(simple.Value) <= 0;
          break;

        default: return new FluentColumnFamily[0];
      }

      return cf.LoggedAndPagedExecuteQuery(expr);
    }

    // Fixed Predicates

    var fixedQ = query as FixedQuery;
    if (fixedQ != null)
    {
      switch (fixedQ.Type)
      {
        case FixedQueryType.ALL:
          return DB.ExecuteQuery("SELECT * FROM " + fixedQ.Name);
        case FixedQueryType.NONE:
          return new FluentColumnFamily[0];
        default: return new FluentColumnFamily[0];
      }
    }
    return new FluentColumnFamily[0];
}
}
```

CassandraMinRCRUD

```
public class CassandraMinRCRUD : MinRCRUD, ICassandraCRUD, IMinRCRUD, ICRUD
{
public CassandraSetupManager CassandraDB {
  get { return DB as CassandraSetupManager; } }
```

```
public CassandraMinRCRUD(ISetupManager db)
{
  DB = db;
}

#region Write

/// <summary>
/// Create(A)
///
/// find(A)
/// insert(A)
/// transfer
///
/// </summary>
/// <param name="col">Collection name</param>
/// <param name="id">Entity id</param>
/// <param name="attrs">Entity attributes</param>
/// <returns>Created entity</returns>
public override object CreateEntity(string col, object id,
  IDictionary<string, object> attrs)
{
  var e = getE(col).AsColumnFamily().CreateRecord(id.ToString());
  foreach (var attr in attrs)
    e[attr.Key] = attr.Value.ToString();

  e["idxall"] = 0;

  CassandraDB.DB.Attach(e);
  CassandraDB.DB.SaveChanges();

  return e;
}

/// <summary>
/// Delete(A):
///
/// a = find(A)
/// transfer
/// delete(a)
/// foreach (R in {A}) {
///   R_a = select(R, R.Aid = @Aid)
///   foreach (r in R_a) {
///       delete(r)
///   }
/// }
///
/// </summary>
/// <param name="col">Collection name</param>
/// <param name="id">Entity id</param>
/// <returns>Operation result: true</returns>
public override object DeleteEntity(string col, object id)
{
  var Ecol = getE(col);
  foreach (var R in Ecol.Relations)
  {
    var r = getR(R).AsColumnFamily();
    var keys = CassandraDB.EKeys(col, R, id.ToString());
    foreach (var key in keys)
    {
      r.RemoveKey(key);
    }
    CassandraDB.RemoveFromEKeys(col, R, id.ToString());
  }
  Ecol.AsColumnFamily().RemoveKey(id.ToString());
  return true;
}

/// <summary>
/// Update(A):
///
/// a = find(A)
/// updateIndex(IX_A)
/// update(a)
/// transfer
///
/// </summary>
/// <param name="col">Collection name</param>
/// <param name="id">Entity id</param>
/// <param name="toUpdate">Attributes to update</param>
/// <param name="consistent">Consistent writes</param>
/// <returns>Operation result: true</returns>
public override object UpdateEntity(string col, object id,
  IDictionary<string, object> toUpdate, bool consistent = true)
{
  var E = getE(col);
  var Ecol = E.AsColumnFamily();
  var rec = Ecol.Get(id.ToString()).First();

  update(rec, toUpdate);

  Ecol.LoggedUpdate(rec);
  //CassandraDB.DB.Attach(rec);
  //CassandraDB.DB.LoggedSaveChanges(col);

  return true;
}

/// <summary>
/// Create(R):
///
/// foreach (X in {R}) {
///         insert(X)
/// }
/// select(R, p)
/// transfer
/// insert
/// transfer
///
/// </summary>
/// <param name="Rcol">Relation name</param>
/// <param name="ids">Relation related entities keys</param>
/// <returns>Created relation</returns>
public override object CreateRelation(string Rcol,
  IDictionary<string, object> ids)
{
  var R = getR(Rcol);

  if (ids.Count != 2 && R.Type != RelationType.Multi)
    throw new InvalidOperationException(
      "only binary relations can be created with constraints");

  var id = DB.GetRelId(Rcol, ids);

  var e1 = ids.First();
  var e2 = ids.Last();
  switch (R.Type)
  {
    case RelationType.R_11:
        if (CassandraDB.EKeys(e1.Key, Rcol, e1.Value.ToString()).Any() ||
          CassandraDB.EKeys(e2.Key, Rcol, e2.Value.ToString()).Any())
          throw new InvalidOperationException(
            "Relation␣type␣constraint␣violation");
        break;
      case RelationType.R_1N:
        if (CassandraDB.EKeys(e2.Key, Rcol, e2.Value.ToString()).Any())
          throw new InvalidOperationException(
            "Relation␣type␣constraint␣violation");
        break;
      case RelationType.R_NM:
        if (R.AsColumnFamily().Get(id.ToString()).CountColumns() != 0)
          throw new InvalidOperationException(
            "Relation␣type␣constraint␣violation");
        break;
      default:
        break;
    }

    var relation = R.AsColumnFamily().CreateRecord(id.ToString());
    foreach (var e in R.Entities)
    {
      var eId = ids.ContainsKey(e) ? ids[e].ToString() : string.Empty;
      if (string.IsNullOrEmpty(eId))
        throw new ArgumentException(
          "ids:␣entity␣not␣found␣in␣relation␣metadata");

      var eDoc = getE(e).AsColumnFamily().Get(eId.ToString()).First();
      if (eDoc.Columns.Count == 0)
        throw new InvalidOperationException(string.Format(
          "Entity␣{0}␣with␣key␣{1}␣not␣found.", e, eId));

      relation[e + "__id"] = eDoc.Key;

      CassandraDB.AddToEKeys(e, Rcol, eId, id.ToString());
    }

    relation["idxall"] = 0;

    CassandraDB.DB.Attach(relation);
    CassandraDB.DB.SaveChanges();

    return relation;
  }

/// <summary>
/// Delete(R):
///
/// r = find(R)
/// transfer
/// updateIndex(IX_R)
/// delete(r)
///
/// </summary>
/// <param name="Rcol">Relation name</param>
/// <param name="ids">Relation related entities keys</param>
/// <returns>Operation result: true</returns>
public override object DeleteRelation(string Rcol,
  IDictionary<string, object> ids)
{
  var id = DB.GetRelId(Rcol, ids);
  getR(Rcol).AsColumnFamily().RemoveKey(id.ToString());

  foreach (var E in ids)
    CassandraDB.RemoveFromEKeys(E.Key, Rcol, E.Value.ToString(),
        id.ToString());

    return true;
}

/// <summary>
/// Update(R):
///
/// find(R)
/// updateIndex(IX_R)
/// update(R)
/// transfer
/// </summary>
/// <param name="Rcol">Relation name</param>
/// <param name="ids">Relation related entities keys</param>
/// <param name="toUpdate">Relation attributes to update</param>
/// <returns>Operation result: true</returns>
public override object UpdateRelation(string Rcol,
    IDictionary<string, object> ids, IDictionary<string, object> toUpdate)
{
    var id = DB.GetRelId(Rcol, ids);

    var rec = getR(Rcol).AsColumnFamily().Get(id.ToString()).First();
    if (rec.Columns.Count == 0)
        return false;

    var toU = toUpdate.Where(a => !a.Key.EndsWith("_id"));
    if (toU.Count() == 0)
        return true;

    update(rec, toU);

    CassandraDB.DB.Attach(rec);
    CassandraDB.DB.SaveChanges();

    return true;
}

#endregion

#region Read

/// <summary>
/// Read(A): find by key
///
/// find(A, Aid)
///
/// </summary>
/// <param name="col">Entity name</param>
/// <param name="id">Entity key</param>
/// <returns>MaxR containing the single entity (autorelation)</returns>
public override IRelation ReadEntity(string col, object id)
{
    var minr = new MinR();
    var E = getE(col);
    var doc = E.AsColumnFamily().GetSingle(id.ToString(), null, null);
    if (doc.Columns.Count == 0) return new MinR();
    var idString = id.ToString();

    if (DB.Relations.ContainsKey(col))
    {
        minr.Add(doc.ToRow());
    }
    else
    {
```

```
    minr.Entities.Add(E.Name,
      new Dictionary<string, IDictionary<string, object>> {
        { idString, doc.ToRow() }
      });
    minr.Add(new Dictionary<string, object> { { E.Name + "__id", idString } });
  }
  return minr;
}

/// <summary>
/// Read(R): find by key
///
/// foreach (var x in Rids) {
///     find(x.entity, x.id)
///     transfer
/// }
/// find(R, Rids)
/// transfer
///
/// </summary>
/// <param name="col">Entity name</param>
/// <param name="id">Entity key</param>
/// <returns>MaxR containing the single relation</returns>
public override IRelation ReadRelation(string Rcol,
  IDictionary<string, object> ids)
{
  var R = getR(Rcol);
  var entities = new Dictionary<string,
    IDictionary<string, IDictionary<string, object>>>();

  var entityKeys = new Dictionary<string, object>();
  foreach (var ename in R.Entities)
  {
    var id = ids.ContainsKey(ename) ? ids[ename].ToString() : string.Empty;
    if (!string.IsNullOrEmpty(id))
      entities.Add(ename,
        new Dictionary<string, IDictionary<string, object>> {
        {
          id,
          getE(ename).AsColumnFamily()
            .GetSingle(id, null, null).ToRow()
        } });
    entityKeys.Add(ename, id);
  }
  var rel = R.AsColumnFamily()
    .GetSingle(DB.GetRelId(Rcol, ids).ToString(), null, null);
  if (rel == null) return null;
  return new MinR
  {
    Rows = new List<IRelationRow> { new MinRRow(entities) {
      Attributes = rel.ToRow(), Keys = entityKeys }
    },
    Entities = entities
  };
}

/// <summary>
/// Read(A): query
///
/// select(A, p)
/// transfer
///
/// </summary>
/// <param name="col">Entity name</param>
/// <param name="query">Read predicate</param>
/// <returns>MaxR containing the entities</returns>
public override IRelation LoadEntities(string col, IQuery query = null)
{
  var minr = new MinR();
  var E = getE(col);
  minr.Entities.Add(col, new Dictionary<string, IDictionary<string, object>>());

  IEnumerable Es;

  if (query == null)
    Es = E.ExecuteCassandraQuery(FixedQuery.All(col), CassandraDB);
  else
    Es = E.ExecuteCassandraQuery(query, CassandraDB);

  foreach (var row in Es)
  {
    var e = new CassandraRow(row);
    var id = e.Key.GetValue<AsciiType>().ToString();
    minr.Entities[E.Name].Add(id, e.ToRow());
    minr.Add(new Dictionary<string, object> { { E.Name + "__id", id } });
  }
  return minr;
}

/// <summary>
/// Read(R): query
///
/// rA = select(A, p)
/// transfer
/// select(R, R.Aid IN rA.id)
/// transfer
/// foreach (var X in ({R}-A)) {
///     ids = distinct(kR)
///     foreach (id in ids) {
///             find(X, id)
///     }
///     transfer
/// }
/// join()
///
/// </summary>
/// <param name="col">Entity name</param>
/// <param name="id">Entity key</param>
/// <returns>MaxR containing the relations</returns>
public override IRelation LoadRelations(string Rcol, IQuery query = null)
{
  IEnumerable<object> Rs, As = null;
  //IDictionary<string, IDictionary<string, object>> As = null;
  var entities = new Dictionary<string,
    IDictionary<string, IDictionary<string, object>>>();

  var R = getR(Rcol);
  var simple = query as SimpleQuery;
  if (simple == null)
  {
    Rs = R.ExecuteCassandraMinRQuery(query ?? FixedQuery.All(Rcol),
      CassandraDB);
  }
  else if (Rcol.Equals(simple.Entity))
    Rs = R.ExecuteCassandraMinRQuery(query, CassandraDB);
  else
  {
    As = getE(simple.Entity).ExecuteCassandraQuery(query, CassandraDB);
    CassandraObject[] keys = new CassandraObject[0];
    if (As.Count() != 0)
    {
      keys = CassandraDB.EKeys(simple.Entity, Rcol, As.ToCassandraKeys())
        .ToArray();
    }
    Rs = LogExtensions.Log<CassandraColumnFamily, IEnumerable<object>>(
      R.AsColumnFamily(), source =>
      {
        if (keys.Length == 0)
          return new object[0];

        //return source.Get(keys).ToList();
        return source.PagedGetKeys(keys).ToList();
      }, new ExpectedCassandraStatistics(Rcol), ret => keys.Length);

    if (As.Count() != 0)
      entities.Add(simple.Entity, As.ToCassandraDictionary());
  }

  foreach (var Ecol in R.Entities)
  {
    // se l'ho già caricata prima (era quella da filtrare)
    if (simple != null && (Ecol == simple.Entity && As != null))
      continue;

    var Es = LogExtensions.Log<CassandraColumnFamily,
      IEnumerable<FluentColumnFamily>>(getE(Ecol).AsColumnFamily(),
      source =>
      {
        if (Rs.Count() == 0)
          return new FluentColumnFamily[0];

        var ids = Rs.ToCassandraRow().Select(r => r[Ecol + "__id"])
          .Distinct();

        return source.PagedGetKeys(ids.ToArray());
      }, new ExpectedCassandraStatistics(Ecol), ret => ret.Count());

    if (Es.Count() != 0)
      entities.Add(Ecol, Es.ToCassandraDictionary());
  }

  if (Rs.Count() == 0) return new MinR();

  var relations = new List<IRelationRow>();
  foreach (var row in Rs)
  {
    var rel = new CassandraRow(row);
    var ents = R.Entities
      .ToDictionary(e => e, e => rel[e + "__id"].GetValue());
    relations.Add(new MinRRow(entities) {
      Attributes = rel.ToRow(),
      Keys = ents
    });
  }
  return new MinR { Rows = relations, Entities = entities };
}

#endregion

#region privates

private static void update(FluentColumnFamily rec,
  IEnumerable<KeyValuePair<string, object>> toUpdate)
{
  foreach (var attr in toUpdate)
```

```
    rec[attr.Key] = attr.Value.ToString();
}

#endregion
}
```

CassandraMaxRCRUD

```
public class CassandraMaxRCRUD : MaxRCRUD, ICassandraCRUD, IMaxRCRUD, ICRUD
{
public CassandraSetupManager CassandraDB {
  get { return DB as CassandraSetupManager; }
}

public CassandraMaxRCRUD(ISetupManager db)
{
  DB = db;
}

#region Write

/// <summary>
/// Create(A)
///
/// find(A)
/// insert(A)
/// transfer
///
/// </summary>
/// <param name="col">Collection name</param>
/// <param name="id">Entity id</param>
/// <param name="attrs">Entity attributes</param>
/// <returns>Created entity</returns>
public override object CreateEntity(string col, object id,
  IDictionary<string, object> attrs)
{
  var e = getE(col).AsColumnFamily().CreateRecord(id.ToString());
  foreach (var attr in attrs)
    e[attr.Key] = attr.Value.ToString();

  e["idxall"] = 0;

  CassandraDB.DB.Attach(e);
  CassandraDB.DB.SaveChanges();

  return e;
}

/// <summary>
/// Delete(A):
///
/// a = find(A)
/// transfer
/// delete(a)
/// foreach (R in {A}) {
///    R_a = select(R, R.Aid = @Aid)
///    foreach (r in R_a) {
///         delete(r)
///    }
/// }
///
/// </summary>
/// <param name="col">Collection name</param>
/// <param name="id">Entity id</param>
/// <returns>Operation result: true</returns>
```

```
public override object DeleteEntity(string col, object id)
{
  var Ecol = getE(col);
  foreach (var R in Ecol.Relations)
  {
    var r = getR(R).AsColumnFamily();
    var keys = CassandraDB.EKeys(col, R, id.ToString());
    foreach (var key in keys)
    {
      r.RemoveKey(key);
    }
    CassandraDB.RemoveFromEKeys(col, R, id.ToString());
  }

  Ecol.AsColumnFamily().RemoveKey(id.ToString());
  return true;
}

/// <summary>
/// Update(A):
///
/// a = find(A)
/// updateIndex(IX_A)
/// update(a)
/// transfer
/// R_a = select(R, R.Aid = @Aid)
/// foreach (R in R_a) {
///         update(R)
///         transfer
/// }
///
/// </summary>
/// <param name="col">Collection name</param>
/// <param name="id">Entity id</param>
/// <param name="toUpdate">Attributes to update</param>
/// <param name="consistent">Consistent writes</param>
/// <returns>Operation result: true</returns>
public override object UpdateEntity(string col, object id,
  IDictionary<string, object> toUpdate, bool consistent = true)
{
  var E = getE(col);
  var Ecol = E.AsColumnFamily();
  var rec = Ecol.Get(id.ToString()).First();

  update(rec, toUpdate);

  Ecol.LoggedUpdate(rec);
  //CassandraDB.DB.Attach(rec);
  //CassandraDB.DB.LoggedSaveChanges(col);

  foreach (var R in E.Relations)
  {
    var Rcol = getR(R).AsColumnFamily();
    var keys = CassandraDB.EKeys(col, R, id.ToString()).ToArray();
    return LogExtensions
      .Log<CassandraColumnFamily, bool>(Rcol, source =>
      {
        if (keys.Length == 0)
          return true;

        foreach (var rel in source.PagedGetKeys(keys))
        {
          update(rel, toUpdate
            .ToDictionary(u => col + "_" + u.Key, u => u.Value));
        CassandraDB.DB.Attach(rel);
      }

      CassandraDB.DB.SaveChanges();
      return true;
    }, new ExpectedCassandraStatistics(R), ret => keys.Length);
  }

  return true;
}

/// <summary>
/// Create(R):
///
/// foreach (X in {R}) {
///         insert(X)
/// }
/// select(R, p)
/// transfer
/// insert
/// transfer
///
/// </summary>
/// <param name="Rcol">Relation name</param>
/// <param name="ids">Relation related entities keys</param>
/// <returns>Created relation</returns>
public override object CreateRelation(string Rcol,
  IDictionary<string, object> ids)
{
  var R = getR(Rcol);

  if (ids.Count != 2 && R.Type != RelationType.Multi)
    throw new InvalidOperationException(
      "only_binary_relations_can_be_created_with_constraints");

  var id = DB.GetRelId(Rcol, ids);

  var e1 = ids.First();
  var e2 = ids.Last();
  switch (R.Type)
  {
    case RelationType.R_11:
      if (CassandraDB.EKeys(e1.Key, Rcol, e1.Value.ToString()).Any() ||
        CassandraDB.EKeys(e2.Key, Rcol, e2.Value.ToString()).Any())
        throw new InvalidOperationException(
          "Relation_type_constraint_violation");
      break;
    case RelationType.R_1N:
      if (CassandraDB.EKeys(e2.Key, Rcol, e2.Value.ToString()).Any())
        throw new InvalidOperationException(
          "Relation_type_constraint_violation");
      break;
    case RelationType.R_NM:
      if (R.AsColumnFamily().Get(id.ToString()).CountColumns() != 0)
        throw new InvalidOperationException(
          "Relation_type_constraint_violation");
      break;
    default:
      break;
  }

  var relation = R.AsColumnFamily().CreateRecord(id.ToString());
  foreach (var e in R.Entities)
  {
    var eId = ids.ContainsKey(e) ? ids[e].ToString() : string.Empty;
    if (string.IsNullOrEmpty(eId))
      throw new ArgumentException(
        "ids: entity not found in relation metadata");

    var eDoc = getE(e).AsColumnFamily().Get(eId.ToString()).First();
    if (eDoc.Columns.Count == 0)
      throw new InvalidOperationException(string.Format(
        "Entity {0} with key {1} not found.", e, eId));

    relation[e + "__id"] = eDoc.Key;
    foreach (var attr in eDoc.Columns)
    {
      relation[e + "_" + attr.ColumnName] = attr.ColumnValue;
    }

    CassandraDB.AddToEKeys(e, Rcol, eId, id.ToString());
  }

  relation["idxall"] = 0;

  CassandraDB.DB.Attach(relation);
  CassandraDB.DB.SaveChanges();

  return relation;
}

/// <summary>
/// Delete(R):
///
/// r = find(R)
/// transfer
/// updateIndex(IX_R)
/// delete(r)
///
/// </summary>
/// <param name="Rcol">Relation name</param>
/// <param name="ids">Relation related entities keys</param>
/// <returns>Operation result: true</returns>
public override object DeleteRelation(string Rcol,
  IDictionary<string, object> ids)
{
  var id = DB.GetRelId(Rcol, ids);
  getR(Rcol).AsColumnFamily().RemoveKey(id.ToString());

  foreach (var E in ids)
    CassandraDB.RemoveFromEKeys(E.Key, Rcol, E.Value.ToString(),
      id.ToString());

  return true;
}

/// <summary>
/// Update(R):
///
/// find(R)
/// updateIndex(IX_R)
/// update(R)
/// transfer
/// </summary>
/// <param name="Rcol">Relation name</param>
/// <param name="ids">Relation related entities keys</param>
/// <param name="toUpdate">Relation attributes to update</param>
/// <returns>Operation result: true</returns>
public override object UpdateRelation(string Rcol,
  IDictionary<string, object> ids, IDictionary<string, object> toUpdate)
```

```
{
  var id = DB.GetRelId(Rcol, ids);

  var rec = getR(Rcol).AsColumnFamily().Get(id.ToString()).First();
  if (rec.Columns.Count == 0)
    return false;

  var toU = toUpdate.Where(a => !a.Key.EndsWith("_id"));
  if (toU.Count() == 0)
    return true;

  update(rec, toU);

  CassandraDB.DB.Attach(rec);
  CassandraDB.DB.SaveChanges();

  return true;
}

#endregion

#region Read

/// <summary>
/// Read(A): find by key
///
/// find(A, Aid)
///
/// </summary>
/// <param name="col">Entity name</param>
/// <param name="id">Entity key</param>
/// <returns>MaxR containing the single entity (autorelation)</returns>
public override IRelation ReadEntity(string col, object id)
{
  var doc = getE(col).AsColumnFamily().Get(id.ToString()).First();
  if (doc.Columns.Count == 0) return new MaxR();
  var maxr = new MaxR(col);
  var prefix = string.Empty;
  if (!DB.Relations.ContainsKey(col)) prefix = col + "_";
  maxr.Add(doc.ToRow(prefix));
  return maxr;
}

/// <summary>
/// Read(R): find by key
///
/// find(R, Rids)
/// transfer
///
/// </summary>
/// <param name="col">Entity name</param>
/// <param name="id">Entity key</param>
/// <returns>MaxR containing the single relation</returns>
public override IRelation ReadRelation(string Rcol,
  IDictionary<string, object> ids)
{
  var R = getR(Rcol);
  var entityKeys = new Dictionary<string, object>();
  var rel = R.AsColumnFamily().Get(DB.GetRelId(Rcol, ids).ToString()).First();
  if (rel.Columns.Count == 0) return null;
  foreach (var ename in R.Entities)
  {
    var id = ids.ContainsKey(ename) ? ids[ename].ToString() : string.Empty;
    entityKeys.Add(ename, id);
  }
  return new MaxR(R.Entities) {
    Rows = new List<IRelationRow> {
      new MaxRRow { Attributes = rel.ToRow(), Keys = entityKeys }
    }
  };
}

/// <summary>
/// Read(A): query
///
/// select(A, p)
/// transfer
///
/// </summary>
/// <param name="col">Entity name</param>
/// <param name="query">Read predicate</param>
/// <returns>MaxR containing the entities</returns>
public override IRelation LoadEntities(string col, IQuery query = null)
{
  var maxr = new MaxR(col);
  var E = getE(col);

  IEnumerable Es;

  if (query == null)
    Es = E.ExecuteCassandraQuery(FixedQuery.All(col), CassandraDB);
  else
    Es = E.ExecuteCassandraQuery(query, CassandraDB);

  if (Es is IEnumerable<ICqlRow>)
  {
    foreach (var e in (Es as IEnumerable<ICqlRow>))
    {
      maxr.Add(e.ToRow(E.Name + "_"));
    }
  }
  else
  {
    foreach (var e in (Es as IEnumerable<FluentColumnFamily>))
    {
      maxr.Add(e.ToRow(E.Name + "_"));
    }
  }
  return maxr;
}

/// <summary>
/// Read(R): query
///
/// rA = select(A, p)
/// transfer
/// select(R, R.Aid IN rA.id)
/// transfer
/// foreach (var X in ({R}-A)) {
///       ids = distinct(kR)
///       foreach (id in ids) {
///               find(X, id)
///       }
///       transfer
/// }
/// join()
///
/// </summary>
/// <param name="col">Entity name</param>
/// <param name="id">Entity key</param>
/// <returns>MaxR containing the relations</returns>
public override IRelation LoadRelations(string Rcol, IQuery query = null)
{
  IEnumerable<object> Rs;

  var R = getR(Rcol);
  if (query == null)
    Rs = R.ExecuteCassandraMaxRQuery(FixedQuery.All(Rcol), CassandraDB);
  else
    Rs = R.ExecuteCassandraMaxRQuery(query, CassandraDB);

  var relations = new List<IRelationRow>();
  foreach (var rel in Rs.ToCassandraRow())
  {
    var ents = R.Entities.
      ToDictionary(e => e, e => rel[e + "__id"].GetValue());
    relations.Add(new MaxRRow {
      Attributes = rel.ToRow(), Keys = ents
    });
  }
  return new MaxR(R.Entities) { Rows = relations };
}


#endregion

#region privates

private static void update(FluentColumnFamily rec,
  IEnumerable<KeyValuePair<string, object>> toUpdate)
{
  foreach (var attr in toUpdate)
    rec[attr.Key] = attr.Value.ToString();
}

#endregion
}
```

Bibliography

[1] 10gen. Csharp language center, mongodb c# / .net driver. http://docs.mongodb.org/ecosystem/drivers/csharp/. Last access: 19/03/2013.

[2] 10gen. Database references. http://docs.mongodb.org/manual/applications/database-references/. Last access: 19/03/2013.

[3] 10gen. Master detail transactions in mongodb. http://blog.mongodb.org/post/7494240825/master-detail-transactions-in-mongodb. Last access: 08/11/2012.

[4] 10gen. mongodb. http://www.mongodb.org/. Last access: 08/11/2012.

[5] 10gen. Perform two phase commits. http://docs.mongodb.org/manual/tutorial/perform-two-phase-commits/. Last access: 08/11/2012.

[6] 10gen. Replica set fundamental concepts. http://docs.mongodb.org/manual/core/replication/. Last access: 19/03/2013.

[7] Amazon Elastic MapReduce Adam Gray, Product Manager. Aws howto: Using amazon elastic mapreduce with dynamodb (guest post). http://aws.typepad.com/aws/2012/01/aws-howto-using-amazon-elastic-mapreduce-with-dynamodb.html. Last access: 04/11/2012.

[8] Amazon. Amazon dynamodb faqs. http://aws.amazon.com/dynamodb/faqs. Last access: 02/11/2012.

[9] Ed Anuff. Indexing in cassandra. http://anuff.com/2011/02/indexing-in-cassandra/. Last access: 19/03/2013.

[10] Ed Anuff. Secondary indexes in cassandra. http://anuff.com/2010/07/secondary-indexes-in-cassandra/. Last access: 19/03/2013.

[11] Apache. Apache cassandra project. `http://cassandra.apache.org/`. Last access: 19/03/2013.

[12] Apache. Cassandra wiki, high level clients. `http://wiki.apache.org/cassandra/ClientOptions`. Last access: 19/03/2013.

[13] Apache. Extension methods (c# programming guide). `http://msdn.microsoft.com/en-us/library/vstudio/bb383977.aspx`. Last access: 01/04/2013.

[14] Apache. Welcome to apache hadoop! `http://hadoop.apache.org/`. Last access: 10/02/2013.

[15] Eric A. Brewer. Towards robust distributed systems (abstract). In *Proceedings of the nineteenth annual ACM symposium on Principles of distributed computing*, PODC '00, pages 7–, New York, NY, USA, 2000. ACM.

[16] Rick Cattell. Scalable sql and nosql data stores. *SIGMOD Rec.*, 39(4):12–27, May 2011.

[17] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Trans. Comput. Syst.*, 26(2):4:1–4:26, June 2008.

[18] Ming-Syan Chen and P.S. Yu. Combining joint and semi-join operations for distributed query processing. *Knowledge and Data Engineering, IEEE Transactions on*, 5(3):534–542, 1993.

[19] Cloudant. Bigcouch, a highly available, fault-tolerant, clustered version of apache couchdb. `http://bigcouch.cloudant.com/`. Last access: 08/12/2012.

[20] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM symposium on Cloud computing*, SoCC '10, pages 143–154, New York, NY, USA, 2010. ACM.

[21] Couchbase. Couchbase server, the nosql document database. `http://www.couchbase.com/couchbase-server/overview`. Last access: 08/12/2012.

[22] Couchbase. Replace your memcached tier with a couchbase cluster. `http://www.couchbase.com/memcached`. Last access: 08/11/2012.

[23] Couchbase. Why nosql ? http://www.couchbase.com/why-nosql/nosql-database. Last access: 08/12/2012.

[24] DataStax. Datastax community edition of apache cassandra. http://www.datastax.com/products/community. Last access: 19/03/2013.

[25] DataStax. Secondary indexes, apache cassandra 0.7 documentation. http://www.datastax.com/docs/0.7/data_model/secondary_indexes. Last access: 19/03/2013.

[26] DataStax. Using column families as indexes, apache cassandra 0.7 documentation. http://www.datastax.com/docs/0.7/data_model/cfs_as_indexes#indexes. Last access: 19/03/2013.

[27] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, January 2008.

[28] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: amazon's highly available key-value store. *SIGOPS Oper. Syst. Rev.*, 41(6):205–220, October 2007.

[29] Brad et al. Fitzpatrick. Memcached. http://memcached.org/, April 2010. Last access: 02/11/2012.

[30] Seth Gilbert and Nancy Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33(2):51–59, June 2002.

[31] Neo Technology Inc. Neo4j, the world's leading graph database. http://www.neo4j.org/. Last access: 10/11/2012.

[32] David Intersimone. The end of sql and relational databases ? http://blogs.computerworld.com/15510/the_end_of_sql_and_relational_databases_part-_1_of_3,http://blogs.computerworld.com/15556/the_end_of_sql_and_relational_databases_part_2_of_3,http://blogs.computerworld.com/15641/the_end_of_sql_and_relational_databases_part_3_of_3, February 2010.

[33] David Karger, Eric Lehman, Tom Leighton, Matthew Levine, Daniel Lewin, and Rina Panigrahy Abstract. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world

wide web. In *In Proc. 29th ACM Symposium on Theory of Computing (STOC*, pages 654–663, 1997.

[34] Jay et al. Kreps. Project voldemort - a distributed database. `http://www.project-voldemort.com/`, 2010.

[35] Jay et al. Kreps. Project voldemort - design. `http://www.project-voldemort.com/voldemort/design.html`, 2010.

[36] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, July 1978.

[37] managedfusion. Fluentcassandra. `https://groups.google.com/forum/#!forum/fluentcassandra`. Last access: 19/03/2013.

[38] Ken North. Databases in the cloud. *Dr. Drobbs Magazine*, September 2009.

[39] Dan Pritchett. Base: An acid alternative. *Queue*, 6(3):48–55, May 2008.

[40] Hibernating Rhinos. Ravendb, open source 2nd generation document db. `http://ravendb.net/`. Last access: 10/11/2012.

[41] Salvatore et al. Sanfilippo. Redis. `http://code.google.com/p/redis/`, 2010. Last access: 02/11/2012.

[42] Scalaris. Scalaris, a distributed transactional key-value store. `http://code.google.com/p/scalaris/`. Last access: 02/11/2012.

[43] Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: a tutorial. *ACM Comput. Surv.*, 22(4):299–319, December 1990.

[44] Stephen Yen. Nosql is a horseless carriage. `http://dl.getdropbox.com/u/2075876/nosql-steve-yen.pdf`, November 2009. Last access: 02/11/2012.