Politecnico di Milano
Scuola di Ingegneria Industriale e dell'Informazione
Corso di Laurea Magistrale in Ingegneria Informatica
Dipartimento di Elettronica, Informazione e Bioingegneria

Mapping Relational Databases into a Distributed NoSQL

Advisor: Prof. Elisabetta DI NITTO
Co-advisor: Marco SCAVUZZO

Master thesis by:
Francesco GRAZIOLI - 784120

Academic Year 2013–2014

Alla mia famiglia

Ringraziamenti

Questo lavoro porta con sè molti grazie: alla relatrice Elisabetta Di Nitto per avermi proposto questo argomento di tesi così interessante, innovativo e stimolante, e per la fiducia e l'incoraggiamento dimostratomi; al correlatore Marco Scavuzzo per il suo grandissimo aiuto, per avermi insegnato e guidato con pazienza e attenzione, grande disponibilità e collaborazione, durante tutto il percorso di realizzazione della tesi.

La mia più sincera gratitutidine a mia mamma e mio zio Daniele per essermi sempre stati vicini con il loro appoggio, sostegno e fiducia in tutti questi anni.

Un ricordo speciale a coloro che oggi non ci sono più ma che sono sempre vivi dentro di me: ai miei nonni ai quali ho vouto molto bene e a mio papà, perchè è anche grazie a lui se oggi ho raggiunto questo importante traguardo.

Un pensiero speciale alla mia ragazza Ardea, che mi è stata vicina e che mi ha dato la forza necessaria per concludere il mio percorso di studio.

Infine voglio ringraziare i miei compagni di corso, gli amici con i quali ho condiviso l'appartamento di Milano e tutti gli altri altri che mi sono stati vicini in questi anni.

Grazie

Milano, 18 Dicembre 2014

Francesco

Contents

List of Figures

List of Tables

List of Algorithms

Abstract

With Web 2.0, storing methods and technologies are radically changed. Nowadays we live in a world where the huge amount of data is distributed around different places, strongly limiting the usage of traditional databases, like RDBMS. In the last years, a new type of databases has found more and more space. They are called NoSQL, distributed databases that have different architectural features from the traditional ones and they also guarantee properties like high availability and scalability. However, they are not the solution to every problem of data management. They show some significant shortcomings that have an impact on how model applications are built and operate. One of these shortcomings is the absence of relational structures, useful to correlate data. One of the research objectives is to improve these aspects, maintaining the peculiar NoSQL database characteristics. This thesis aims at offering to designers the possibility to exploit scalability of NoSQL still adopting a relational model, so to take advantage from both approaches. To this end, we define a mapping model for data from a typical relational structure into a NoSQL one, based on the queries known at design time that will be performed on the database. The thesis presents the proposed approach and exploits HBase as target NoSQL for the mapping. HBase is particularly interesting because of its scalability and of its integration with Hadoop, that is, a map/reduce-based parallelization approach to support the execution of computations on large datasets. The approach is compared with a traditional relational one and the result we have achieved shows that it is promising in terms of improved performance on query execution.

Estratto

Con l'avvento del Web 2.0, i metodi e le tecnologie per la gestione dei dati son dovuti cambiare radicalmente. Al giorno d'oggi viviamo ormai in un mondo dove la grande quantità di dati, distribuita su tutto il territorio, limita fortemente l'utilizzo di database tradizionali, quali sono i RDBMS. A partire da alcuni anni si parla infatti di NoSQL, database distribuiti che presentano caratteristiche architetturali diverse da quelle tradizionali e che garantiscono alta disponibilità e scalabilità in base al carico richiesto. Dall'altro lato però, questo tipo di database presenta alcuni difetti che hanno un impatto sul modo in cui operano le applicazioni. Uno di questi difetti è la mancanza di strutture relazionali, utili per correlare dati. Uno degli obiettivi di ricerca è migliorare questi aspetti, mantenendo le caratteristiche peculiari dei database NoSQL. Questa tesi vuole offrire ai progettisti la possibilità di sfruttare la scalabilità di un database NoSQL, adottando comunque un modello relazionale, in modo da trarre vantaggio da entrambi gli approcci. Per questo, abbiamo definito un modello per la mappatura di dati, da una tipica struttura relazionale ad una NoSQL, basata sulle query che andranno eseguite sulla struttura finale, conosciute in fase di progettazione. La tesi presenta l'approccio proposto e valorizza l'utilizzo di HBase come database NoSQL per la mappatura. HBase è particolarmente interessante per la sua scalabilità e per la sua integrazione con Hadoop, un approccio di parallelizzazione basato su map/reduce, per supportare l'esecuzione computazionale su grandi quantità di dati. L'approccio è paragonato con uno relazionale tradizionale, e il risultato che abbiamo raggiunto mostra come sia promettente in termini di miglioramento delle prestazioni delle richieste ricevute.

Chapter 1

Introduction

Thanks to the increasing diffusion of technologies like cloud computing and smartphones, to the widespread use of data-intensive services like social networks, and to the possibility of building complex systems, the importance of collecting and storing data concerning industrial and economical processes as well as people habits, is growing more and more every day.

We are talking about very large amounts of data (they are called "Big Data" [14]) that have to be stored with increasingly velocity, and used and analyzed for different kind of purposes.

The new requirements involve technologies able to store huge quantities of not-structured data (we are talking about 2.5 PetaBytes of data everyday, like figure 1.1 shows), maintaining acceptable query performances, requirements not satisfiable by a traditional database.

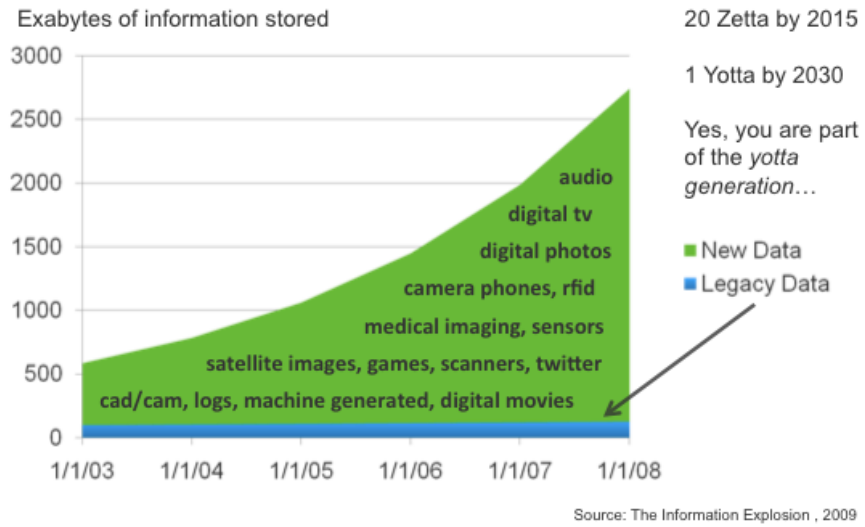


Figure 1.1: Information stored

In front of these new urgent requests, NoSQL (Not SQL, means not relational)

datastores found more and more space, characterized by the fact that they can reach higher performances and be more adaptable (scalable) to workload changes, all based on a distributed architecture, not centrally located as in traditional RDBMS.

For this reason, NoSQL datastores do not provide many of the structured mechanisms of storing data provided by RDBMS systems.

In fact, one of the main lacks of NoSQL datastores is that they do not provide join structures or structured query languages, like SQL in RDBMS. This fact can affect their performances on queries execution.

A challenging research objective is to improve the manageability of NoSQL datastores, still keeping their remarkable characteristics. This thesis aims at offering a contribution towards the achievement of this challenge. It proves that this lack can be partially avoided when the queries, that are going to be performed on the database, are known at design time. This result is achieved proposing a way of mapping data from an initial relational structure into a NoSQL datastore structure, based on some given input query. In particular, we analyzed all Relational Algebra query operands, proposing a feasible mapping model for all of them. This model allows the user to exploit NoSQL features, still adopting a relational model, in order to take advantage from both approaches. The final aim of this work is to compare this model with a relational based model, focusing on different performances obtained after query tests execution.

In order to be able to perform this comparison, we chose a NoSQL datastore, HBase. It is a column-based datastore, particularly adapt to store unstructured data, providing excellent scalability and fault tolerance features. In particular, we needed to understand how to configure it, and its splitting policies, useful to optimize data distribution for our tests.
About the configuration, we built a bash Linux-based script that avoids a long work of copying files into all cluster machines. We also built a Java tool that provides all necessary structures and functions as a proof-of-concept for the mapping model proposed.

The final part consists in text execution. Thanks to the bash Linux-based script and to the Java tool, we were able to perform tests on a chosen dataset, significant for our model. Tests execution compares the results obtained by the proposed model with a classical relational model, mapped into an HBase cluster. Test results show how our model provides extremely better performances performing the execution of the queries given at design time.

Outline of the Thesis

This document is structured as follows:

- Chapter 2 describes the State of the Art. In this section are presented the main features of NoSQL databases and one of their possible classification based

on the data model. In particular, we presented two databases, compared for their architectural differences and characteristics, focusing on consistency and availability requirements. At the end of this Chapter we describe architectural and properties of the datastore we decided to use for our analysis, HBase.

- Chapter 3 presents two approaches of denormalization of data from a relational point of view into a NoSQL-adaptable one. In particular, in this section is presented the denormalization model for first approach, based on a E-R query schema and the schema of the query, known at design time. The model analysis is sustained by two examples and an analysis of its limits.

- Chapter 4 shows a deeper analysis on HBase, in order to be able to execute the tests shown in Chapter 5. In particular, we analyzed HBase splitting policies, useful to understand how to design a good row-key for our model. Secondly is shown how to configure and install a working HBase cluster. At the end of the Chapter we presented a Java tool able to denormalize entities following our model equations.

- Chapter 5 shows the execution of some tests. We propose them to compare our model with a relational based one. For each test we present its description, execution and results.

- Chapter 6 is dedicated to final conclusions and possible future works.

Chapter 2

State of the art

This Chapter introduces the main characteristics of NoSQL datastores, especially focusing on the main differences between them and the traditional RDBMS.

Main NoSQL datastores properties will be introduced from a theoretical point of view and they are classified starting from a data-model perspective.

Then we compare two different datastores, analyzing their different architectures structures and provided properties.

At the end of the Chapter we will focus on a particular NoSQL database, HBase, presenting its general features and describing why we choose it for our work.

2.1 Introduction to NoSQL

The term NoSQL ("Not SQL", the query language used in relational databases) was used for the first time to describe a relational database that omitted the use of SQL in 1998. It does not have a single and unique definition but we can say that it represents a new kind of databases (or better, datastores) that do not use relational models, as traditional RDBMS, and that are increasingly used nowadays for their main characteristics: horizontal scalability, high availability and schema-free. We will see their definitions in the following paragraph.

Usually a NoSQL datastore is preferred over a RDBMS database when is necessary a more efficient way to store and manage huge amounts of data, while relational databases fit well for data that is rigidly structured with relations and allows for dynamic queries expressed in SQL language.

However, today web data is very different: it is not well structured and it does not need dynamic queries because many applications use prepared procedures and statements.

In addition, RDBMS work well in centralized systems, while nowadays systems are often fully distributed, thanks to a growing and cheaper utilization of cloud services.

By the fact that one of the most common scenarios of everyday data stored from web, applications, or industrial and economical processes need to be used to understand user's habits and preferences or to better optimize industries processes, this amount of data need to be mostly read in a very efficient way and in the shortest possible time. NoSQL databases are the perfect way to store them and to optimize this kind of queries for the reasons we are going to see in the following paragraphs.

2.2 NoSQL properties and characteristics

There is no a unique and formal definition of "NoSQL", but we can say that a NoSQL datastore is a distributed database where there are few (or none) relations and there is a very flexible data model.

All datastores marked as NoSQL are characterized by these particular features:

Horizontal Scalability: it is the ability of a system (a database) to handle a growing (or decreasing) amount of work in a capable and user-transparent way. This property is achieved thanks to the possibility of "*sharding*" the data, or better to split horizontally the data in a database. This function allows to access smaller pieces of data in a faster and easily managed way and allows the system to add or delete nodes from the datastore, without changing its structure and without a loss of performances in reads or writes. The main problem that this feature break out is that Join operations become very complex and inefficient.

Fault tolerance: when the amount of data increases, the datastore is able to recover after single or multiple failures. To do that, all NoSQL datastores provide data replication, storing different copies of the same piece of data; when a single replica is lost, the datastore can use its copies and it does not lose any information.

High availability: each data stored in a NoSQL database has to be reachable in a very limited time; like fault tolerance, this features is provided thanks to data replication. When the client asks the datastore for some data, the system answers with the nearest copy of the data requested, reducing time needed to answer and allowing multiple simultaneous answers to different clients, also when their number increases in a small time period.

Distributed and cloud oriented: NoSQL databases are based on a distributed infrastructure, distributing data on different nodes that can be managed in a very elastic way.

This kind of configuration permits to be more efficient and have better performances, distributing the workload on each node and not collect it on a single central node.

Flexible data schema (or schema-free): in contrast with traditional RDBMS, the new NoSQL datastores have not a fixed data schema.
Unlike RDBMS fixed table structures, in this kind of systems different types of data can be stored. Due to this property, they are defined "free-schema" databases.

All these properties make NoSQL datastores particularly suitable for different scenarios:

- **BigData Managment**: this situation implies storing very huge amounts of data, respecting its availability, scalability and distribution requirements. Most of NoSQL datastores are involved in this scenario.
  They have different and complicated methods for replicating and managing data, in order to resolve any kind of failure.
  Google BigTable is the main example for this kind of scenario: it states a solid storing basis for cloud services and for all Google web tools.

- **RDBMS features:** few of NoSQL datastores try to maintain the relational model used in RDBMS. These ones try to resolve complex queries and also queries with relations, not a peculiar structure in NoSQL. RavenDB [1] is probabily the most known datastore that tries to emulate a RDBMS database and its properties.

- **Hashing tables:** key-values NoSQL datastores can be used also as hashing tables (also called "cache-tables") for RDBMS databases.
  They are used as map-tables to get simple key values, after used in a RDBMS database to get the full row information.

We add below another property that characterize NoSQL datastores: it is replication.

Replication

Replication means that all data stored in distributed databases are stored not only in one single physical space.

This feature involves an improvement of database performance, allowing the load balancer (the component that is responsible of assigning a node to each client operation) to distribute read operations over different storing nodes, helping a very high improvement for systems availability. Replication helps also the datastore to recover rows of data when a node is offline or dead, providing a backup function.

The main problem caused by storing not one but many copies of a single piece of data can be write operations. If all copies of same data are stored consistently, the datastore can reach better performances to answers read operations.

[1] Hibernating Rhinos, http://ravendb.net/

There are two main approaches that characterize NoSQL databases: the first is to wait for the write of all copies before giving to clients the positive acknowledge (synchronous writes).

The second is to guarantee the positive ack for the first copy and the asynchronous update of the other replicas in a sequent time, providing the correct value for read operations.

These two approaches are better explained later in this work with Brewer's "CAP Theorem" (2.2.1).

2.2.1 CAP Theorem

To define some NoSQL datastores features it is necessary to introduce a theorem called "*CAP Theorem*".

It was introduced by Brewer in 2000 [4] and consists of a simple concept: "it's impossible for a web service to provide Consistency, Availability and Partition-Tolerance at the same time". This statement is proved by contradiction in Brewer's paper.

All of these three properties listed before are nowadays expected from a real-world web service. Their definition are the following:

- **Consistency:** all clients provide the same answer to the same request done at the same time. It means that all copies of the same data should be updated to latest values;
- **Availability:** every client that makes a request to a node, will provide always an answer. It refers to the fact that all the queries that are performed must be completed;
- **Partition-tolerance:** the system continues to work despite message loss, as node or part of the system fails or if the network is divided in two or more partitions. The system has to be able to supply an alternative way when these problems arise.

As a result of the theorem, we can have a database with only one of these three kind of combination properties: CA, CP or AP (figure 2.1).

NoSQL databases first aim is to maintain the Partition-tolerance property, because they are designed as distribute systems: if they are designed choosing Consistency and Availability properties, when a node failure occurs, first of all it can be comparable to a loss of availability, and second it is impossible to recover its values consistently with the correct ones.

From this point of view, a loss of Availability follows a Partition loss, so the choice they have to do is to select either Consistency or Availability, in addition to Partition Tolerance.

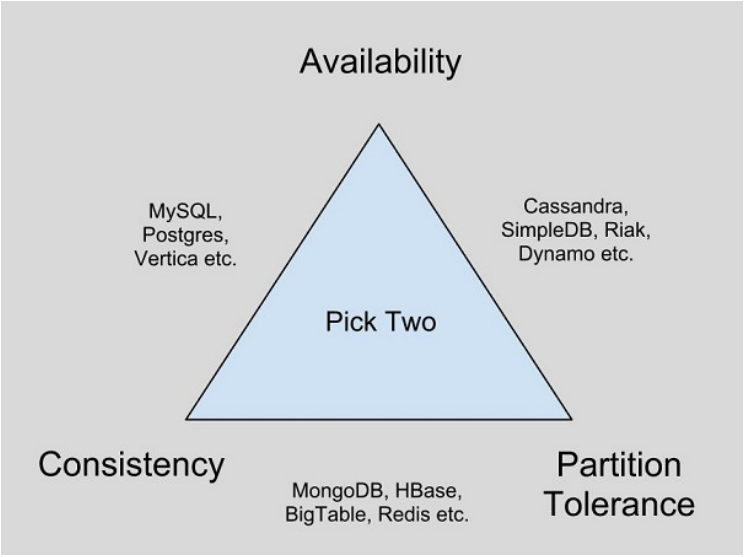


Figure 2.1: CAP theorem

As we have already said in the previous paragraph, this choice is made, from a certain point of view, when a datastore selects how to perform data writes. If it prefers strong consistency, it will be described as "CP". "C" means consistency and it mainly refers to this property of having all copies of same data consistent at any point in time.

The other face of the medal is, instead, the Availability ("A") option: in systems where availability is preferred over consistency, a write operation finishes after the first copy of data is stored and the other copies are considered "consistent in a not specific future time". Last concept can be described with the definition of "Eventual Consistency". This property can't be described with classic RDBMS set properties (called "ACID", Availability, Consistency, Isolation and Durability), but it must be described with a new one. This new set of properties is the subject of the next paragraph.

2.2.2 A new set of properties: BASE

As we introduced in the previous paragraph, ACID properties can not be used to describe NoSQL datastores. Through the years a new set of flexible properties has been approached to explain most of NoSQL systems.

They are called BASE properties:

"BA" stands for *Basically Available:* it means that the the system responds to any request at any time; it can be a "failure", but it will be a response. "E" stands for *"Eventually consistent"*: as we already said, it means that in a NoSQL database can exists a moment where all the copy of the same data are not updated at the latest value.

Can exist a state of the system (the *"Soft state"*, "S") where all data are not completely consistent. BASE systems do not use two-phase locking (main feature of Relational systems, characterized by ACID properties), but they guarantee that sooner or later all data will be consistent.

These particular properties allow the system to be Basically Available all the time, unlike the RDBMSes where all update transactions have to be atomic, causing delays in answers.

All these features are peculiar for all NoSQL datastores that choose CAP Theorem's AP perspective, due to the fact that Eventual Consistency does not guarantee the "full C" property. They are the basis of Twitter, Google, Amazon and Yahoo services that the majority of people in the world use everyday.

2.3 NoSQL Classification

Over the past 10 years many different NoSQL datastores have been developed in order to answer specific companies requirements regarding high scalability performance, maintenance or availability.

There are some different approaches to classify all this kind of datastores. The most used is the one that considers their data model structure. There are also other classifications that rely on query models, and sharding or replication methods.

We will focus basically on data model classification because it is probably the best known method, quoting also other methods if it reveals necessary for a better explanation.

With regard to the data model, all NoSQL databases can be classified into four macro-groups: key-value, column-oriented, document-based and graph-based. In the following subsections we are going to explain each of these four categories.

2.3.1 Key-Value

All the rows stored in this kind of storage are represented under a form of a map between keys and values (figure 2.2).

Storing data with this method enables to write huge amounts of different types of data in a very easy way and to simply optimize its distribution. In fact the tables can be easily divided horizontally (sharded) and data distributed on different nodes allocated on multiple machines.

These kind of databases are usually used like hash-tables because they can easily be accessed using the key attribute. Storing data with different and disordered keys can be a disadvantage for sequential key-range queries.

The key-value databases are usually completely schema-free and the main operations *(put, get, delete)* are available. Examples of this kind of databases are *Project*

Voldemort [10], Redis [18] and Amazon DynamoDB [16] .

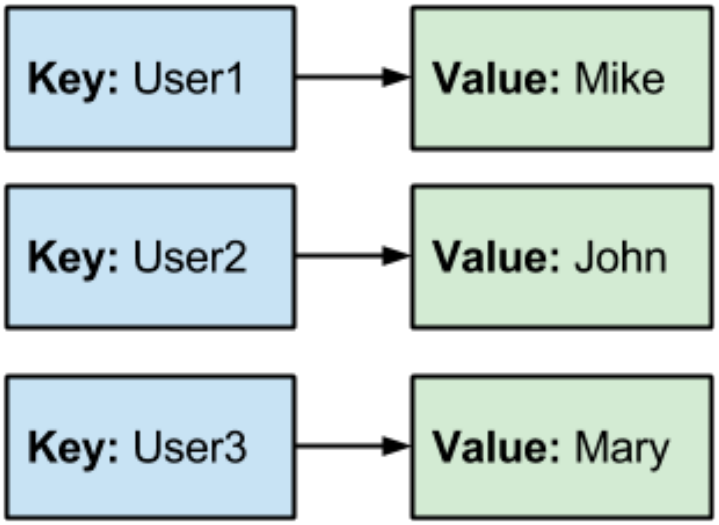


Figure 2.2: Key-value datastore

2.3.2 Column-oriented

They are similar to key-value stores (2.3.1), as they classify data through a key distribution. They can manage and distribute huge amount of rows over many and different physical machines.

These databases are the best for scaling-up and for managing big data volumes, because they can be partitioned both horizontally (on rows) and vertically (on families).

Concerning the data model aspect, the majority of column-oriented databases are inspired by Google Big Table [6] data model. They create collections of one or more key/value pairs that match a record.

Column families are NoSQL objects that contain columns of related data. Each row can have his own set of columns, different from the other rows (they are called «sparse datastores»), not requiring a pre-structured table. Each record comes with one or more columns containing the information. Basically, these datastores are two dimensional arrays whereby each row (key/record) has one or more key/value pairs attached to it and these management systems allow very large and unstructured data are kept and used.

For example, in figure 2.3, the first row can have only columns "Name" and "Age", while the other two can have all the three columns described ("Name", "Age" and "State").

They are very powerful and be reliably used to keep important data of very large size.

There are many different examples of this kind of datastores: *HBase* [2], *Cassandra* [3] and *Hypertable* [4].

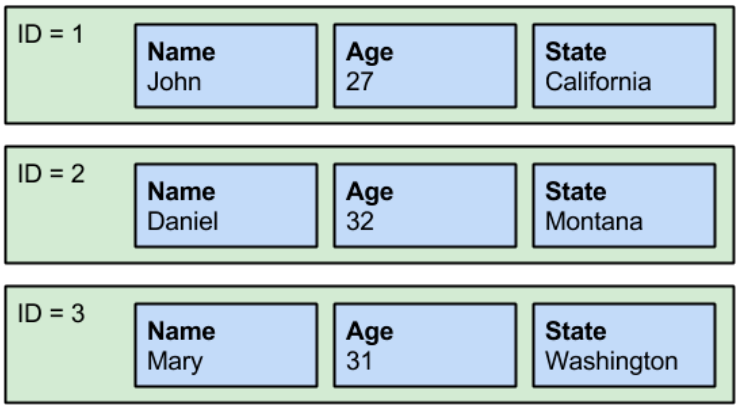


Figure 2.3: Column-based datastore

2.3.3 Document-oriented

This category of database manages document-oriented data (semi-structured data).

Each document can have a different set of attributes that can be different from other documents collected in the same group. The document-based databases can be considered like key-document datastores (similarly as key-value datastores), like figure 2.4 shows.

The main concepts provided on keys in 2.3.1 section are quite the same here. The *Value* field here is more complex and structured. A Document has more attributes or other nested documents, like XML files. These type of documents can be filtered and scanned, but with lower performance then the Key-Value ones, because of their data structure.

One of the most critical functionalities of these datastores is that they interact with applications through Javascript Friendly JSON.

Query model and API are very rich and a set of high level features is provided, like indexes, views, triggers, transactions, similar to RDBMS functionalities.

The most used document-oriented datastore used is *MongoDB* [5]. Another example can be *CouchDB* [6].

[2] Apache HBase, http://hbase.apache.org, The Apache Software Foundation.
[3] Apache Cassandra, http://cassandra.apache.org, Apache Software Foundation
[4] Hypertable Inc, http://hypertable.org/.
[5] Inc. MongoDB. http://www.mongodb.org/.
[6] Apache CoucheDB, http://couchdb.apache.org/, Apache Software Foundation

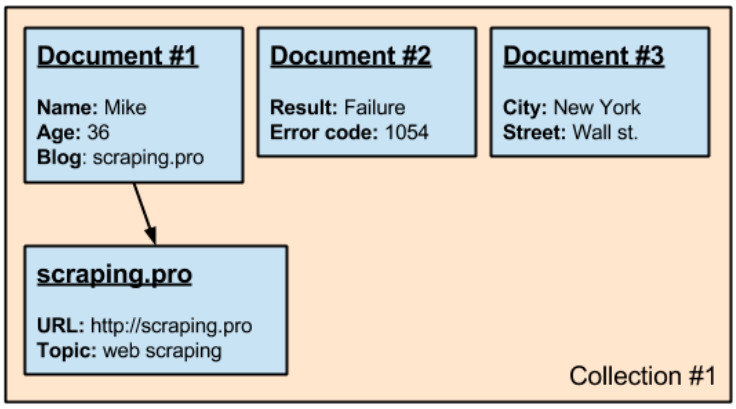


Figure 2.4: Document-based datastore

2.3.4 Graph-based

Last family of NoSQL database is the Graph-based family. This kind of datastore uses a graph representation to describe the entity structure.

Each node of the graph corresponds to an entity in the database and each edge between two entities represents a relationship between them (figure 2.5).

These kinds of database are quite different from the others. In graph-oriented datastores relationships are taken into account like a fundamental part of the database, unlike all the other datastores previously described. Social relationships (social networks) and maps problems can be easily described and queried with this kind of representation.

Due to the presence of relationships, ACID properties are generally provided.

Due to their particular structure, these databases provide query structures that allow classical graph functions as path, distance, neighbor, etc. .

They are particularly used when the data model is complex with many connections between entities and various degrees of other entities indirectly related to them.

Examples of this type of datastores are *Neo4j* [7] and *Infinite Graph* [8].

[7] Inc. Neo Technology, http://http://www.neo4j.org/.
[8] Objectivity InfiniteGraph, http://www.objectivity.com/infinitegraph/.

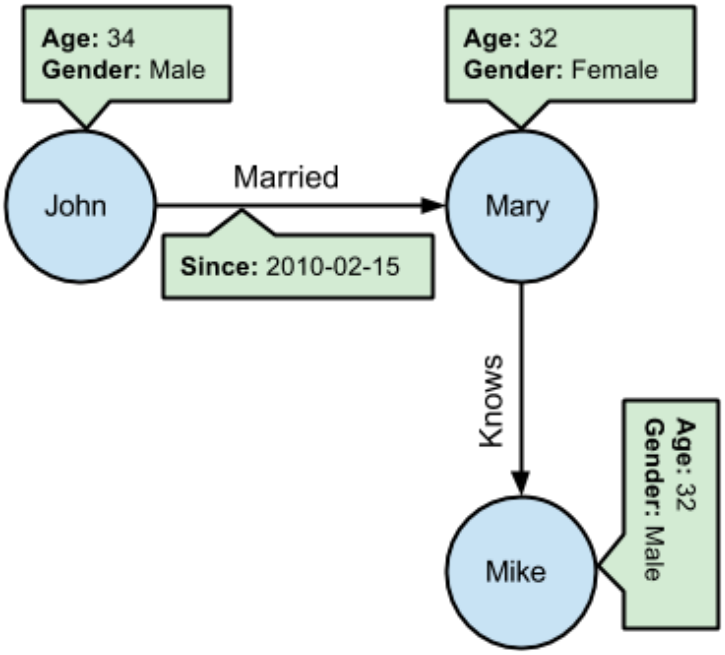


Figure 2.5: Graph-based datastore

After this classification we have to say that each architectural structure gives to each datastore different properties. One of the best ways to classify them from this point of view is to distinguish them from CAP Theorem's point of view, or better considering them as "CP" or "AP" datastores, as we have already said in paragraph 2.2.1.

The remaining part of this Chapter shows this comparison, highlighting similarities and differences between the two.

2.3.5 Consistency vs Availability

To better understand the difference between a datastore that provides Consistency and one that provides Availability, we now present two datastores mentioned during the previous classification that belong to the two different CAP theorem's perspective (CP and AP), described in 2.2.1, trying to show their features and how their architectural structure infers them specific properties.

The first is Amazon DynamoDB.

2.3.5.1 Amazon DynamoDB

One of the main representative key-value datastore is DynamoDB , as an example of an "AP" datastore, by the fact that it is a highly available key-value storage system.

As mentioned in "Dynamo: Amazon's Highly Available Key-value Store" [9], Amazon platform needs some operational requirements in terms of performance, reliability and efficiency, and to support continuous growth it has to be highly scalable. Reliability is probably one of the most important requirements because "even the slightest outage has significant financial consequences and impacts customer trust".

To achieve this level of availability, Dynamo sacrifices consistency under certain failure scenarios.

Its main features are listed below:

- **Data model:** key-value hash column and an optionally sorting key column for sorting data; each table is a collection of Items, each one formed by many attributes. The primary key is required.
- **CAP Theorem:** it is an AP (Availability - Partition Tolerance) datastore. BASE properties are guaranteed and is possible to set the eventual consistency level (reaching strict consistency grades); its default value is low and it maximizes the read throughput.
- **RDBMS:** no classical transactions are supported and a secondary index is not implemented;
- **MapReduce:** MapReduce is a "programming model for processing large datasets" [8]. In DynamoDB it is possible to perform MapReduce jobs.
- **Partitioning:** data can be auto-sharded into machines with key hashing mechanism. It is driven by table dimension or provisioned throughput.
- **Replication:** replication is performed by replication of data into different nodes (usually three copies). They are updated asynchronously so Eventual Consistency is a feature of this datastore.
- **Architecture:**

Dynamo is a fully distributed datastore without a single point of failure and it can be accessed via web service APIs. It can serve any level of request traffic and store any amount of data, paying a very low price for that.
Dynamo stores objects with keys in what can be represented as a peer to peer architecture. Its structure can be represented as a ring, where all nodes are at same level, as described in figure 2.6.
Dynamo allows *get* and *put* operations, performed finding the correct key value on the hash-ring.
One of the main requirements of this database is that it must scale incrementally. Dynamo's partitioning relies on consistent hashing to distribute the load across the different nodes. Each data is assigned to a single node by hashing the data item's

key and the correct node is chosen walking clockwise from the hash location (figure 2.6).
To achieve high availability, each data stored in Dynamo is replicated N times. After storing the first copy, the remaining N-1 copies are stored at the N-1 clockwise successor nodes in the ring. In the figure, if replication factor is 3, B replicates the *key* in nodes C and D, in addition to its copy. This fact makes Dynamo extremely highly available and durable.
Dynamo provides Eventual Consistency, so it is possible that a *put* request returns

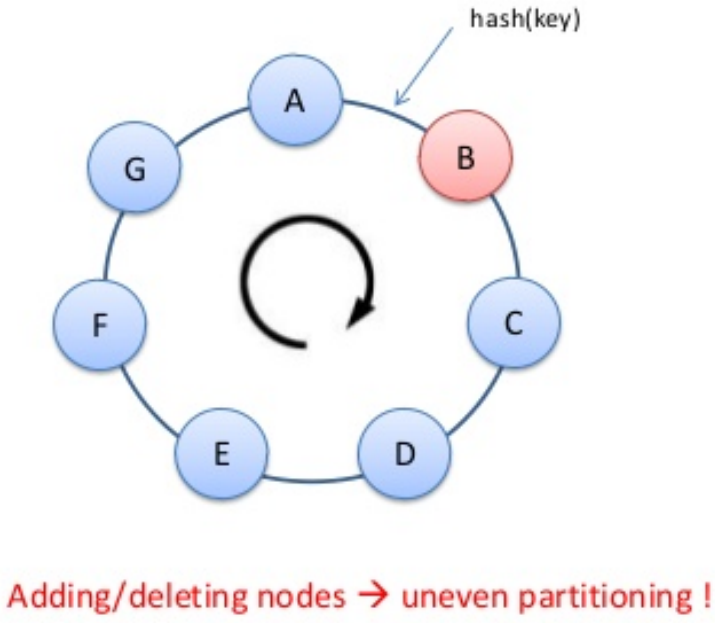


Figure 2.6: DynamoDB architecture

to its caller before the update has been applied to all replicas. If there are no node failures, Dynamo guarantees that each read operation on that value is correctly performed.
Dynamo, in fact, do not use a traditional quorum approach to maintain consistency among the replicas, because it would be unavailable during server failures and network partitions. It uses a "sloppy quorum". Quoting from the paper: "all read and write operations are performed on the first N healthy nodes, which may not always be the first N nodes encountered while walking the consistent hashing ring".
Considering again the figure 2.6, if A is temporally down during a write operation, its replica is sent to node D. This can maintain availability and durability. D stores the replica knowing that that would be stored into node A. If D detects that node A has been correctly recovered, the replica is sent to A and deleted in D, in order to maintain the total number of replicas in the system.
Using this method, a write operation is rejected if and only if all nodes are not available.

When a new node is added to the ring, a new key range is assigned to it. Due to this allocation, some other existing nodes have to send some of their keys to that new node. If a node is removed from the ring, the process is the same but reverse. This approach distributes the load of keys uniformly on the ring, ensuring latency requirements.
Most of the important techniques used by DynamoDB are summarized in the table below. All these techniques are described in [9].

Problem	**Technique**	**Advantages**
Data Partitioning	Consistent Hashing	Incremental scalability.
High Availability for writes	Vector clocks with reconciliation during reads	Version size is decoupled from updates 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 detenction	Gossip-based membership protocol and failure detenction	Preserves symmetry and avoids having a centralized registry for storing membership and node liveness information.

Table 2.1: Dynamo techniques ([9])

As we said, all described techniques are designed to make Dynamo extremely highly available, sacrificing Strong Consistency. This fact permits to categorize Dynamo as an "AP" datastore, according to CAP theorem point of view.
We now introduce another datastore, BigTable, that is quite different from Dynamo, both from requirements and architectural points of view.

2.3.5.2 Google BigTable

BigTable project was presented for the first time in 2004 by Google. It is build on its own file system (Google File System) and few other Google technologies. Of course it is mostly used for Google applications, such as Google Maps, Google Earth, Google Code, YouTube, Gmail, etc. .

We have to say that there are different classifications of this datastore, some

referring to it as a "column-based store", some as a "key-value datastore". The definition given by its own paper [6] is: “a 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”.

The main reason it was conceived was “scale the capacity of their clusters by simply adding more machines to the system as their resource demands change over time”. One of its particular features is that it allows to add or delete nodes in the order of seconds and it is able to manage huge amounts of data in a very small time.

These properties make BigTable one of the first datastores to prefer Consistency over Availability, in addition to Partition Tolerance, following CAP Theorem's directives. In order to maintain all five (default) replicas of each row, BigTable allows indeed a small loss of availability.

Replicas are stored synchronously so can occur a request failure when the datastore is managing these data.

Its main characteristics are:

- It can be defined as "a sparse, distributed, persistent and multidimensional sorted map": *sparse* because data stored can be very different and belong to different columns; *distributed* because data can be stored in hundreds of different nodes; *persistent* is a way to ensure no data losses; *multidimensional* due to its data model presented in the next point.

- **Data model:** the data model is a sort of three dimensions table with row-key, column-key and timestamps as axes: each data stored is a meeting point of these elements.
  Columns can be grouped under the definition of Column Families, that are simply a set of columns.

- **MapReduce:** MapReduce, "a programming model for processing and generating large data sets with a parallel, distributed algorithm on a cluster" [8], is the main feature of this database, allowing fast answers to client requests, sharding it into smaller queries.

- **RDBMS:** no secondary indexes, transactions atomic only on single rows.

- **CAP Theorem:** it is a CP (Consistency, Partition Tolerance) datastore. Each read and write is atomic under a single row-key: it ensures data consistency through the different replicas.

- **Partitioning:** when a table reaches a certain amount of data (100-200 MB), it is divided (sharded) into two subtables, in order to have better performances.

- **Architecture:**

BigTable architecture is completely different from Dynamo one. It is not structured as a peer-to-peer but it can be described with a Master-Node structure. As we have already said it uses its own File System (GFS) to store data files. BigTable stores data in tables called *SSTables,* that provide a "persistent, ordered immutable map from keys to values". Each operation consists of looking up the value linked to each key. Data are stored in blocks; each block has a starting point that is can be used as to locate it.
BigTable provides also a highly-available and persistent distributed lock service called Chubby [5]. It has the responsibility to maintain all the five replicas of each data consistent in face of failure. It provides also a directory file that can be used as a lock, to perform atomic reads and writes.
These characteristics make BigTable quite different from Dynamo. While Dynamo, as we said, sacrifices Strong Consistency on write operations to reach a higher availability, BigTable provides locking mechanisms to ensure Strong Consistency on all copies of data written, accepting loss of Availability.
We have also to say that Chubby constitutes the single point of failure of BigTable, since when Chubby is no available, BigTable becomes unavailable too.
As all Master-Slaves architectures, BigTable has one master server and many tablet servers (slaves), dynamically added to respond to workload changes.
The master is responsible for assign tablets (a group of tablets form a table) to tablet servers and balancing loads, while the tablet servers are responsible to manage a set of tablets and receive read or write requests from the clients.
If a tablet has too many data, it is splitted into two smaller tablets (this process is called "sharding"). This structure allows the client to communicate directly with the tablet servers, avoid overloading the master.
All tables and tablets information and locations are traced in tables called METADATA. Chubby stores a file that contains the first METDATA table location. This one contains all the locations of other METADATA tables, each one collecting tables and tablets locations. With this mechanism BigTable can address 2^{34} tables using only 128 MB of disk space. There are many other functions and features provided, whose description is shown in [6].

We described Dynamo and BigTable, trying to underline their architectural differences that bring us to consider Dynamo an highly Available datastore, while BigTable a "Consistent and Partition Tolerant" one. When choosing the database to use for our analysis, we took into account the better predisposition of a column-oriented datastore like BigTable to store sparse and very different types of data, but we also decided to set aside "AP" databases like DynamoDB because of BigTable's feature that guarantees strong consistency on write operations on single rows.

This particular property will be very useful for our considerations in the following Chapter.

Since it is not possible to use BigTable for our analysis (it's not distributed outside Google), we tried to select one of its derived column-oriented datastores, useful for our purposes. Finally, our choice fell on HBase.

Its dynamic data schema and the possibility to perform range based scan operations are additional reasons for our choice.

We now present HBase datastore, showing its features and architectural properties.

2.4 HBase

In this section we are going to present HBase datastore, as it will be the base for our work. As definition, HBase is "an open source, distributed, sorted map datastore modeled after Google Big Table". It means that:

- **Open source:** it is free downloadable from Apache website;
- **Distributed:** it can store and read data on a huge number of machines (from 1 to over tested 700);
- **Sorted map:** it provides a total ordering on its data-keys.

2.4.1 HBase architecture

Before analyzing HBase architectural structure, it's important to know that it runs on top of HDFS, the Hadoop Distributed File System [3], that we are now going to describe in the following paragraphs.

2.4.1.1 Hadoop

As presented in Apache WebSite [9], "Hadoop is an open source software framework for storage and large-processing of data-sets on commodity cluster hardware".

It is composed of four different modules: *Common*, *Yarn*, *MapReduce* and *Distributed File System*.

Common package contains the necessary Java ARchive (JAR) files and scripts needed to start Hadoop.

Yarn is a "resource-management platform responsible for managing compute resources in clusters (available from versions 2.0)".

MapReduce is "a programming model for processing and generating large data sets with a parallel algorithm".

The *Hadoop File System* is described in the paragraph below, as it is a consistent part also of HBase architecture.

[9] http://www.apache.org/, Apache Software Foundation

2.4.1.2 Hadoop HDFS

An Hadoop cluster usually consists of a single Namenode and many Datanodes.

The Namenode is the centerpiece of HDFS. It keeps the directory tree of all files in file system and tracks where across the cluster the file data is kept. Client applications talks with Namenode when they need to locate file.

These kind of systems do not support automatic recovery in case of Namenode failover. This is a well known and recognized single point of failure in Hadoop.

There is an optional SecondaryNamenode that can be hosted on a different machine. It is not like a second Namenode, but it only creates checkpoints of the system and holds an older copy of Namenode metadata.

Datanodes store data in the HDFS. They are grouped into racks, a common name to define a group of datanodes.

Hadoop provides data replication, that can be set by the user (default value is 3). If its value is set greater than 2, Hadoop provides the concept of "Rack Awareness", that means that the system guarantees that two copies of the same data are stored in a different nodes of the same rack, while the remaining copy is stored in a completely different rack. This is a plus property that helps to reach further better performances after a node failure.

All data received from clients are stored in blocks and all replica of these blocks are stored synchronously, providing Strong Consistency on writes of single rows.

Blocks

Like all existing File Systems, Hadoop provides this fundamental unit, but with bigger dimensions that in usual cases.

The common default dimension of a single block in File System is 512 KB, while in Hadoop HDFS it is set to 64 MB or 128 MB [10].

This fact is justifiable by the fact that this File System needs to store huge amounts of data in very small times, and this kind of setting allows it to reduce disk seek time, that is the time needed by the hard disk controller to locate the specific piece of stored data. This feature provided by Hadoop is one of the main reasons that improves HBase performances.

Strong Consistency and Replication

As mentioned in Brad Hedlund's article [12], Hadoop provides strong consistency on writes operations, storing all copies of the same value when this type of operation is requested.

When Hadoop receives a write request, the Client asks the Namenode to do that. The NameNode can give back to the client a list of n datanodes, where n is

[10] Apache Software Foundation. Hdfs-default settings, http://hadoop.apache.org/docs/r2.2.0/hadoop-project-dist/hadoop-hdfs/hdfs- default.xml. Accessed: 2014-03-04.

the replication value, where the client can write its data. Once the Namenode has provided the location of the datanodes to the client application, the client can talk directly to the Datanodes, asking them to store the data, as required.

The write operation is performed in blocks of data and each block can not be stored if the previous block is not stored in all of the n nodes, like as a pipelined write. This feature is quite similar to BigTable's Chubby function of performing write operations.

Namenode also tracks all the blocks written in the File System and knows perfectly in each moment where are all files. If a node or rack failure occurs, the NameNode knows that there are less copies of a single block in the cluster, so provides to store in a new datanode all the blocks lost.

In this section we presented the Hadoop features and File System, necessary to understand HBase architecture, presented below. We start from the basic unit of HBase tables, the *Region*, describing how they are organized and how they works, focusing on Region splitting policies in paragraph **??**.

2.4.1.3 Regions

HBase, like BigTable, has a Master-Node structure and stores files in tables. Each HBase table is associated to one or many *Regions*. They are the basic element of availability and distribution for tables.

Regions are non-overlapping; it means that a single row key belongs to exactly one region at any point in time. When a region is very large and contains a lot of data, it can be splitted, according to HBase splitting policy. We are going to analyze it in Section4.1.1.

Each region can hold data from only one table, but one table can have many regions.

There are two default tables in HBase, -ROOT- and .META.: the first one (do not used from version 0.96.0) keeps track of -META- table's location, while .META. keeps a list of all regions in the system and for each region the starting key of data that belongs to that field.

This simple table organization allows HBase to access all tables and regions in a very optimized way, like BigTable does.

HBase system is composed by two main processes: Master Server and Region Server.

1. **Master Server** is responsible for monitoring all Region Server instances in the cluster, and is the interface for all meta data changes.
   In a distributed cluster, the Master typically runs on the same machine of Hadoop Namenode (2.4.1.2), but if this machine goes down, HBase is shut down with a high probability of losing data.

Since version 0.20.0 HBase supports multiple Masters to provide higher availability. With this configuration, when the Master is down, an eligible Region Server is chosen as new Master Server.

2. **Region Server** must manage and serve regions. In a distributed cluster, a Region Server runs on an Hadoop Datanode machine.

An additional service used by an HBase cluster is Apache Zookeeper [13]: it is a sort of external coordinator that configures and synchronizes Master Server(s) and Region Server(s).

Following the formal definition it is a "distributed, open-source[11] coordinator service for distributed applications".

Its integration with Master and RegionServer nodes is shown in figure 2.7.

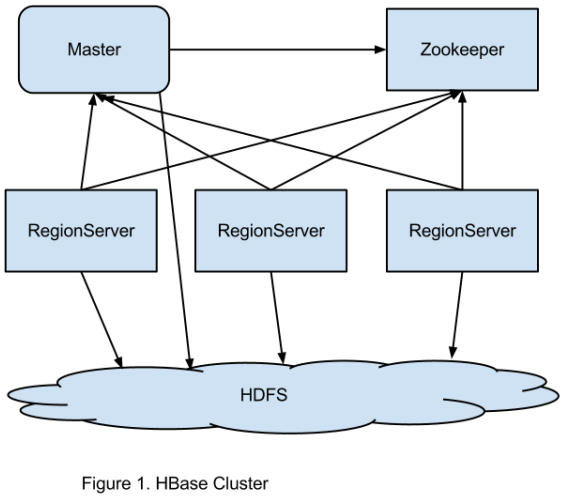


Figure 2.7: HBase cluster architecture

2.4.2 Data Model

As we have already seen, HBase system runs on top of HDFS (Hadoop Distributed File System), , that provides a fault-tolerant way of storing large quantities of sparse data.

We also said that each table is composed by many regions, each one is structured by the following dimensions:

Rows which are identified by a string-key of arbitrary length.

[11] Apache Zookeeper, http://zookeeper.apache.org/, Apache Software Foundation

Column Families which can occur in arbitrary number per row. As in Bigtable, column-families have to be defined in advance, during the creation of the table. The number of column families per table is not limited. Since HBase stores the name of the column family per each row stored, a good design implies short column family names.

Columns have a name and store a number of values per row which are identified by a timestamp (like in Bigtable). Each row in a table can have different number of columns, that can be defined at run-time. Client applications may specify the ordering of columns within a column family.

Timestamp which is created when the instance is stored. Different timestamps can be assigned to the same value, so HBase can store different copies of the same data. The user can also modify the timestamp in order to create different versions of the same entity.

The typical representation of a HBase data model is described below:

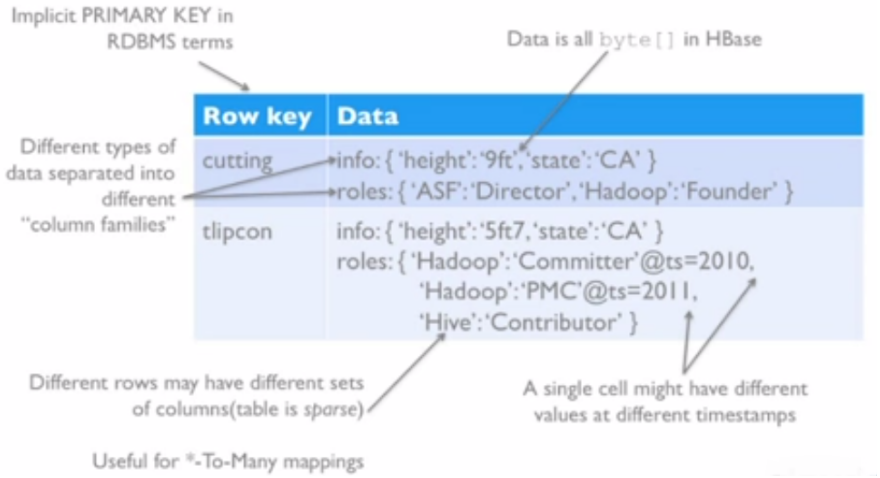


Figure 2.8: HBase Data Model

In figure 2.8 are described two different HBase rows, each one characterized by two column-families that group different columns. Each of these columns can be defined at run-time by the client (while column families need to be declared when the table is first created) and can be very different in each row. The annotation '@' also describes the presence of timestamps, useful if we want to track the past copies.

HBase is a column-oriented datastore, it provides an important feature that other datastores of the same data-model set (Cassandra for example) do not offer, that is *Ordered Partitioning*. This property implies that all rows are stored following Row-Key order, that means that HBase provides *row-Scans* operations among a range of Key-Values. This feature allows his datastore to provide better performances than the others with this type of searches.

Designing a good row-key on the basis of each single query implies only one or few *Scans* operation for each query, instead of many queries on different tables like in RDBMS.

2.4.3 Replication

When we talk about HBase Replication we can distinguish between HBase replication and Hadoop replication.

The first means that HBase provides one (or multiple) copy of all data and cluster structure. It means that copies of single rows and also of tables and regions information are kept only to be used when the whole cluster is unavailable and to restore it after a disaster. This replication is asynchronous and allows clusters to be geographically distant. This also provides a gap of consistency between the copies and can be not updated to the latest values every time.

Hadoop replication, as we have already said in HDFS paragraph (2.4.1.2), means that all rows stored into its file systems are saved together with one or two (or many) replicas.

2.4.4 API

HBase is mostly written in Java, so there are some Java Native API, useful to access and manage data faster.

In this paragraph are described the main classes and operations that are needed by a user that wants to manage an HBase cluster.

`HBaseAdmin:` an interface to manage HBase database metadata and general administrative functions. It can create, drop, disable or enable tables or column families.

`HTableDescriptor:` contains details about an HBase table such a descriptor of all the column families.

`HColumnDescriptor:` stores information about an HBase column family such the number of versions and it is used as input when creating a table or adding a column.

`HTable:` it is used to communicate to a single HBase table.

`Put:` is used to perform Put operations (create or update) for a single row.

`Get:` is used when the user wants to get all information about a single row.

`Scan:` it is the same as Get operation but it is performed on multiple rows. An optional StartRow and StopRow may be defined.

2.5 Summary

In this chapter we presented a global description of NoSQL datastores and a possible classification based on their data models. We also showed how NoSQL datastores are different from traditional RDBMS.

We presented two particular databases, DynamoDB and Google BigTable, and we showed why they are considered different from CAP Theorem's (cfr 2.2.1) point of view, due to their different architectural characteristics. In the final section we presented a particular column-oriented datastore, HBase, in union with its main features, as important starting point of our work.

Chapter 3

Denormalizing data starting from a relational query schema

The previous Chapter shows how NoSQL databases do not provide strict structures to store data, as RDBMS do. This lack affects performance of queries that need to perform a Join-like operation between different tables in a NoSQL datastore.

This Chapter proposes a possible solution to this problem. In particular, it introduces two different mapping approaches that denormalize data from the relational data model, predisposing it for a NoSQL environment. These mapping solutions are based on queries known at design time. After this introduction, the second part of the Chapter proposes a model for the first approach, analyzing it adaptability to all possible Relational Algebra queries.

This model is also supported by two examples and, in the last section, we are going to analyze its limits.

3.1 Definition of the problem

As Chapter 2 shows, NoSQL datastores are used in front of high scalability and performance requirements. They also do not support relationships as RDBMS do. Operations like Cartesian product and join are provided as standard operations in RDBMS, but they are not available in NoSQL datastores.

NoSQL datastores are often used in scenarios in which large amounts of data are stored before being processed and analyzed. The queries the user is going to perform are often known at design time, before storing all data. Most of queries performed in Big Data scenarios are read queries, performed on different tables through basic standard logic query operators: union, join, selection, projection, etc. .

This work offers a solution for these types of scenarios. In particular, it identifies different approaches to map a relational dataset structure into a NoSQL-adaptable one. This objective is achieved given the E-R schema and the queries known at design

time. The main aim of this work is to partially resolve NoSQL lack of performing join operations, maintaining their important features like high availability, horizontal scalability and partition tolerance.

The analysis of strong consistency, high availability and node sharding policies of NoSQL databases guides us to two different approaches.

In the following section we are going to present them, while in the second part of the Chapter we are going to present a specific model for the first of them, trying to analyze its adaptability to all possible Relational Algebra queries and to identify its limits.

3.2 Possible approaches for data denormalization

This section shows two possible approaches identified for data denormalization from a relational entity structure into a NoSQL-adaptable one.

Starting from the analysis of strong consistency and availability requirements, we achieve these two possible mapping solutions.

The scenario we are going to consider is the one where the user has a relational data model and wants to map it in a NoSQL datastore, in order to exploit its features. We also suppose that the user knows the queries he is going to perform and that they are given at design time. We also suppose that the dataset is primarily designed for read queries, as nowadays BigData datasets are.

We also point out that this denormalization process does not have to lose any information about data structure or data relationships. The final dataset must be formally equal to the one given as input.

We now present two different approaches, achieved from the need to obtain different features:

1. The first approach wants to exploit strong consistency property, provided by NoSQL datastores on single row write operation. This property prompted us to find a way to store all data related by a query in a single row of the new datastore.
   So the new designed entity will collect all data from the entities involved in a single query. This entity is designed in this way in order to be better predisposed to answer to that particular query, probably reaching better performance. The point this approach wants to solve is the lack of NoSQL datastores to perform operations between tables, using data structuring to perform them at design time and avoiding multiple scans or other expensive operations at run-time.

2. The second approach is conceived to solve availability requirements. This approach, like the first, wants to denormalize data, starting from a relational

schema of a dataset, in order to adapt it for a NoSQL-adaptable schema, given a specific query. However, this time, all data requested by the query are mapped into multiple rows.
This approach does not allow us to exploit strong consistency property of NoSQL datastores. However, in order to reach the highest possible availability, it needs to guarantee that all related rows of data are stored in the same physical space.
This model opens to further considerations about arranging database nodes before its population, managing insertions of data and maintaining consistent information in the same table node. This particular point is addressed by section 4.1.4.

We propose now a possible solution for the first approach, analyzing also its limits, while the second approach is left as a possible future work.

3.3 Basic assumptions

Before presenting our solution to the first approach, it is necessary to introduce some assumptions that can be useful to better understand the concepts explained in the next sections.

First of all it is necessary to introduce Relational Algebra, as basis of global relational query language. In a second phase we are going to introduce few operators that will be used to define our model's equations.

3.3.1 Relational Algebra

According to its definition [1], "Relational Algebra is a procedural language, based on algebraic concepts. It consists of a collection of operators that are defined on relations, and that produce relations as result".

It provides the main theoretical foundation for RDBMS, in particular for their query languages. The basic operators of Relational Algebra are five: set union, set difference, Cartesian product, projection and selection. The first three operations must involve two operands, while the last two need only one. There are also other operators that can be derived from these six primitive operators. They can be defined as follows:

UNION: $A \cup B = \{x | x \in A \vee x \in B\}$. "All data that belongs to entities A or B, is selected". The operands must be union-compatible, means that A and B must have the same set of attributes.

DIFFERENCE: $A - B = \{x | x \in A \wedge x \notin B\}$. "All data that belongs to entity A but not B, is selected". Like union operands, A and B must be difference-compatible.

CARTESIAN PRODUCT: $A \times B = \{x, y | x \in A \wedge y \in B\}$. "If entity A has n tuples and B has m tuples, the result is the combination of n and m tuples". Cartesian product relations must have different attribute subsets.

SELECTION: $\sigma_F(A) = \{x | x \in A \wedge F(x) = T\}$. Given a condition F on A, it returns a subset of tuples that respect the condition.

PROJECTION: $\pi_y(A) = \{x[Y] | x \in A\}$. It selects just few attributes of an entity to be shown.

An additional operator frequently used in relational database queries:

NATURAL JOIN: $A \bowtie B = \sigma_F(A \times B) = \{x \cup y | x \in A \wedge y \in B \wedge Fun(x \cup y)\}$. "If A and B shares one attribute name, the result is a join of the two entities, based on this shared attribute". It is probably the most used operator when a query is generated. In this work we will prefer this one instead of the Cartesian product, that is less frequently used. If the selection is performed on attributes with different names, the Join takes the name of "THETA JOIN", where Theta (Θ) is the selection condition.

There is an another operator that is used in basic relational algebra, that is

RENAME: $\rho_{y \leftarrow x}(A) = \{x[Y/X] | x \in A\}$. It simply renames the attribute's name; it is often used before union or difference operators to have equal attributes names. We will not consider it as a proper operation, by the fact that it does not imply significant changes to data stored in tables.

All operations work on one or more relations and the result is always a relation. This property is called *closure property* and it allows nesting expressions, like in arithmetic.

3.3.2 Operators and symbols

In this section are presented some operators that can be useful for the comprehension of the rules proposed in the next section. They are defined on purpose to better understand the model we are going to introduce. The main symbols used are:

- A: a capital letter is used to assign a name to a table; a table is a collection of rows stored in a database (a single row of a table is also called *tuple*).
- $\Sigma attA$: it is the sum of all the attributes of table A, except the primary-key attribute;

- $A := B$: given tables A and B, it means that all the tuples of table B have to be included into table A. This symbol has different aspects that must be explained:

  - If A has the same attribute names of B, no more attributes will be created;
  - If A has different attribute names from B, in A will be created attributes with the same name of attributes of table B;
  - $\pi_{C,D}(A) := \pi_{E,F}(B)$: "extract only attributes E and F from all the attributes of table B and put them into table A under the attribute's names C and D". Attribute list cardinality has to be the same in both of the expression operands. Again, if table A already has C and D attributes, they will not be created. On the other hand, attributes E and F for table B must already exists;
  - If B is composed by a single tuple, only a single tuple is inserted into A;
  - if both A and B are a single tuple, tuple B is copied into A. In this case if A already has attribute values filled, they will be overwritten.

- $A := [B]_{valid=bit}$: it is a shortcut symbol that is used only when we want to assign the bit value "bit" into the attribute "valid" in all the tuples of A included from B. We will better explain the attribute `valid` in section 3.4.2. From this definition we can deduce the following rule, useful when multiple assignments are made on the same value:

$$A := [(B)_{valid=x}]_{valid=y} \implies A := [B]_{valid=z} \, where \, z = x \wedge y$$

We also point out few restrictions to our entity model, in order to be sure that the rules we are going to explain will be well posed for all possible cases. They are the following:

- Attributes can not be in complex form;
- Entity generalization concept is not supported.

Attributes in simple form In a general E-R schema, attributes can be presented as complex values. They can be two or more aggregated values, one nested in each other. Each one of them can be transformed in a simple value, simply following the transformation schema explained in the following figure:

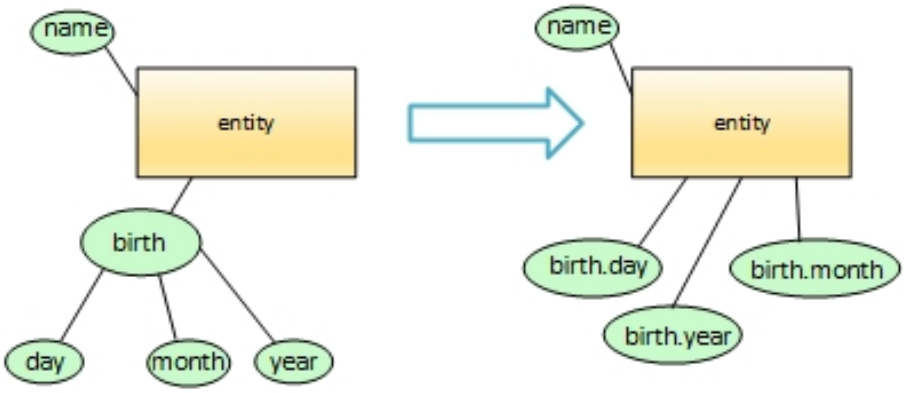


Figure 3.1: Transformation of nested attributes

No Entity generalization The concept of entity hierarchy can be resolved sharing parent entity attributes among child (children) entity (entities) attributes, as figure 3.2 explains.

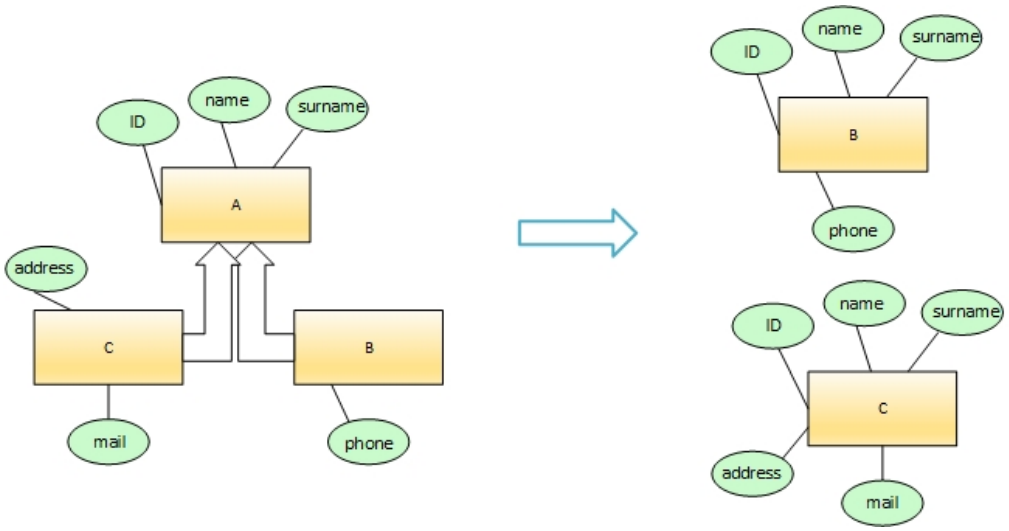


Figure 3.2: How to prevent from entity generalization

3.4 The model

In this paragraph we are going to explain our idea of denormalizing data, using an E-R query schema as input, and organizing the new data schema according to the queries that will be expressed on the data, trying to use the approach briefly explained previously (3.2).

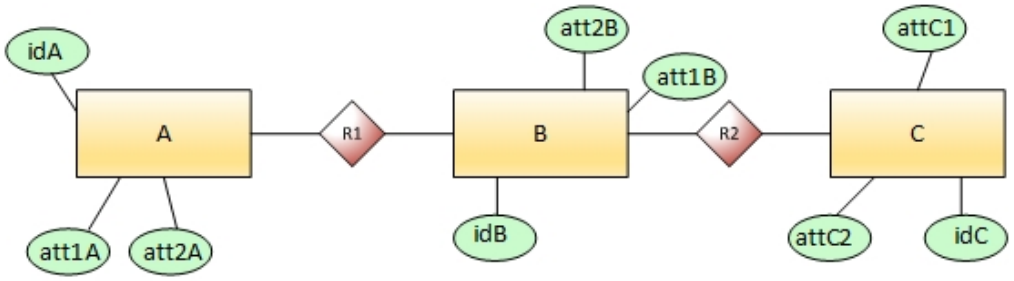


Figure 3.3: Entity-Relation (E-R) schema of a database

3.4.1 Input schema

The main idea is to conceive a model that, given an entity-relationship schema that represents the structure of the database (like the one presented in figure 3.3) and a query schema of it, builds an output entity that collects denormalized data on the basis of that specific query schema.

A query schema can be defined as a sub-model of the entity-relations schema. On the basis of a single E-R schema, different query schemas that involves different entities or relations can be constructed. For example, on the basis of the E-R schema described in figure 3.3, we can provide the query schemas described in figures 3.4 and 3.5.

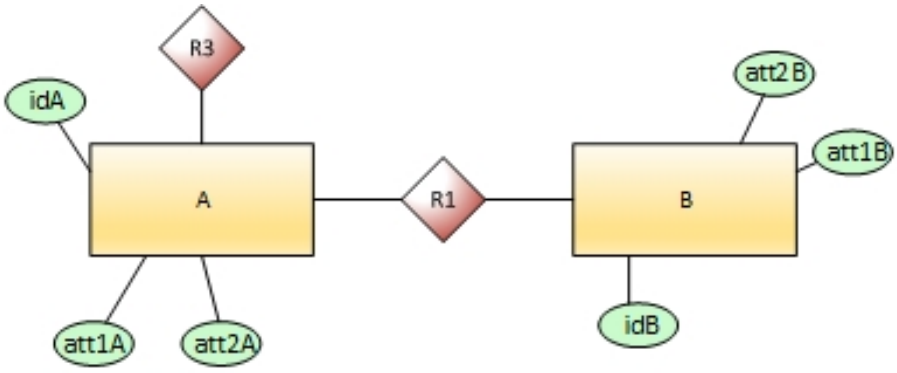


Figure 3.4: First relational query schema

As we can see they are built form the same E-R schema but they can involve different entities and relations. In addition, a query schema does not represent a single specific query, but it can represent many. For example the query described by figure 3.22 can be simply $A \bowtie B$ or it can be more complex as $\pi_{idA,attA1,attA2}\big(A \cup \rho_{idA \leftarrow idB, attA1 \leftarrow attB1, attA2 \leftarrow attB2}(B)\big)$. This is due to the fact that each relation can represent different operators.

The relation "R3" in figure 3.4 represents a relation on a single operand, like a selection or a projection.

Relations between two entities can be union sets, difference sets or join sets.

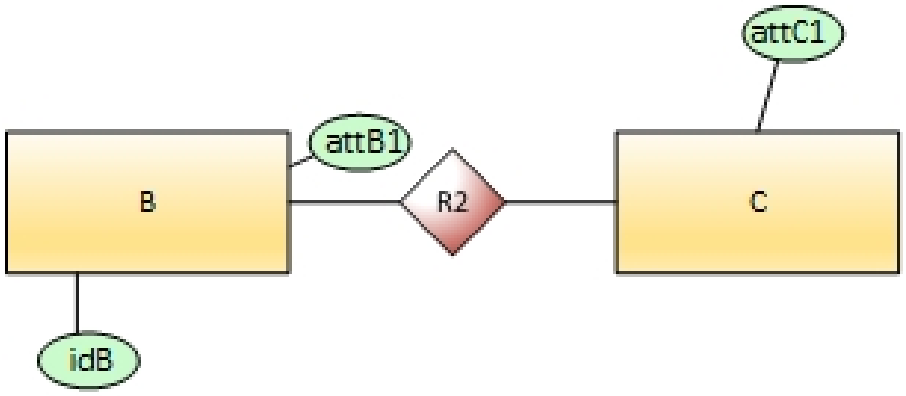


Figure 3.5: Second relational query schema

Each single query schema does not involve the whole database and all query schemas performed on a single database can produce different possible overlapping sub-models.

Given this introduction about query schemas, our main purpose is now to build a common single entity where all original data can be stored after having been denormalized on the basis of a single query schema.

3.4.2 Output entity

The final entity we are going to build has to be best-adaptable for all possible query schemas that can be performed on the initial E-R schema, taken as input. This final entity is called "Q" and it can be deducted from the starting E-R schema with few rules that we are going to explain in the next paragraph. These rules should fit to all possible query schemas, they have to maintain all the original information about the entities and the relationships and they do not have to cause any loss of data. The final single entity is something like the one presented in the figure below:

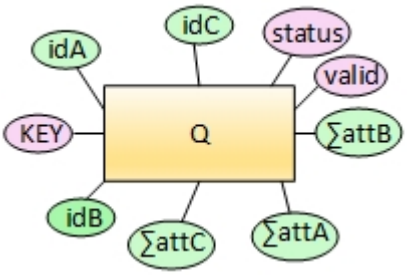


Figure 3.6: Single entity result "Q"

This new entity will have different attributes:

- `idA`, `idB` and `idC` are the same attributes as the original ones;
- $\Sigma attA, \Sigma attB$ and $\Sigma attC$ are all other attributes from A, B and C;

- `KEY` is the new key-value (its generation method will be discussed in chapter 4);

- `valid` is a single bit attribute used for store data validity, so this bit notifies if a row of data is valid for a query: "1" if yes, "0" if not. Since this process of denormalization should not cause data losses, also the rows that do not match the result of the query have to be stored in this entity. This attribute is used to distinguish the valid ones among those are not valid. Every single query performed on an entity like this has to return only the rows that have bit valid equal to "1". All the other rows are stored in the database to not lose data and information, but they do not satisfy the query request.
  If two operations are performed consequently, the final `valid` attribute is equal to the AND operation between `valid` bits of single operations.

- `status` : this attribute is composed by a set of bits that are predisposed to be used by the application layer to correctly show the attributes that the user wants to see. It is a sort of mapping between all the attributes of entity Q (3.4.2) and the attributes the user want to see. We provide this attribute instead of replicating the same row, the first with the attributes the user requires, the second with all the others. From a certain point of view it is a translation of *projection* operator. Its length is equal to the total number of the attributes (excluded `KEY`, `valid` and `status` itself), and each bit position corresponds to the attribute that has the same position in the attribute list.
  For example, an hypothetical entity A has three attributes, `att1`, `att2` and `att3`; we suppose `status` attribute value equal to "110". Mapping the first bit with `att1`, the second with `att2` and the third with `att3`, the final user will only get attributes with bit equal to "1". Hence, in this example, only `att1` and `att2` values will be returned.

We have to say that `KEY, valid` and `status` attributes will never be sent to the final user, but they can be used by the application layer to understand which rows and columns have to be returned as query answers. The `KEY`, as we will see, will be designed in order to identify the set of rows that are denormalized in order to answer to that specific query. With the bit `valid` it is possible to distinguish the "valid" rows from those that do not answer to the query, while `status` attribute is useful to understand which attributes have to be selected.

3.4.3 Equations

Given this notions we can proceed to explain the rules that allow us to switch from the canonical ER representation to the one presented previously and called "Q". For each single operation we will present a simple example, in order to better

explain the concept. Referring to figure 3.4, we can say that a relationship of a query schema can have one or two operands. We focus now on relation "R3", with only one operand. In particular it can be:

1. SELECTION ($\sigma_F(A)$): in this case, all the tuples of A are stored in Q, the ones that respect the condition F are copied with a "valid" attribute equal to "1", the others with "0". This choice is made to be able to distinguish the rows that answer correctly to the query from all the others, without losing any data:

$$Q := [\sigma_F(A)]_{valid=1} \cup [A - \sigma_F(A)]_{valid=0}$$

The figure below shows an example of this operator on table A:

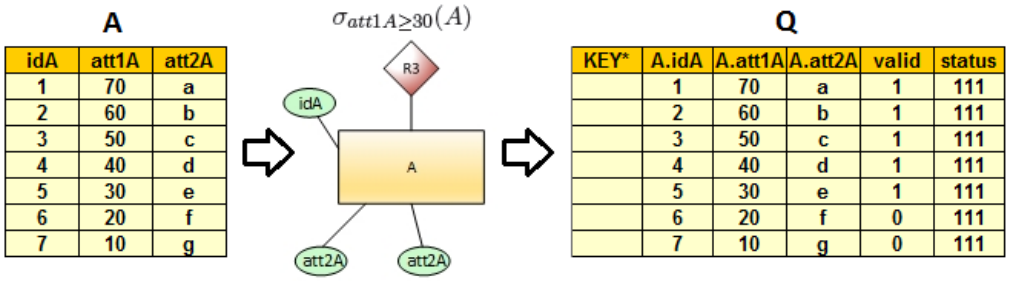


Figure 3.7: Selection example

KEY attribute is empty because its particular design will be better explained in Chapter 4.

2. PROJECTION $\pi_{attA}(A)$: in this case there is not an horizontal table splitting but only a selection of the attribute the user wants to be shown. Since all the values of all the attributes from the original tables must be stored into "Q", this operator will be replaced by `status` value, as we already have explained in paragraph 3.4.2.
Below is explained an example of this operation:

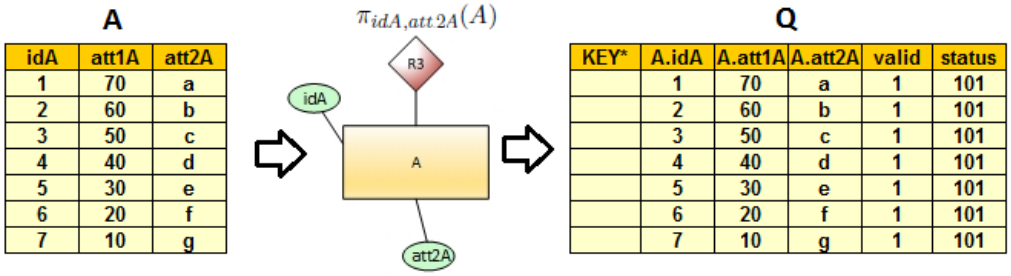


Figure 3.8: Projection example

We will see that Q maintains the *closure property* of Relational Algebra, that means that Q can be used as operand of another query. This opens to the possibility to perform a query composed of two projections. This particular case has already been addressed in section 3.4.2.

We explained all possible query operations that involve just one operand. We are now going to explain queries with two operands.

Regarding relation "R1" of figure 3.4, we notice that it has two operands, so it can have three different possibilities:

1. It is a UNION ($A \cup B$), so "Q" contains all the tuples stored in both A and B, and all of them are set to be valid:

$$\pi_{A.idA,\Sigma A.attA}(Q) := \left[\pi_{idA,\Sigma attA}(A)\right]_{valid=1}$$

$$\bigcup$$

$$\pi_{B.idB,\Sigma B.attB}(Q) := \left[\pi_{idB,\Sigma attB}(B)\right]_{valid=1}$$

   Example 3.9 shows how all the tuples are marked with valid bit equal to "1":

2. It can be a DIFFERENCE ($A - B$), so in Q only the rows of A that are not stored in B are marked as valid, all the others are marked with "0":

$$Q := \left(A - \rho_{idA,\Sigma attA \leftarrow idB,\Sigma attB}(B)\right)_{valid=1}$$

$$\bigcup$$

$$Q := \Big(A - \left(A - \rho_{idA,\Sigma attA \leftarrow idB,\Sigma attB}(B)\right)\Big)_{valid=0}$$

$$\bigcup$$

$$Q := (B)_{valid=0}$$

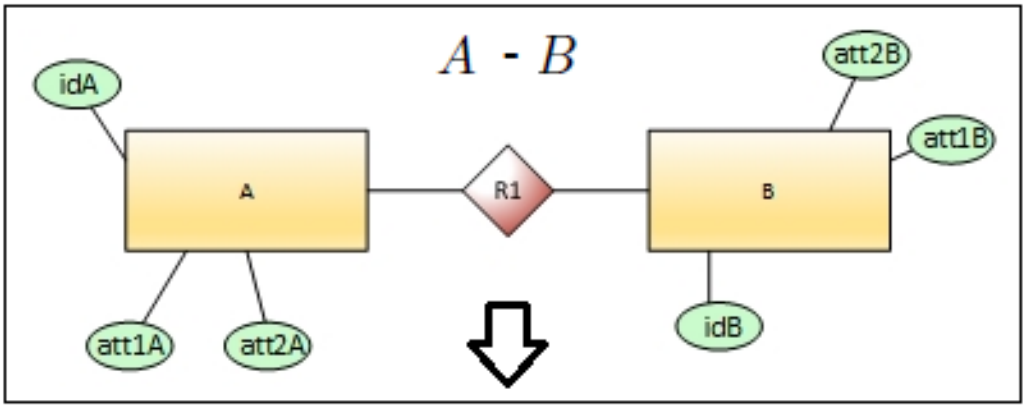
   In this case entities A and B must have the same attribute cardinality and names, in order to be able to execute the difference operation, as shown in example 3.10.

A

idA	att1A	att2A
1	70	a
2	60	b
3	50	c
4	40	d
5	30	e
6	20	f
7	10	g

B

idB*	att1B	att2B
1	70	a
2	60	b
3	40	d
4	40	d
5	20	e
6	10	f



Q

KEY*	A.idA	A.att1A	A.att2A	B.idB	B.att1B	B.att2B	valid	status
	2	60	b				1	111111
	3	50	c				1	111111
	5	30	e				1	111111
	6	20	f				1	111111
	7	10	g				1	111111
	1	70	a				0	111111
	4	40	d				0	111111
				1	70	a	0	111111
				2	60	b	0	111111
				3	40	d	0	111111
				4	40	d	0	111111
				5	20	e	0	111111
				6	10	f	0	111111

Figure 3.10: Difference example

3. If R1 is a JOIN ($A \bowtie B$), we have to distinguish between 3 cases:

   (a) Simple relation: the cardinality of the relationship is 0:1 or 1:1 on both hands; it means that A (or B) is linked to at least zero, or one and at most one, item of B (or A). So in Q " only the tuples that represent a join between the two entities will be included as "1, as all the others are marked with "0":

$$Q := (A \bowtie B)_{valid=1}$$

$$\bigcup$$

$$Q := \big(A - \pi_{idA,\Sigma attA}(A \bowtie B)\big)_{valid=0}$$

$$\bigcup$$

$$Q := \big(B - \pi_{idB,\Sigma attB}(A \bowtie B)\big)_{valid=0}$$

$$A \bowtie B$$

   (b) In this case as "Join" we mean a "Natural Join", that implies that A and B entities must share an attribute name (it can be any of their attributes), as figure 3.11 shows. The definition explained above is also used for "Theta Join" and all the the other Join-derived expressions. They are composed by one or more operations, so they can be performed executing each single operation in cascade.

   (c) The relationship R1 is a one-to-many relation: each item of A (or B) can be linked to 0 to n item of B (or A), while each item of B (or A) can be linked with 0 or 1 item of A (or B).
   In this particular case the solution we try to adopt is quite different from the one we already proposed.
   These type of relations can be seen in two different ways, depending on which side of the relation we want to consider. We can in fact denormalize n item of B in relation with an item of A or, on the other side, we can store into Q at most one item of A related with a single item of B.
   The solution is highly dependent on the scenario the one-to-many relation is applied to: sometimes the first way of denormalization can be the best, sometimes it would be better to use the second.
   Before showing the equation for this type of relation, we now present an entity Q that is slightly different from the one we presented in the previous paragraph.
   Since there is the possibility of mapping all n entities of B in relation with an entity of A all in the same row of Q, we have to make some changes to its structure. The result of this change can be seen in figure 3.12, where

it is shown how a single entity of A can be stored in the same row with many items of B, as demonstrate the presence of multiple instances of `idB` and $\Sigma attB$.

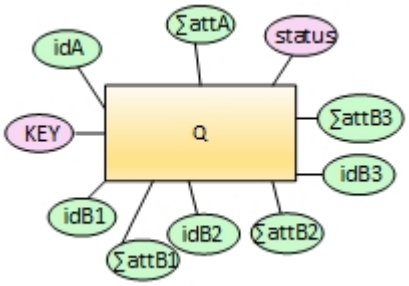


Figure 3.12: One-to-many entity Q

The formula to obtain this entity is the following:

Algorithm 3.1 One-to-many join equation

$Q :=\Big($
for each $a \in A$
new $q \in Q$;
$\pi_A(q) := \pi_A(a)_{valid=0}$
for each $b_i | b_i \in B \wedge (b_i \bowtie a) \neq \emptyset$ **do**
$\pi_B(B_A) := \pi_B(b_i)$
$\pi_{Bi}(q) := \pi_B(b_i)$
$\pi_{valid}(q) =' 1'$
update status field
end for
add q
end for $\Big)$
$\bigcup$
$Q := \pi_B(B - B_A)$
\\WHERE $B_A$is a support table composed of all the tuples of B that are in join with at least a tuple of A

Algorithm 3.1 means that: for each item of A, all linked B instances are taken, pushed in the same row (`"q"`) and stored in Q entity. The whole row has a bit "valid" value set to '1', but the status field can hide, if necessary, the values of a `'b'` item located in the row. This is expected in case of a previous operation on `'b'`, that excludes it from the final result. Now we present the different scenarios in which a one-to-many relation can appear, and we are going to analyze each of them, supplying its solution for each one.

i. There is only a OneToMany relationship involved in the query:

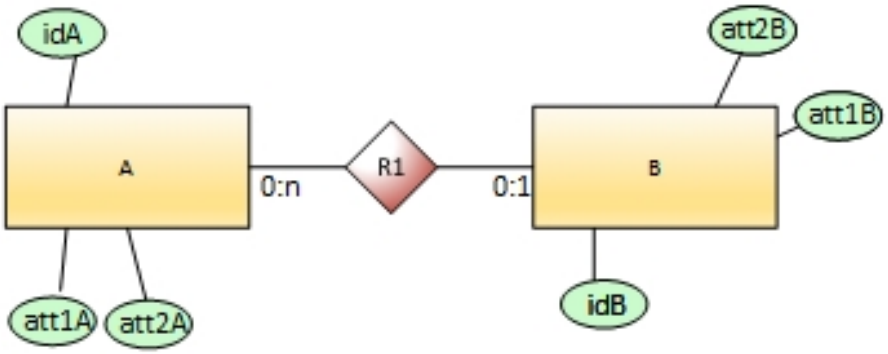


Figure 3.13: OneToMany entity schema

This is the easiest scenario. We have to decide if it is the case to map every item of B related with one item of A into Q, or to map each duplicate copy of A with at most one of B. There is a trade-off in the choice; it can be guided by different motivations: physical storage optimization, relationship dimensions, or the number of attributes in table A can induce us to choose one or the other depending on the circumstances.
When the designer does not focus on disk optimization, the items of table B are connected with a few items of table A or the number of attributes of items in table A is not very large, can be a good solution to duplicate the items of table A with each related item of table B into a single row of entity Q.
If, on the other side, the dimension of the rows of table A is very big or each item of table B is related to a very huge number of item of table A, it will be better to store, into each row of the final entity Q, a row with all the items of table B in relation with one item of table A, structured like in figure 3.12.
In our work, we do not focus physical space optimization: we are denormalizing data in order to represent it on the basis of a query model, so we claim that data replication and big disk space are acceptable. We also said that the main purpose of this denormalization process is to find a way to store all the possible related data together into a single row.
Starting from this point of view we decided to exploit the option of mapping all items of table B with a single item of table A into a single row of entity Q. The only case that can be excluded from this solution can be when table 'A' has only one or at most two attributes (in addition to the key attribute) and it is not related with any other table. In this situation entities contained in 'A' can be considered as

a single attribute of table B and its content can be replicated for each item of table B.
We will see in the Chapter 5 a single scenario of this type.

ii. There are two or many one-to-many relations arranged in a cascade:

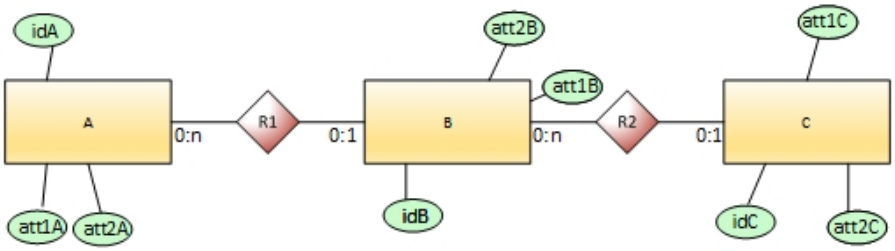


Figure 3.14: Two OneToMany relations in cascade

this scenario can be the natural consequence of the first. The result will be a row of entity Q that will collect a single item of table A, all the item of table B related to it, and all the items of table C related with those items of table B. Again, if the number of attributes of the table are limited to one, they can be duplicated and stored with the item of the table they are in relation with. For example if A had only `att1A` attribute and B had only `att1B` attribute, first of all we would duplicate each occurrence of item of table A in the same "Q row" with each related item of table B. Since B had two attributes, we would duplicate them with their related item of table C in the final row of entity Q.

iii. There are two one-to-many relations but this time they are not in a cascade situation:

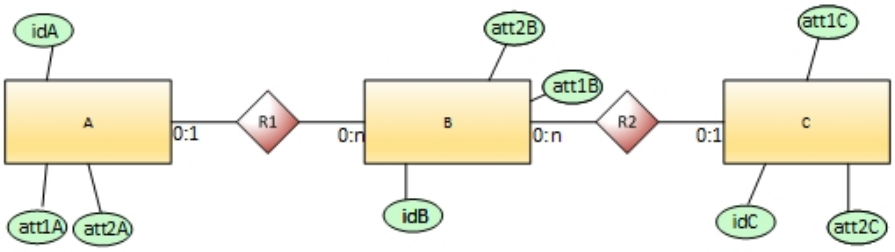


Figure 3.15: Two OneToMany relations situation 2

again also in this situation, we apply what we said in the first point: items of tables A and C, related with items of table B, are mapped with it into a unique row of entity Q with one item of table B and many 'a' and 'c' items.

iv. There are, again, two one-to-many relations, arranged like in the figure below:

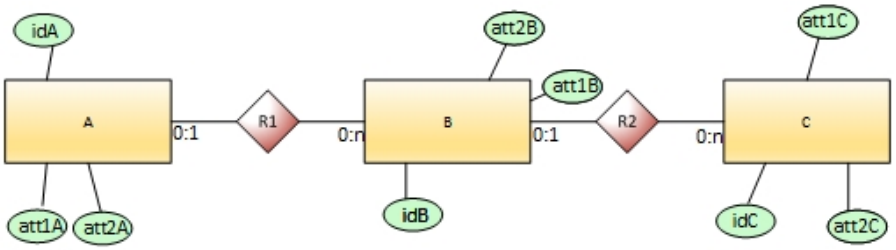


Figure 3.16: Two OneToMany relations situation 1

In this scenario, table B can be seen as a bridge-table, more frequently seen in a many-to-many relations. The approach here can be different, and we delay it to the point c) of this list, where we are going to analyze many-to-many relations.

(d) R1 is a many-to-many relation. Its representation is shown in figure 3.17.

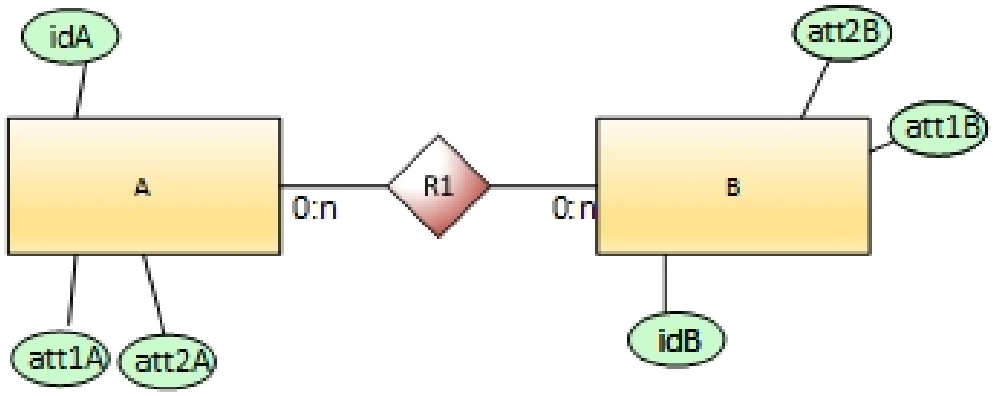


Figure 3.17: Many-to-many entity schema

As we can deduct and already said, NoSQL datastores are not the ideal tool to represent this particular situation. The main problem is that Scan queries on these type of datastore can be performed only on the `KEY` attribute, so it can not be possible to optimize the query for each of the two entities involved, as we can do in RDBMS.
We propose here three possible mapping solutions, describing an example for each of them. For clarity, we remove `KEY`, `valid` and `status` attributes from the tables of figures 3.19,3.20 and 3.21 , in order to better focus on the denormalizing solution of the many-to-many relation.

- Data replication among the copies of same data has to be limited to the minimum, in order to reach consistency requirements, and to build a second table for storing essential information about the other

side. Many-to-many relations can be seen as two different one-to-many relations, so it can be decided to represent one of the two with the method explained at the beginning of point b), and to represent the other relation with a table that collects only the essential key values of all data, leaving to designers the task to get the correct values when this relation is performed.
Figure 3.18 and figure 3.19 describe an example that adopts this mapping approach. The original tables are represented in figure 3.18 (where the table on the right represents the many-to-many relation). The first table (1) in figure 3.19 represent one side of the relation: all rows from table A are listed with the entities of table B they are in relation with. The second table (2) represents the opposite relation. As we can see, in the last one are collected only the `id`'s of the items of table A. Performing a query on this table may not be enough to get all requested information, so there could be the possibility of using another Scan or Get operation.

A

idA*	att1A	att2A
1	70	a
2	60	b
3	50	c
4	40	d
5	30	e
6	20	f
7	10	g

B

idB*	att1B	att2B
1	3	a
2	4	b
3	5	d
4	6	d
5	7	e
6	3	f

$A \bowtie B$

idA	idB
5	3
4	4
1	4
2	1
3	4
5	2

Figure 3.18: many-to-many example tables

(1)

KEY*	A.idA	A.att1A	A.att2A	B1.idB	B1.att1B	B1.att2B	B2.idB	B2.att1B	B2.att2B
	1	70	a	4	6	d			
	2	60	b	1	3	a			
	3	50	c	4	6	d			
	4	40	d	4	6	d			
	5	30	e	2	4	b	3	5	d
	6	20	f						
	7	10	g						

(2)

KEY*	B.idB	B.att1B	B.att2B	A1.idA	A2.idA	A3.idA
	1	3	a	2		
	2	4	b	5		
	3	5	d	5		
	4	6	d	4	1	3
	5	7	e			
	6	3	f			

Figure 3.19: Many-to-many denormalization - first approach

Adopting this kind of solution can be a good idea for limiting data replication of all attributes of the same copies of data into the data-store, avoiding consistency problems after following operations on these data, but it can produce bad performances when the second query is executed, due to the fact that only necessary attributes are stored and it must be necessary to scan all the two tables created to have a complete answer to that query.

- The second scenario is quite similar to the previous one, but it differs for the fact that are collected in the second table not only the keys, but also all attributes and data necessary to answer the query.
  With this scenario a second copy of data collected in the first table can be stored, causing possible future problems with consistency among the different copies of the same rows.
  Figure 3.20 shows this mapping solution, taking as input the two tables of figure 3.18. The first table is the same as the one of the first approach (cfr 3.19). The second differs for the fact that it collects not only the `id`'s of the items of table A, but also all the attributes.

(1)

KEY*	A.idA	A.att1A	A.att2A	B1.idB	B1.att1B	B1.att2B	B2.idB	B2.att1B	B2.att2B
	1	70	a	4	6	d			
	2	60	b	1	3	a			
	3	50	c	4	6	d			
	4	40	d	4	6	d			
	5	30	e	2	4	b	3	5	d
	6	20	f						
	7	10	g						

(2)

B.idB	B.att1B	B.att2B	A1.idA	A1.att1A	A1.att1A	A2.idA	A2.att1A	A2.att2A	A3.idA	A3.att1A	A3.att2A
1	3	a	2	60	b						
2	4	b	5	30	e						
3	5	d	5	30	e						
4	6	d	4	40	d	1	70	a	3	50	c
5	7	e									
6	3	f									

Figure 3.20: Many-to-many denormalization - first approach

Unlike the first approach, this one can catch up better performances when the second query is executed. The choice between the two possibilities can be dictated by the structure of data model or performance and consistency requirements, preferring minimum replication over better performances or viceversa.

- The third approach is the one that we are going to use for the practical work in the next chapters and it is quite different from the other two points.
  One of the assumptions we made in 3.2 paragraph was that we need to design the new entity model on the basis of the query schema(s)

we know at design time, possibly reaching better performances on its execution.
Starting from this point of view we allow strong data replication in order to have data better stored and to answer client's requests in the shortest possible time. We can now consider the many-to-many relation as it is represented in relational databases. In RDBMS it can be described as a bridge-table (or *join-table*), where are collected the essential information about the two entities that belong to the relation. Figure 3.21 describes this approach, always on the basis of the tables given by figure 3.18. As we can see, the central columns of this table are the same as the Join Table of figure 3.18, with all other attributes of the two entities stored in the same row.

A.att1A	A.att2A	A.idA	B.idB	B.att1B	B.att2B
30	e	5	3	5	d
40	d	4	4	6	d
70	a	1	4	6	d
60	b	2	1	3	a
50	c	3	4	6	d
30	e	5	2	4	b
20	f	6			
10	g	7			
			5	7	e
			6	3	f

Figure 3.21: Many-to-many denormalization - third approach

From this point of view we can store in our entity Q not only the key attributes of the entities, like in a relational approach, but also all the other attributes and values that belongs to that two entities. This approach is quite expensive from disk space usage perspective (if table A has |n| rows and B has |m| rows, the join table could have until |m*n| rows), but all data are better designed to answer to user queries, allowing a possible substantial improvement of performances. We can also say that, if the query does not require the entire relation (for example, there is a selection before it), will be replicated only the tuples requested, leaving all the others as they are in the original bridge-table.

This situation is probably the main problem all designers have to deal with when a not relational datastore is chosen. Due to their structure they are not well designed to collect data structured as equal as in a relational database (They are called "not relational" exactly for this reason!).

3.4.4 Examples

The equations proposed in the previous paragraph maintain the important property of closure already explained in 3.3.1 section: using these formalisms we are able to combine them to respond to all possible composed queries.

It's important to highlight the fact that after every single operation (union, difference, join or select), an instance of Q is created. The sequent operation of the query can be executed directly on this entity, and not on the original operands of the first operation. Below are presented two examples of nested queries, using different entities and two different relationships.

3.4.4.1 Example 1

The query we want to perform is a subset of the one already seen and it is represented in figure 3.22.

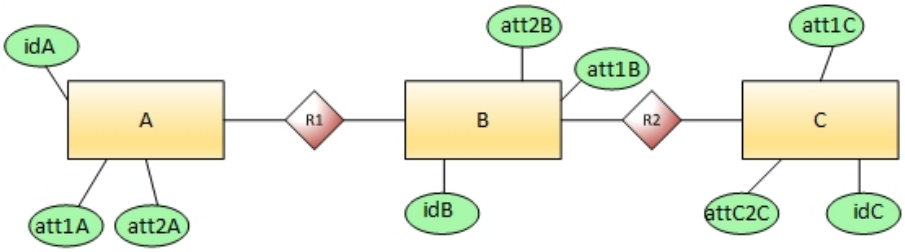


Figure 3.22: Example 1 query schema

The mentioned figure can represent a query like $A \bowtie (B \bowtie C)$. Assuming that R1 and R2 are one-to-one relations and the couples B-C and A-B both share an attribute name, we apply the equations we presented in 3.4.3 paragraph:

$$
\begin{aligned}
B \bowtie_{R2} C = \\
& Q := (B \bowtie_{R2} C)_{valid=1} \\
& \bigcup \\
& \pi_{B.idB,\Sigma B.attB}(Q) := \Big[\pi_{idB,\Sigma attB}\big(B - \pi_{idB,\Sigma attB}(B \bowtie_{R2} C)\big)\Big]_{valid=0} \\
& \bigcup \\
& \pi_{C.idC,\Sigma C.attC}(Q) := \Big[\pi_{idC,\Sigma attC}\big(C - \pi_{idC,\Sigma attC}(B \bowtie_{R2} C)\big)\Big]_{valid=0}
\end{aligned}
$$

A

idA*	att1A	att2A
1	70	a
2	60	b
3	50	c
4	40	d
5	30	e
6	20	f
7	10	g

B

idB*	att1B	att2B	idA
1	one	600	2
2	two	500	4
3	three	400	6
4	four	300	
5	five	200	
6	six	100	

C

idC*	att1C	att2C	idB
1	alfa	e	1
2	beta	d	4
3	gamma	c	5
4	delta	b	
5	epsilon	a	

Figure 3.23: Example 1: A, B, C input tables

So, if we have input tables A,B,C like in figure 3.23, the result of the previous assignment can be as described in the following figure. To a better understanding, in Appendix are attached also the other intermediate tables.

KEY*	idB	att1B	att2B	idA	idC	att1C	att2C	valid	status
	1	one	600	2	1	alfa	e	1	
	4	four	300		2	beta	d	1	
	5	five	200		3	gamma	c	1	
	2	two	500	4				0	
	3	three	400	6				0	
	6	six	100					0	
					4	delta	b	0	
					5	epsilon	a	0	

Figure 3.24: Example 1: entity Q after first Join

Now we can use the Q entity as input for the second Join operation, thanks to closure property. This and the next equations, used to obtain the result explained by the following formula and shown in figure 3.25, are appended in Appendix A.

The final result obtained is the subsequent:

$$
\begin{aligned}
A \bowtie_{R1} (B \bowtie_{R2} C) = \\
&\pi_{idA,\Sigma attA}(Q^*) := \Big[A \bowtie_{R1} (B \bowtie_{R2} C) \Big]^{(a)}_{valid=1} \\
&\cup \Big[A \bowtie_{R1} \big(B - \pi_{idB,\Sigma attB}(B \bowtie_{R2} C)\big) \Big]^{(b)}_{valid=0} \\
&\cup \Big[A \bowtie_{R1} \big(C - \pi_{idC,\Sigma attC}(B \bowtie_{R2} C)\big) \Big]^{(c)}_{valid=0} \\
&\bigcup \\
&\pi_{idA,\Sigma attA}(Q^*) := \Big[A - \pi_{idA,\Sigma attA} \Big(\Big(A \bowtie_{R1} (B \bowtie_{R2} C) \Big) \\
&\cup \Big(A \bowtie_{R1} \big(B - \pi_{idB,\Sigma attB}(B \bowtie_{R2} C)\big) \Big) \\
&\cup \Big(A \bowtie_{R1} \big(C - \pi_{idC,\Sigma attC}(B \bowtie_{R2} C)\big) \Big) \Big) \Big]^{(d)}_{valid=0} \\
&\bigcup \\
&\pi_{idB,\Sigma attB,idC,\Sigma attC}(Q^*) := \Big[\pi_{idB,\Sigma attB,idC,\Sigma attC} \Big((B \bowtie_{R2} C) \\
&\cup \big(B - \pi_{idB,\Sigma attB}(B \bowtie_{R2} C)\big) \\
&\cup \big(B - \pi_{idC,\Sigma attC}(B \bowtie_{R2} C)\big) \Big) \\
&- \pi_{idB,\Sigma attB,idC,\Sigma attC} \Big(\Big(A \bowtie_{R1} (B \bowtie_{R1} C) \Big) \\
&\cup \Big(A \bowtie_{R1} \big(B - \pi_{idB,\Sigma attB}(B \bowtie_{R2} C)\big) \Big) \\
&\cup \Big(A \bowtie_{R1} \big(C - \pi_{idC,\Sigma attC}(B \bowtie_{R2} C)\big) \Big) \Big) \Big]^{(e)}_{valid=0}
\end{aligned}
$$

In particular these equations represent:

(a) Tuples that belong to the entire query, marked with "1" as validity;

(b) Tuples of A that are not in join with B tuples from $B \bowtie C$ table, marked with "0";

(c) Tuples of A not in join with C tuples from $B \bowtie C$ table, marked with "0" ($\emptyset$);

(d) Tuples of A not in (a), (b) or (c), marked with "0";

(e) Tuples from B, C or $B \bowtie C$ that are not in join with A, marked with "0";

In the figure below is presented the Join between table A and table Q, that generates table Q*:

KEY*	idB	att1B	att2B	idA	idC	att1C	att2C	att1A	att2A	valid	status
	1	one	600	2	1	alfa	e	60	b	1	
	2	tow	500	4				40	d	0	
	3	three	400	6				20	f	0	
				1				70	a	0	
				3				50	c	0	
				5				30	e	0	
				7				40	d	0	
	4	four	300		2	beta	d			0	
	5	five	200		3	gamma	c			0	
	6	six	100							0	
					4	delta	b			0	
					5	epsilon	a			0	

Figure 3.25: Example 1: Q* table

3.4.4.2 Example 2

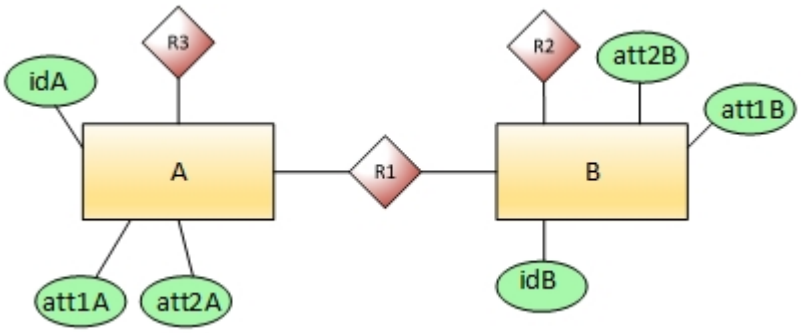


Figure 3.26: Example 2: query schema

This time we explain how a single operator works: taking figure 3.26 as E-R query schema, we can assume that R3 and R2 are two selections and R1 a union; that schema may produce this kind of query: $\sigma_{att1A \geq 40 \wedge att2B \geq 200}(A \cup B)$. Thanks to distributive property of selection over union, it can be transformed into the sequent query: $\big(\sigma_{att1A \geq 40}(A)\big) \cup \big(\sigma_{att2B \geq 200}(B)\big)$. Following the rules explained in 3.4.3 paragraph, we can get the following passages:
First of all we apply selection rules on both entities A and B:

$$Q_A := \big(\sigma_{att1A \geq 40}(A)\big)_{valid=1} \cup \Big(A - \big(\sigma_{att1A \geq 40}(A)\big)\Big)_{valid=0}$$

$$Q_B := \big(\sigma_{att2B \geq 200}(B)\big)_{valid=1} \cup \Big(B - \big(\sigma_{att2B \geq 200}(B)\big)\Big)_{valid=0}$$

Then we use union's rule to get the final Q entity:

$$Q = Q_A \cup Q_B =$$
$$\pi_{A.idA,\Sigma A.attA}(Q) := \Big(\big(\sigma_{att1A \geq 40}(A)\big)_{valid=1} \cup \Big(A - \big(\sigma_{att1A \geq 40}(A)\big)\Big)_{valid=0} \Big)_{valid=1}$$
$$\bigcup$$
$$\pi_{B.idB,\Sigma B.attB}(Q) := \Big(\big(\sigma_{att2B \geq 200}(B)\big)_{valid=1} \cup \Big(B - \big(\sigma_{att2B \geq 200}(B)\big)\Big)_{valid=0} \Big)_{valid=1}$$

and using assignment operator properties:

$$Q = Q_A \cup Q_B =$$
$$\pi_{A.idA,\Sigma A.attA}(Q) := \big(\sigma_{att1A \geq 40}(A)\big)_{valid=1} \cup \Big(A - \big(\sigma_{att1A \geq 40}(A)\big)\Big)_{valid=0}$$
$$\bigcup$$
$$\pi_{B.idB,\Sigma B.attB}(Q) := \big(\sigma_{att2B \geq 200}(B)\big)_{valid=1} \cup \Big(B - \big(\sigma_{att2B \geq 200}(B)\big)\Big)_{valid=0}$$

the representation of the final table is showed in Appendix A.

3.4.5 Considerations

In the previous paragraphs we showed the rules that generates the unique entity Q and we proposed two practical examples. In this section we want to focus on few aspects we don't mention in the previous section.

Join attributes

When we showed the Join rules, we presented them only for Natural Join, not for other types of Join. As we can see, in Relational Algebra we can have different kind of Join: left and right join, full join, cross join, inner and outer join, etc. . All this kind of operators can be somehow derived from original Cartesian product, as natural Join, so all the rules we showed for natural join can be used for all the other types of this operator. In fact they usually add a projection or a selection over the result returned by Cartesian product, so they can be considered as two nested operations. This statement implies also that Join on normal (not-key) attributes will produce a correct result.

In example 1 (3.4.4.1) we also considered two one-to-one relationships and we assumed that they were represented by idA attribute in B table and idB attribute in C table. We also have to notice that the rules proposed for join are the same also if we had an idB attribute in A table or an idC attribute in B table, as it can be possible according to E-R logical schema for relational databases.

Relationship attributes

Relationship attributes are particular attributes that don't belong to a specific entity but to a specific relation. In relational database design they are added as a new attribute of one of the entities that participate in the relation. So they can be considered in our work as an entity attribute, sharing all the comments we have already done for them. We will also see, in model evaluation section (Chapter6), an example of this situation.

Loop relationships

It may happen that a set of entities and relationships can generate a loop. This situation can cause different problems, especially if the relations involved are OneToMany or ManyToMany relations. Analyzing this type of situation can be quite hard and it may be dependent on the situation and the entities or relationships characteristics. The immediate solution to this problem can be the duplication of one of the entities involved in the loop, and it can be the less expensive in terms of disk space occupation and in number of attributes. Otherwise it can be possible to eliminate a relationship in order to replace it with a combination of other two that are part of the loop. We leave the choice to the E-R schema designer, assuming that the E-R schema given as input by our model do not includes loop relationships.

Self join-relationships

A self-join relationship is a Join where both operands are the same single entity. In fact, if union and difference operators show particular properties ($A \cup A = A$, $A - A = \emptyset$) for that, join operations can be done over different attributes of the same table, in order to create a particular new connection. Using the rules we presented, they can generate some copies of the same data, as the example below can show:

$$
\begin{aligned}
A \bowtie_{A.id=A.SonId} A = \\
& Q := (A \bowtie_{A.id=A.SonId} A)_{valid=1} \\
& \bigcup \\
& Q := \big(A - \pi_{A1.idA,\Sigma A1.attA}(A \bowtie_{A.id=A.SonId} A)\big)_{valid=0} \\
& \bigcup \\
& Q := \big(A - \pi_{A2.idA,\Sigma A2.attA}(A \bowtie_{A.id=A.SonId} A)\big)_{valid=0}
\end{aligned}
$$

that produces the following result on Q:

A

idA	name	SonId
1	a	8
2	b	8
3	c	9
4	d	1
5	e	10
6	f	
7	g	3
8	h	
9	i	
10	j	

Q

KEY*	A1.idA	A1.name	A2.idA	A2.name	A2.SonId	valid	status
	1	a	8	h		1	
	2	b	8	h		1	
	3	c	9	i		1	
	4	d	1	a		1	
	5	e	10	j		1	
	7	g	3	c		1	
	6	f				0	
	8	h				1	
	9	i				2	
	10	j				3	
			2	b	8	4	
			4	d	1	5	
			5	e	10	6	
			6	f		7	
			7	g	3	8	

Figure 3.27: Self join example

as we can see we have data replication on the same instances, that may cause problems of redundancy of equal values. A possible solution to this problem can be considering both the projections of A (one for A1, one for A2) together, and not separated:

$$
\begin{aligned}
A \bowtie_{A.id=A.SonId} A = \\
& Q := (A \bowtie_{A.id=A.SonId} A)_{valid=1} \\
& \bigcup \\
& Q := \Big(A - \big((\pi_{A1.idA, \Sigma A1.attA}(A \bowtie_{A.id=A.SonId} A)) \\
& \cup (\pi_{A2.idA, \Sigma A2.attA}(A \bowtie_{A.id=A.SonId} A)) \big) \Big)_{valid=0}
\end{aligned}
$$

in this case both A1 and A2 instances are considered only one time, avoiding data replication of same rows.

3.5 Limits

In this paragraph we are going to analyze the negative aspects that this mapping model can cause. The main limits identified are:

- the final entity Q is optimized to respond to a given specific query: the rules we explained in the previous section, as we have already said different times, maintain all information about the original data schema, all entity attributes and all relation attributes that convey to the query, but also to all the other entities not involved in it.
  So Q entity represents under a different way the same structure as the original

one. Thanks to this feature all possible queries can be executed on Q entities, having the same data results as we can have on the original ones.
Obviously they can be performed not just after one scan of the Q table, as the query used to design it, but probably after few and complicated researches. This can cost a loss of in terms of query performances and an increasing latency time. If the frequency execution time of these different queries is low and their cost acceptable, the designer can take into account this option.

- the model is designed for read-only databases; insert, update and delete queries are not taken into account in this analysis.
  We can say that any of these operations performed on a database organized following our model have to maintain its properties and characteristics, correctly answering to the query that generates it.
  We will discuss in the final chapter about future possible works and features linked to this thesis. One of them is to try to understand a possible way to perform insert and update executions on entity Q, without violate its characteristics.

3.6 Summary

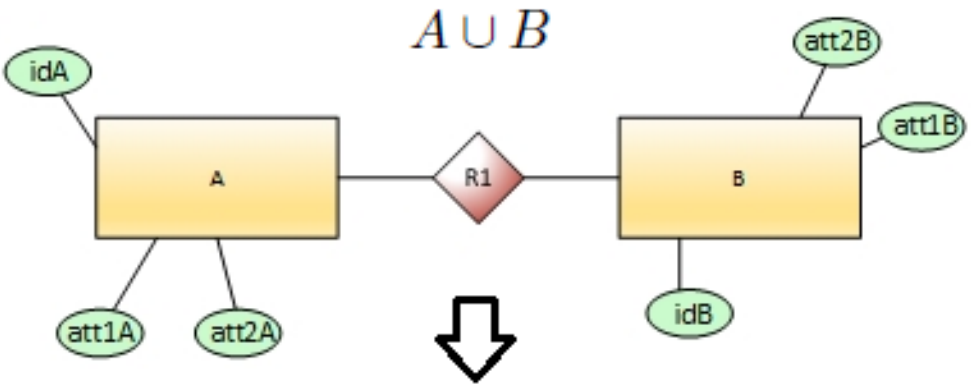
In this chapter we presented two alternative strategies to denormalize data from a typical relational structure into a NoSQL-adatptable structure. After few premises, we proposed a mapping solution to the first of these strategies and we looked over its advantages but also its limits. In the next chapter we will present architectural characteristics of HBase, a NoSQL datastore, and we will show how to map data from the model we proposed in this chapter to it.

A

idA	att1A	att2A
1	70	a
2	60	b
3	50	c
4	40	d
5	30	e
6	20	f
7	10	g

B

idB*	att1B	att2B
1	3	a
2	4	b
3	5	d
4	6	d
5	7	e
6	3	f



Q

KEY*	A.idA	A.att1A	A.att2A	B.idB	B.att1B	B.att2B	valid	status
	1	70	a				1	111111
	2	60	b				1	111111
	3	50	c				1	111111
	4	40	d				1	111111
	5	30	e				1	111111
	6	20	f				1	111111
	7	10	g				1	111111
				1	3	a	1	111111
				2	4	b	1	111111
				3	5	d	1	111111
				4	6	d	1	111111
				5	7	e	1	111111
				6	3	f	1	111111

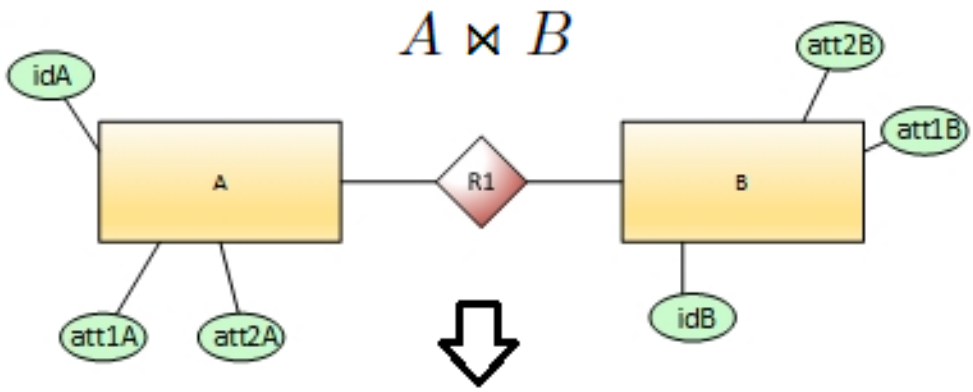
Figure 3.9: Union example

A

idA	att1A	att2A	idB
1	70	a	3
2	60	b	4
3	50	c	2
4	40	d	1
5	30	e	
6	20	f	
7	10	g	

B

idB	att1B	att2B
1	3	a
2	4	b
3	5	d
4	6	d
5	7	e
6	3	f



Q

KEY*	A.idA	A.att1A	A.att2A	B.idB	B.att1B	B.att2B	valid	status
	1	70	a	3	5	d	1	111111
	2	60	b	4	6	d	1	111111
	3	50	c	2	4	b	1	111111
	4	40	d	1	3	a	1	111111
	5	30	e				0	111111
	6	20	f				0	111111
	7	10	g				0	111111
				5	7	e	0	111111
				6	3	f	0	111111

Figure 3.11: One-To-One example

Chapter 4

Applying denormalization on HBase

This Chapter deals with the technical part of the thesis. In Chapter 3 we described two possible approaches of data denormalization from a relational structure into a NoSQL-adaptable one.

In particular, we proposed a model for the first of them.

We selected HBase, a columnar NoSQL datastore (shown in detail in Chapter 2), because it can guarantee both approaches requirements and we want to compare query performances between the model proposed and a relational structured one.

We will show the entire evaluation process in Chapter 6. In this Chapter we are going to show all intermediate phases from the end of the description of the model to the beginning of test execution.

In particular, the first section of this Chapter shows a deepening on HBase node splitting policies, focusing in particular on the possibility to design row-keys and organize pre-splitted HBase tables, in order to better organize data distribution among HBase regions. This analysis leads us to gain a solution adaptable for the two approaches we have discussed in 3.2.

After this analysis we are going to show the main parameters that need to be configured to install a full HBase cluster. We also provide in Appendix B, a brief explanation of a self-made Linux based script that, given few essential parameters, allows to automatically deploy a working HBase cluster.

Finally we show how data denormalization, showed in Chapter 3, is performed by introducing a proof-of-concept Java tool whose main classes are appended in Appendix C.

4.1 Optimize data distribution

In this section we are going to show the analysis conducted on the possibility of HBase to create predefined regions, integrated with studies on how to design an adequate key, in order to better optimize the data distribution in our database.

After brief theory concepts, we will propose two possible solutions adaptable for the approaches we showed in section 3.2.

4.1.1 HBase region splitting policy and pre-splitting methods

As Enis Soztutar's wrote in [1] : "In a HBase cluster, a region is only served by a single Region Server at any point in time, which is how HBase guarantees strong consistency within a single row. A table tipically consists of many regions, which are in turn hosted by many region servers. When a table is first created, HBase, by default, will allocate only one region for the table".

When a region is too large, HBase divides ("splits") it into two smaller regions. The decision of splitting a region is taken following one of these splitting policies:

- *CostantSizeRegionSplitPolicy*[2], used for HBase versions before 0.94, splits a region when the total data size for one of the stores (corresponding to a column-family) in the region gets bigger than the configured parameter *hbase.hregion.max.filesize*, which has a default value of 10 GB.

- *IncreasingToUpperBoundRegionSplitPolicy* [3] used for HBase version after 0.94, splits the regions using the number of regions hosted by its region server. When a region reaches an amount of data equal to `min( R^2 *` "*hbase.hregion.memstore.flush.size*", "*hbase.hregion.max.filesize*"), where R is the number of regions of the same table hosted on the same region server and the default memstore flush size is usually 128 MB.

- Additionally we also mention a splitting method called *KeyPrefixRegionSplitPolicy* [4]: after having configured the length of the prefix for the row keys, this splitting policy ensures that the regions are not split in the middle of a group of rows having the same prefix.

When an HBase cluster is created, all requests will go to a single region server, regardless of their number. For this reason, when it is started, this datastore cannot use the whole capacity of the cluster, but operates on a single node.

[1] Enis Soztutar, http://hortonworks.com/blog/apache-hbase-region-splitting-and-merging/, Hortonworks Inc.

[2] http://hbase.apache.org/apidocs/org/apache/hadoop/hbase/regionserver/ConstantSizeRegionSplitPolicy.html/

[3] http://hbase.apache.org/apidocs/org/apache/hadoop/hbase/regionserver/IncreasingToUpperBoundRegionSplitPolicy.html,

[4] http://hbase.apache.org/apidocs/org/apache/hadoop/hbase/regionserver/KeyPrefixRegionSplitPolicy.html/

However, the main reason HBase does not do default pre-splitting (splitting the regions before loading data) is because it can not know a-priori the data distribution.

The tables pre-splitting option is advisable only when the data distribution is known. If key distribution of all data is known, it would be possible to organize the regions of the cluster in order to distribute the workload in all region servers from the beginning of the work.

This phase is very delicate because a pre-splitting will ensure that the initial load can be more distributed throughout the cluster, but if the key distribution is not known, HBase will not truly distribute the load. In fact if the region split point is chosen poorly, there will be heterogeneous data distribution in the regions, limiting cluster performances.

HBase provides different client-side tools to manage this pre-splitting process.

Split points can be created choosing a predefined number of sub-regions and a corresponding key value, as shown by the code below:

```
hbase(main):001:0> create 'table', 'cf1', SPLITS => ['a', 'g', 'm']
```

This HBase shell [5] command shows how to create a table with a single column family ("`cf`") divided into four initial regions, providing three splitting points ("a", "g" and "m"). This command divides all key space in four ranges, assigning each of these ranges to each single region.

Regardless of pre-splitting type used, once a region reaches the default limit, HBase will automatically continue to divide it into two regions.

HBase also allows clients to manually force splits by means of a shell command. The following command can be used when a region gets too big and a region/table need to be splitted:

```
hbase(main):001:0> split 'table', 's'
```

"s" means the point where HBase applies the region splitting. In this case all data is distributed into two sub-regions: one that collects all data whose key starts with a letter in A-R range and the other does the same thing in S-Z range.

After this introduction on HBase splitting policies and pre-splitting methods, we now introduce how a row-key can be designed, in order to optimize data distribution and facilitate region pre-splitting.

[5] Apache Software Foundation, http://wiki.apache.org/hadoop/Hbase/Shell

4.1.2 Row-key design

HBase can query each table only by row-key. Key-ordering allows HBase to provide a scan interface, which allows to retrieve an ordered subset of data in optimized way. For example:

```
Scan s = new Scan(Bytes.toBytes("startrow"),Bytes.toBytes("stoprow"));
ResultScanner scanner = table.getScanner(s);
```

This operation returns all the rows whose key is between "*startrow*" and "*stoprow*".

It is straight forward that a good key-design is fundamental to limit the number of operations on the database and thus obtain better performance.

Starting from this analysis we have to say that there is not an absolute and perfect solution for all scenarios, because it depends a lot on how the database is designed and on the performed queries.

To better understand this point we propose this example:

EXAMPLE

To prove that is possible to reach better performance by properly choosing a good row-key design, we propose the following example.

We want to perform a Scan operation structured as described in figure 4.1 on a full-filled table with 1000 rows.

Column	Possible Values	Distribution	Filtered Value
cf.A	1,2,3,4,5,6,7,8,9,0	uniform	'4'
cf.B	all English alphabet letters (#26)	uniform	'C'
cf.C	bit value [0/1]	"0": 80% - "1": 20%	'1'

Table 4.1: Scan example on HBase table: for each column it is shown the column name (structured as ColumnFamily.ColumnName, all the possible values, their distribution and the value we want to filter).

HBase selections on a Scan result are provided by *Filters* as the following code shows:

```
Scan s = new Scan();
SingleColumnValueFilter filter = new SingleColumnValueFilter( cf, column, CompareOp.EQUAL, Bytes.toBytes("my value") );
scan.setFilter(filter);
ResultScanner scanner = table.getScanner(s);
```

The above operation consists of the complete scan of the given table and the application of a value filter on all obtained rows.

Referring to this example, we now propose different scenarios of how a good key-design can bring big improvements to query performances.

SCENARIO 1: the row key is a random value (for example an incremental integer value).

This is the code for this specific Scan operation:

```
Scan s = new Scan();
SingleColumnValueFilter filterA = new SingleColumnValueFilter( Bytes.toBytes("cf"),
Bytes.toBytes("A"), CompareOp.EQUAL, Bytes.toBytes("4") );
scan.setFilter(filterA);
SingleColumnValueFilter filterB = new SingleColumnValueFilter( Bytes.toBytes("cf"),
Bytes.toBytes("B"), CompareOp.EQUAL, Bytes.toBytes("c") );
scan.setFilter(filterB);
SingleColumnValueFilter filterC = new SingleColumnValueFilter( Bytes.toBytes("cf"),
Bytes.toBytes("C"), CompareOp.EQUAL, Bytes.toBytes("1") );
scan.setFilter(filterC);
ResultScanner scanner = table.getScanner(s);
```

Each filter is set in order to perform a selection over a single column, given a defined value. The three filters are applied in order, from the on performed on column `"cf.A"`. The whole cost in terms of considering unitary cost the effort to get a single row is:

cost = 1000 (filterA) + 1000/10 (filterB) + 1000/(10*26)(filterC) = 1103.85

SCENARIO 2: the row-key is designed following this way:
rowKey = [valueOfColumnA] + [random value number]. The "+" operand means "concatenation", made between the two strings. The code this time will be quite different:

```
Scan s = new Scan(Bytes.toBytes("4"),Bytes.toBytes("5"));
SingleColumnValueFilter filterB = new SingleColumnValueFilter( Bytes.toBytes("cf"),
Bytes.toBytes("B"), CompareOp.EQUAL, Bytes.toBytes("c") );
scan.setFilter(filterB);
SingleColumnValueFilter filterC = new SingleColumnValueFilter( Bytes.toBytes("cf"),
Bytes.toBytes("C"), CompareOp.EQUAL, Bytes.toBytes("1") );
scan.setFilter(filterC);
```

This time we will Scan the table only for lines that begin with the value of '4' (the ones that have '4' in cf.A value). The cost will be:

cost = 100 (filterB) + 100/26 (filterC) 103.85

SCENARIO 3: the row-key is designed following this way:
rowKey = [valueOfColumnA] + [valueOfColumnB] + [a random value number].

```
Scan s = new Scan(Bytes.toBytes("4c"),Bytes.toBytes("4d"));
SingleColumnValueFilter filterC = new SingleColumnValueFilter( Bytes.toBytes("cf"),
Bytes.toBytes("C"), CompareOp.EQUAL, Bytes.toBytes("1") );
scan.setFilter(filterC);
```

The Scan selects all lines with row-key that starts with '4c' value. That gets the same results as applying filters on columns A and B. The cost will be:

cost = 100/26 (filterC) 3.85

As we can see we have a very huge difference between performing the same query in different scenarios!

In scenarios 2 and 3 the key is designed in order to contain information about the value of its columns, and we saw that as the information, contained within the key grows, we are able to gain better performance.

We also have to point out that this row key design is performed in order to respond to a particular single query.

A problem may arise when different queries need to be performed on a single row-key which was designed for answering to a different query.

For example if we want to obtain the rows of the table showed in Scenario 2 with column C value equal to '0', we have to Scan again all rows in the table, returning only rows with value equal to '0'.

As we said before, there is not a single predefined choice: we can design a row-key to predispose the rows to a specific query, but we can not do it for all queries we are going to issue on that table.

The choice depends on two main different parameters:

- query execution frequency: if a query is executed many times, it could be better to design a row-key based on that query.
- distribution of data among the columns: if we want to perform a query like the one described in table 4.1, without the filter on column 'A', we can choose to design the key in different ways:

- valueOfColumnC + random number; the total cost will be a filter on column B on rows with key equal to '1xxxxx' (=$1000 * 20\% = 20$)
- valueOfColumnB + random number; the total cost will be a filter on column C on rows with key equal to 'Cxxxxx' (=$1000/26 = 38.46$)

Sometimes, designing one table with the first method, and a second table with the second method, can be a better solution rather then performing not optimized queries.

We tried to make a complete overview on the main problems that arise when designing a row key. Other observations and studies can be found at this link on Apache Website [6].

After showing HBase splitting policies and possible pre-splitting methods, and introducing possible key design studies, we are going to merge this knowledge, in order to present a possible row-key design and region pre-splitting solution for the two possible approaches that we have presented in section 3.2.

4.1.3 Splitting policy and key design for entities of the first approach

Considering the first approach described in section 3.2 and all observations made in 4.1.1 and 4.1.2, in this paragraph we show our thoughts on how to design a row key and organize a pre-splitting method for the first proposed approach (which was completely analyzed in Chapter 3).

This analysis is shown in order to better optimize data distribution, in order to be able to perform even more performant queries.

The table that will be stored in HBase is based on the entity (called entity "Q", cfr 3.4.2) we produced with the model we conceived in section (cfr 3.4).

Starting from a E-R schema, the number of generated Q-entities will be much as the number of queries known at design time.

Following the approach showed in the previous section about designing a row-key (4.1.2), the first and crucial information we want to include in the `KEY` attribute is a query identifier. Since each query produces a different denormalization of the data, according to each query schema, each row of the final entity Q has to be classified, first of all, on the basis of the identifier of that query:

rowKey = [identifier of the query] + [something else] .

Starting from this point of view, we can try to add more information about the data stored in each row.

However, our model optimizes the dataset, in order to be able to distinguish rows that answer our queries from the one that do not.

[6] Apache Software Foudation, http://hbase.apache.org/book/rowkey.design.html

This approach, indeed, maps a set of entities and relationships into an entity Q on the basis of a query schema and produces an entity that will have all data predisposed to answer to those specific queries. The unique information that we need when performing those queries on the produced entity is gained by attribute `valid` value. As we have already explained, this attribute helps to distinguish the rows that generate a positive answer to a given query, so we decided to include this information inside the key.

So the key we conceived will be composed as follows:

rowKey = [query identifier] + [value of bit `valid`] + [random string (can be a number or a string)] .

When a query is performed on such key, it will return only the rows that correspond to its identifier and that will have `valid` bit equal to '1'. This key-design will guarantee only one Scan operation per each performed query, without any further Get, Scan or Filter operations.

As regards HBase splitting policy for this approach, we do not have strong consistency requirements that force us to store different rows in the same physical node. A possible future work can be the study of query frequency execution, in order to speculate on how to assign each region to each query, distributing the workload on all the cluster nodes. This could be a possible method of pre-splitting a table, but we can not prove it will guarantee a performance improvement both from consistency and workload optimization points of view.

4.1.4 Splitting policy and key design for the second approach

Even though we did not show a resolutive method with a practical solution to the second approach (cfr 3.2), here we can show a possible splitting policy and the consequent key design that can be adopted to better optimize data distribution among the nodes of the cluster.

This study is guided by the fact that this approach was conceived to store consistent rows in the same node, in order to be able to reach higher availability.

With consistent row we mean a set of rows that compose a complete answer to a query.

For this second approach, unlike the first one, we have no information about how this approach maps the original data. This analysis is left as a future work.

Notwithstanding, we can make few considerations about the possible splitting policies that can be adopted by this second approach.

In the previous Chapter (section 3.2), we showed that, starting from an E-R schema and a query-schema, we can create somehow a new mapping method that will denormalize the data into a multiple row solution. Unlike the first approach, here we can have different rows that are somehow "in relation" and all of them have to be found to return a complete answer to a query.

We want related rows to stay in the same region and prevent HBase from splitting a region thus dividing related rows. This considerations led us to adopt the *KeyPrefixRegionSplitPolicy* that was explained in section 4.1.1: this splitting policy ensures that the regions are not split in the middle of a group of rows that have the same prefix. With a key designed in order to exploit this policy, and an HBase cluster prepared for this option, we can be sure that related rows will be stored in the same region.

We conclude here the analysis of HBase Region splitting, saying that this type of analysis is still nowadays in development.

4.2 HBase Configuration

In this section we give an overview of the steps needed to configure an HBase cluster. HBase depends on Hadoop and Zookeper, as shown in section 2.4.1.

The final configuration that we propose here will be also the one adopted for the model evaluation, done Chapter 5.

HBase has two run modes: "Standalone" and "Distributed". As HBase Apache guide[7] explains, the *Standalone* mode means a not distributed HBase deployment that runs on the local machine. This is not the configuration we want to adopt because, as we already said in Chapter 2, one of the main advantages of NoSQL databases is the possibility scale across several machines.

Hence, the *distributed* mode is the one we are going to explore. In particular our aim is to gain *fully-distributed* mode, that is, "where the daemons are spread across all nodes in the cluster"[7].

It is not easy to configure Hbase according to the distributed mode, as it is for other NoSQL databases like MongoDB or Cassandra, because, before installation, it requires a working Hadoop cluster and an active Zookeeper service.

As we have already seen at the end of Chapter 2, HBase runs on top of HDFS [3] (cfr 2.4.1.2), so the first thing to do is to configure a working Hadoop cluster.

Paragraphs 4.2.1 and 4.2.2 explain in detail which parameters and files have to be modified to do all the installation process.

Before starting to show them, it is necessary to point out few details:

- all commands and configurations presented will refer to a machine with Linux based OS.

- the following paragraphs do not show all the possible parameters that can be configured. The ones not mentioned there can be found in the HBase guide [11].

[7] Hbase, Apache Foundation, http://hbase.apache.org/book/

- it is very important that all machines that are going to be part of the cluster can communicate among each other through *SSH* protocol. The same private key must be shared in each machine of the cluster.

- in all machine the `hosts` file must contain *usernames* and *ip addresses* of all the machines in the cluster;

- each machine requires a Java JDK installation. JDK 6 and 7 have been already successfully tested, while we suppose JDK 8 will work, but it has not been well tested yet.

4.2.1 Hadoop configuration

Hadoop can be freely downloaded from Apache website [8].

There are two main different versions of this framework, 1 and 2.

Hadoop 1 is the oldest version and it has some architectural limits: it is limited up to 4000 nodes per cluster, it supports only one namespace for managing HDFS, has a recognized bottleneck in JobTracker (a process that farms out MapReduce tasks to specific nodes in the cluster [9]) and only MapReduce jobs can be performed.

In contrast with the newest version of this framework which can run potentially up to 10000 nodes per cluster, supports multiple namespaces for managing HDFS metadata and introduces a new technology called Yarn [10], that splits the original JobTracker's major functionalities (resource management and job scheduling).

A deeper explanation of the differences is provided by the official guide [17].

The Hadoop version must be the same through all the machines in the cluster. Before choosing it, we have to point out a problem that we met during the first installation: HBase, like Hadoop, has different versions: each version is not fully compatible with all Hadoop versions. Table 4.2 shows the compatibility between the two. As we can see, Hadoop 1 is compatible with HBase's 0.92, 0.94 and 0.96 versions, while Hadoop 2.x has been successfully tested with HBase 0.96 and 0.98. The following table is shown on HBase Configuration WebSite[11]:

[8] http://hadoop.apache.org/
[9] Apache Wiki, http://wiki.apache.org/hadoop/JobTracker
[10] Apache Foundation, http://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html
[11] Apache HBase, http://hbase.apache.org/book/configuration.html

	HBase-0.92.x	HBase-0.94.x	HBase-0.96.x	HBase-0.98.x (support for Hadoop 1.x is deprecated)	HBase-0.98.x (support for Hadoop 1.x is NOT deprecated)
Hadoop-0.20.205	S	X	X	X	X
Hadoop-0.22.x	S	X	X	X	X
Hadoop-1.0.0-1.0.2 (HBase requires Hadoop 1.0.3 at minimum)	X	X	X	X	X
Hadoop-1.0.3+	S	S	S	X	X
Hadoop-1.1.x	NT	S	S	X	X
Hadoop-0.23.x	X	S	NT	X	X
Hadoop-2.0.x-alpha	X	NT	X	X	X
Hadoop-2.1.0-beta	X	NT	S	X	X
Hadoop-2.2.0	X	NT - `pom.xml` has to be changed	S	S	NT
Hadoop-2.3.x	X	NT	S	S	NT
Hadoop-2.4.x	X	NT	S	S	S
Hadoop-2.5.x	X	NT	S	S	S

S = supported and tested
X = not supported
NT = it should run, but not tested enough

Table 4.2: Hadoop version support matrix

After downloading and unpacking the desired version in the correct directory, there are few settings files that have to be configured.

The fundamentals are `core-site.xml`, `hdfs-site.xml` and `hadoop-env.sh`.

In the first file we need this two properties:

```
<property>
<name>fs.default.name</name>
<value>hdfs://IP:PORT</value>
<description>namenode uri</description>
</property>

<property>
<name>hadoop.tmp.dir</name>
<value>HADOOP-DIR</value>
<description>temp directories</description>
</property>
```

This two properties are needed to configure the Namenode: the first indicates its IP address and it access port; the second refers to its meta-data directory. All the other possible properties can be found at [12].

As regards the second file (`hdfs-site.xml`), one important property needs to be configured:

```
<property>
<name>dfs.replication</name>
<value>VALUE</value>
<description> number of replication </description>
</property>
```

This property is used to configure the replication factor of the hadoop cluster.

`hadoop-env.sh` file needs only the configuration of `"JAVA_HOME"` variable, in order to make Hadoop able to locate Java installation path. This is necessary because Hadoop is written in Java.

There are also other two files, called `masters` and `slaves`, that respectively must be filled in with the ip addresses of Secondary Namenode and Datanodes ip addresses.

All these files must be copied in all the machines declared in the master and slaves files.

Before starting the whole cluster, the file system must be formatted.

In order to do that, we have to run this command on Namenode machine:

$HADOOP_INSTALL_DIR/bin/hadoop namenode -format

We can now start the whole cluster, running the following command on Namenode machine:

[12] Apache Hadoop, http://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-common/core-default.xml

$HADOOP_INSTALL_DIR/bin/start-dfs.sh

4.2.2 HBase configuration

After installing Hadoop, we can do quite the same thing with HBase.

Again, we must configure two files: `hbase-site.xml` and `hbase-env.sh`, similar to the one configured for Hadoop, the last one needs `"JAVA_HOME"` to be configured; `hbase-site.xml` needs following properties to be configured:

```
<property>
<name>hbase.master</name>
<value>IP-MASTER:PORT</value>
<description>master host port</description>
</property>

<property>
<name>hbase.rootdir</name>
<value>hdfs://IP-NAMENODE:PORT/hbase</value>
<description>directory shared by regionservers</description>
</property>

<property>
<name>hbase.cluster.distributed</name>
<value>true</value>
<description>fully distributed</description>
</property>

<property>
<name>hbase.zookeeper.property.clientPort</name>
<value>PORT</value>
<description>Port at which the client will connect</description>
</property>

<property>
<name>hbase.zookeeper.quorum</name>
<value>IP(S)</value>
<description>list of servers in zookeeper quorum</description>
</property>
```

hbase.master property contains the IP address and port number (default: 54310) of HBase Master machine.

hbase.rootdir value contains the Namenode IP address and port (the same provided in Hadoop configuration).

hbase.zookeeeper.property.clientPort contains the value of the port that Zookeeper uses to listen for client connections.

The last property contains the IP addresses of all members of Zookeeper cluster, because, as we have already said in section 2.4.1.3, Zookeeper is a distributed service.

In addition to these two files also `regionservers` file must be configured with the IP addresses of all machines used as region servers.

Again, as it happened for Hadoop configuration, we have to copy all these files in all HBase cluster nodes. After that we can start our HBase cluster, executing the *./bin/start-hbase.sh* command from the HBase folder of the machine elected as HBase Master. This will automatically run all processes, included the region server(s) ones.

The HBase installation process is quite long, especially if the cluster is composed of a huge number of nodes. During the first installation process, we decided to build up a Linux script able to auto configure and install an HBase cluster, that helped us avoiding the long and tedious work of copying all installation files in all cluster machine.

Further information on this script can be found in Appendix B.

4.3 How mapping is executed

In the remaining part of this Chapter we are going to introduce the last technical things we made before starting our model evaluation.

In particular we present how the realized tool technically performs the theoretical denormalization we explained in section 3.4.

The tool is designed to provide appropriate classes and methods useful to perform the rules of the model proposed in Chapter 3.

In particular, it provides the structures needed to perform these four phases:

- extracting a E-R schema from an XML File provided by the user;
- extracting some queries from an additional XML file;
- create a new single entity Q that represents the denormalization of entities given by the E-R schema, following the model showed in section 3.4;

We decided to use XML files as input because they can be easily generated and parsed. We added in fact a Java library, called JDOM [13], that gives us all functionalities to process XML files.

For better understanding how the tool works, few methods are reported in Appendix C.

E-R schema

[13] The JDOM Project, http://www.jdom.org/

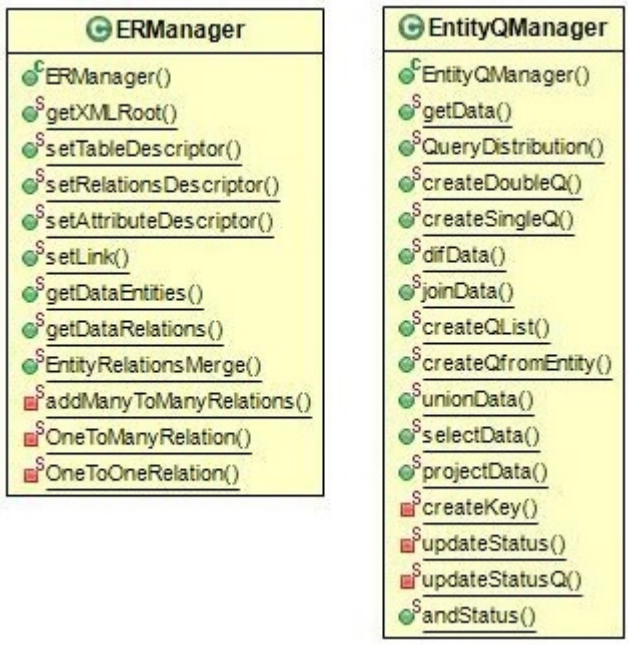


Figure 4.2: Tool main classes

The E-R schema is fundamental to logically understand the composition of the database. This file allows us to extract all entities and all relationships of the database, making us deduce its logical schema. The file must be provided by the user. It has to be XML compliant with the DTD structure shown in figure 4.1.

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE er [
<!ELEMENT er (entity*|relationship*)>
<!ELEMENT entity (attribute*)>
<!ATTLIST entity id CDATA #REQUIRED>
<!ATTLIST entity key CDATA #REQUIRED>
<!ATTLIST entity name CDATA >
<!ELEMENT attribute (#PCDATA)>
<!ATTLIST attribute name CDATA #REQUIRED>
<!ELEMENT relationship (link+|attribute*)>
<!ATTLIST relationship id CDATA #REQUIRED>
<!ATTLIST relationship name CDATA >
<!ELEMENT link (#PCDATA)>
<!ATTLIST link id  CDATA #REQUIRED>
<!ATTLIST link minCard CDATA #REQUIRED>
<!ATTLIST link maxCard CDATA >
]>
```

Figure 4.1: DTD - E-R schema

ERManager class is the class instantiated for this task. It is shown in figure 4.2.

There are three methods that need to be mentioned: *setEntityDescriptor*, *setRelationDescriptor* and *setAttributeDescriptor*; they are the direct methods responsible

for:

1. processing the XML file and get all useful information about entities, relationships and attributes;

2. creating a single entity for each of them: *EntityDescriptor*, *RelationshipDescriptor* and *AttributeDescriptor*. These three classes are designed in order to describe any kind of entity, relation or attribute. They are provided due to the fact that we do not deal with a predefined number of entities or relationships, but we have to be able to deal with any of the possible database structures. They are described by figure 4.3.

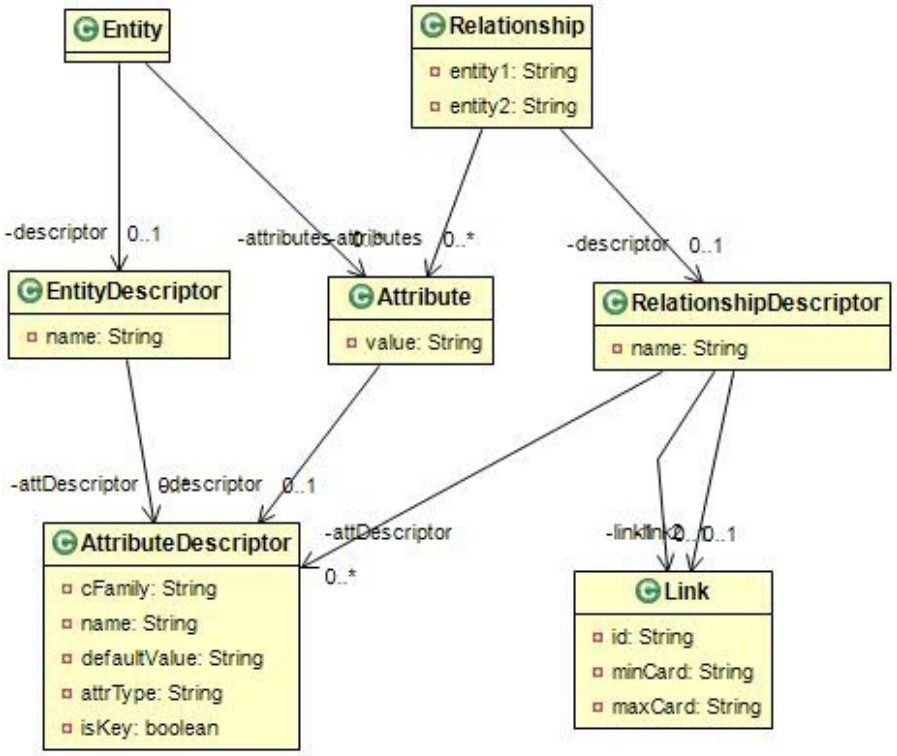


Figure 4.3: Entities and relationships class diagram

Extract a query

As it was conceived, our model needs a query schema as input, in order to denormalize data. The user must provide the XML file that represents the E-R query schema. It has to be compliant with the DTD shown in figure 4.4.

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE query [

<!ELEMENT query (union|join|projection|selection|difference)>
<!ELEMENT union ((query|entity)+)>
<!ELEMENT join ((query|entity)+)>
<!ELEMENT difference ((query|entity)+)>
<!ELEMENT projection ((entity|query)?,attribute?)>
<!ELEMENT selection ((entity|query)?,attribute?,qualifier?)>
<!ELEMENT attribute (#PCDATA)>
<!ELEMENT qualifier (#PCDATA)>

<!ATTLIST union id CDATA #REQUIRED>
<!ATTLIST join id CDATA #REQUIRED>
<!ATTLIST projection id CDATA #REQUIRED>
<!ATTLIST selection id CDATA #REQUIRED>
<!ATTLIST difference id CDATA #REQUIRED>
]>
```

Figure 4.4: DTD - Query schema

The method responsible for extracting a query is *getAllQueries*, inside *QueryManager* class (**??**). Since the query can be a composed one, this method processes the input XML schema specific for the query and extracts all the possible nested queries, in order to be able to perform them. The output is a list of queries that will be used for the next step.

Denormalization

Once the E-R schema and the query are obtained, the next step is to put into practice the model equation we conceived in 3.4.3.

In particular, the tool is structured as follows:

EntityQ class represents the output entity of the mapping process. It has all the original attributes of the original *Entity*, plus `key, valid` and `status` attributes (4.3).

EntityQManager is responsible for getting the queries, and create the corresponding entity Q. The method instantiated to do that is *createQFromEntity*.

The tool provide also additional three methods, useful for creation and update of `KEY`, `valid` and `status` attributes.

The following code shows how the `KEY` attribute is created:

```
private static String createKey(String query, String status)
int key = //variable is set to a random number
return String.valueOf(query+status+key);
```

As we can see, the key is composed by an initial number that identifies the query, the value of the `status` attribute and a random value, as we design at the beginning

of this Chapter (4.1.1).

The second method is *updateStatusQ* that is responsible for updating the `status` value. We described this process when we introduced entity Q (3.4.2). Since the Java method that implements this process is quite verbose, we attached it in Appendix C.

Last observation is on the update method concerning `valid` attribute. The following piece of code is explains how the update operation is performed:

```
public int updateValid(int old, int new){ new = new * old;
return new;
}
```

The tool provides the structure of a new entity, called *entityQ*, that represents the result of our mapping process.

4.4 Summary

In this Chapter we showed a brief analysis of how the HBase splitting policy can be managed in order to reach better query performance and guarantee given requirements.

In the central part (4.2) we showed how an HBase cluster can be configured and installed, setting the essential properties.

In the final section (4.3) we showed how all the phases of the denormalization process, starting from an entity-relation and query schema as input, are technically performed.

Both two last phases are supported by a Linux script and a Java tool, whose code is explained in Appendix B and C.

All these considerations will be resumed in the next Chapter where we will set up a working cluster to evaluate the model analyzed in Chapter 3.

Chapter 5

Model evaluation

In this chapter we will describe the tests that we decided to implement for comparing queries an original entity-relation based dataset and the corresponding Q-model based datasets.

Following the model explained in Chapter 3, and using the configuration and the tool presented in Chapter 4, we carried out query execution tests to compare the performance between the two models.

We will compare query tests on our model with query tests executed on the original E-R data model, simply mapped in a HBase cluster, expecting an improvement mainly on queries that involve a relation between at least two entities.

These tests will be used to determine which model is the best and in which situation, according to all preliminary assumptions we have already made in the previous Chapters.

5.1 Models

The tests we have executed consist in a comparison of two models:

1. **Relational Model (R):** given an E-R schema and its dataset, each table is mapped into a single different table of an HBase cluster with the same structure and attributes as if they were stored in a RDBMS database.
   All traditional rules used to translate a Logical Database Schema from a E-R schema are maintained ([1], section 7.3). In particular:

   (a) in a many-to-many relation: each entity will generate a table with the same name and the same attributes contained in the original entity, while each relationship will generate a table with its name and attributes, and the identifiers of the entities involved.

   (b) in a one-to-many relation: the translation is the same as a many-to-many relation, but no relation table is generated. The identifier attributes of a

table are merged into the related table.

(c) in a one-to-one relation: each entity generates a table with the same name and attributes, and the identifier of one of the two tables is added as attribute in the second table.

The tests performed on entities that follow this model will be called "*TestR*".

2. **EntityQ-based Model (Q):** given a E-R schema and its dataset, for each query schema given at design time, an entity of the type "Q" (cfr 3.4.2) is created with the rules explained in Chapter 3. This entity Q is mapped to a table inside an HBase cluster as shown in section 5.3.2.
   The KEY attribute is stored according to the solution we proposed in 4.1.3, that is appending the value of `valid` attribute to the query identifier (we assume equal to a string with value "00000") as prefix and adding a random number value at the end.
   The tests performed on entities that follow this model will be called "*TestQ*".

5.2 Dataset structure

To execute our tests we need an example dataset. In order to be significant, we would have preferred if this had at least three entities and two relationships (at least one Many-To-Many relation).

The dataset we chose was found in the repository of University of California, Irvine [2], is called "Restaurant & consumer data, recommender systems domain" and it is described in figure 5.1. This dataset was previously used to test different approaches, like "The collaborative filter technique on rating entity" and "the contextual approach generated the recommendations using remaining eight data files".

We decide to describe it with traditional E-R and logical schema to give a complete view, even though we are not going to use a traditional RDBMS data model. It is composed by four entities and four relations:

- USER (U): this entity describes the person's details. The dataset contains 138 rows and 19 attributes.
- GEO-PLACE (GP): this entity describes all restaurants details. The dataset contains 130 rows and 21 attributes.
- REGIONAL_CUISINE (RC): this entity describes all the types of cuisine. The dataset contains 103 rows and 2 attributes.
- PARKING (P): this entity describes all possible parking types. The dataset contains 7 rows and 2 attributes.

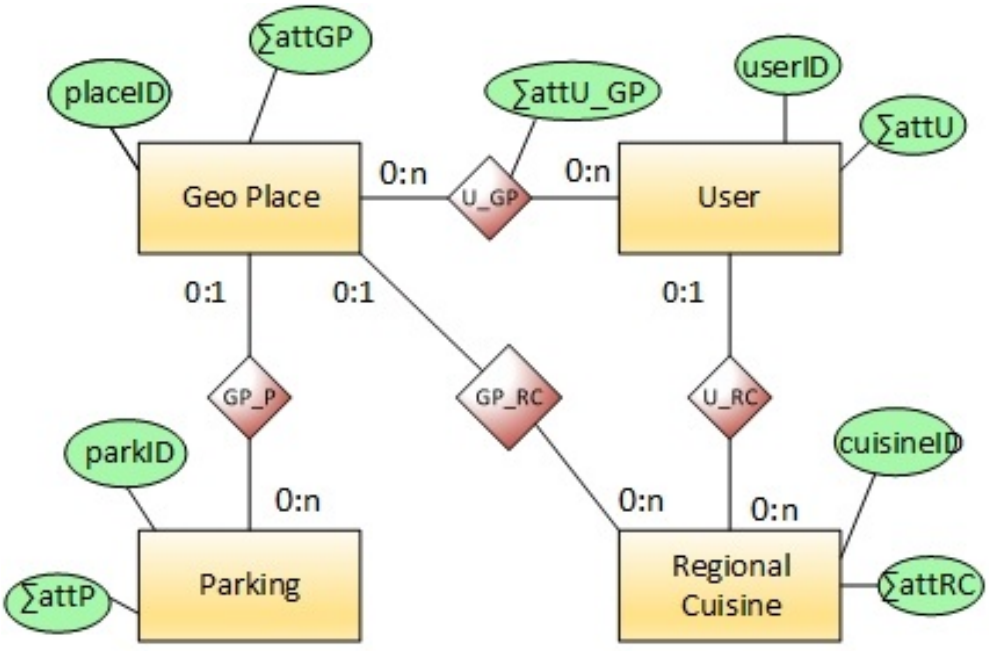


Figure 5.1: E-R schema

- USER_CUISINE (U_RC): it is a one-to-many relation between U and RC tables. It represents the user's preferred regional cuisine.
- GEO-PLACE_CUISINE (GP_RC): it is a one-to-many relation between GP and RC tables. It represents the type of cuisine offered by each restaurant.
- GEO-PLACE_PARKING (GP_P): it is a one-to-many relation between GP and P tables. It represents the type of parking offered by each restaurant.
- RATING (U_GP): it is a many-to-many relation between U and GP tables. It has 3 attributes and represents the users' evaluation about restaurants.

The E-R schema described above would generate this tables in a relational database:

1. **USER** (userID, latitude, longitude, the_geom_meter, smoker, drink_level, dress_preference, ambience, transport, marital_status, hijos, birth_year, interest, personality, religion, activity, color, weight, budget, height)
2. **GEO-PLACE** (placeID, latitude, longitude, the_geom_meter, name, address, city, state, country, fax, zip, alcohol, smoking_area, dress_code, accessibility, price, url, Rambiance, franchise, area, other_services)
3. **PARKING** (parkID, lot)
4. **REGIONAL_CUISINE** (cuisineID, name)
5. **U_GP** (userID, placeID, rating, food_rating, service_rating)

Attributes are not totally filled. HBase is a good way to store this kind of dataset.

In figure 5.2, attached at the end of the Chapter, we can see the XML ER schema, expressed in term of the DTD shown in Chapter 4.1.

The dataset can be freely downloaded in csv format on the University Repository Website [1] . In order to be able to use our Java tool, described in section 4.3, we need to convert it in XML format. We used Microsoft Access [15] to do it.

5.3 Testing method

To execute tests, we are aware of a tool, called YCSB, Yahoo Cloud Service Benchmark, [2] ([7]), that is very useful to make random tests on all the main NoSQL datastores: HBase, Cassandra, MongoDB, etc. .

This tool allows to perform the main database operations (insert, read and update) and to collect their performance. This tool is a good way to perform tests on our selected datastore, but is based on a random-values not-manipulable dataset and provides many options that are not fundamental for our test work, such as the number of client threads, the total number of random operations or different preset workloads.

It is not very feasible to the relational structure of the chosen dataset, by the fact that it does not expect relation structures like the ones we have, so we found difficulties in implementing its classes and methods.

We concluded to write our single class, on the basis of the test class of the YCSB tool, for our test purposes. It is appended in Appendix C.

5.3.1 Testing enviroment

To execute our tests, as we have already said in Chapter 2, we chose HBase as datastore, in particular we built up an HBase cluster with a single Master machine (Hadoop Namenode plus HBase Master) and two slave machines (Hadoop datanode plus HBase RegionServer). We do not focus on how many regions or in which RegionServer the regions are built up since we are not testing how HBase reacts to insert or scan queries, but we only want to test performances between the two methods we have mentioned.

The main parameter we want to monitor is latency; we call latency the time spent from the beginning of a single operation and its end. For operation we intend any function that involve an action on the database. We are going to show latency for the single insert or get operation, for all query operation (sum of all operations) and average latency on insert operations.

[1] University of California, Irvine, http://archive.ics.uci.edu/ml/datasets.html
[2] Yahoo Cloud Serving Benchmark, https://github.com/brianfrankcooper/YCSB/wiki

All tests are made from the same Java application and under the same constraints, so we have to take into account the time taken to reach the node of the cluster. This time has been monitored for all tests and we have calculated its variance value equal to 2,89%.

5.3.2 HBase mapping

In section 4.3 we showed the tool we have implemented. In order to perform the entities in the database, we built up a Java Application that exploit tool classes and methods to be able to store the data into an HBase working cluster. This section describes how we stored these entities into the HBase cluster.

As described in 2.4.4, HBase provides Java Client API to perform operations on a working cluster, so we decided to perform these tests from a Java Application.

The entire process can be divided in two sub-phases: first of all we need to set up a new HBase connection, in order to be able to communicate with the Master node; to the complete set up, some Java libraries are necessary. We do not explain here the details because they can be found in different HBase Java guides. We refer once again to Apache HBase Guide ([11] Chapter 10).

In a second phase we create all necessary tables useful for our tests.

To perform testR we need to create a single table for each of the tables of the original dataset (cfr 5.2), while for testQ we need only a single table, called "Q".

We implemented a method called "create" in *HBaseConnection* class that perform both functions: configuration and table creation. The specific piece of code is appended in section C.2.

After doing it, we can start executing our tests.

5.4 Tests description

In this section we are going to present and execute the test queries. Each query will be performed for testQ and testR.

For each test we are going to divide its development in six parts:

1. aim and presentation of the test; as presentation we will show its query formal equation, expressed in Relational Algebra (3.3.1) and its E-R schema, expressed in term of the DTD shown in 4.4;

2. the dataset structure for both tests: testR and testQ;

3. specific operation performed for each model; in specific, how the query is technically executed on the database.

4. the test hypothesis: what we expect from each test execution.

5. test results: each specific test result table exposes the number of rows involved, the average latency of the single operation and the total latency of the entire operation. We have to mention that we executed each test different times, depending on the test, and we report the average values of these executions. Single test variance has been calculated and it can be considered nonsignificant.

6. Comment: after the execution, a comment on each single test result is provided.

Referring to the E-R schema in figure 5.1, the test queries we are going to perform are the following:

- Test 1 - insert query; we will perform the entire insertion of the dataset into the database;
- Test 2 - selection query: we will perform a selection on a single attribute of `GeoPlace` table;
- Test 3 - Join query: we will perform a Join between `GeoPlace` and `Parking` tables;
- Test 4 - Complex and nested query: after a selection on table `User`, we will perform a Join between its result and tables `GeoPlace` and `Regional Cuisine`.

5.4.1 Test 1: insert

This first test analyzes the performance when inserting all the rows of our dataset into an HBase cluster, for both models, R and Q.

Aim

For model R, it is necessary to insert all data just once, while for model Q we have to insert all rows of a new and different table each time we have a query known at design time and we apply the denormalization method explained in Chapter 3.

This test is conceived to compare the two insert operations, trying to evaluate performance between the two models and between each insertion of Q entities in the database.

For this test, we do not provide a Relational Algebra query-schema, as we are only testing the insertion of entities into the database.

TestR dataset structure

We have already mentioned in 5.1 that testR dataset is stored in HBase with the same structure as if it was in a RDBMS, maintaining all the Relational rules of join tables.

In particular, we mapped each table in the dataset into a single HBase table. In particular, the original database tables give name to HBase tables and column

families, while attribute names are mapped in HBase columns. Column families are called like tables because HBase needs almost one column family per each created table.

The key-values of the rows for each table are the key-values described by the logical schema of the original dataset, listed in 5.2.

About the table that represents the many-to-many relation ("U_GP"), we decided to store key-values built attaching `placeID` value to the `userID` value. Figure 5.3 describes an example to better understand the concept:

KEY	U.userID	U.smoker	GP.placeID	GP.name
00564aty982	00564	...	aty982	...
84342abc342	84342	...	abc342	...

Figure 5.3: testR key-value design

This solution is not the unique solution, as we have already analyzed in paragraph 4.1.2 of the previous Chapter.

TestQ dataset structure

Since the fact that we do not have a static dataset for testQ but it depends on the query given as input to create it, in this test we will monitor many model dataset insertions.

The first is a simple model where all rows are inserted into the datastore, simply adding only `KEY`, `valid` and `status` attributes for each row of the original dataset (figure 5.4).

The `valid` bit is set to "1" and `status` is built with all attributes enabled (cfr 3.4.2).

The other results collected are given from the Q-models of the other tests we are going to show in the next paragraphs. As we will see they are datasets created denormalizing the original data, each one on the basis of a single different query.

Column Families	Q	GP	P	U	U_GP	RC
COLUMNS	-status -valid	-placeID -name -address -city -state -country -...	-parkID -lot	-userID -smoker -drink_level -dress_preference -transport -...	-rating -food_rating -service_rating	-cuisineID -name

Figure 5.4: HBase testQ table

Operation

The operation needed for this type of query is the same for both testR and testQ and it consists only in a sinlge `Put` operation, performed for each single HBase row we want to insert. Put operations for testQ instances are a little bit different from testR, because attributes valid and status need to be taken in consideration. The following is the `Put` operation for testR:

```
Put p = new Put(list.get(i).getIdAttribute());
for(int t= 0 ; t < list.get(i).getAttributes().size(); t++)
p.add(Bytes.toBytes("Q"),
Bytes.toBytes(list.get(i).getAttributes().get(t).getDescriptor().getClass()),
Bytes.toBytes(list.get(i).getAttributes().get(t).getValue())); }
table.put(p);
```

.

We present here testQ `Put` operation: it differs from the previous by the fact that valid and status attributes must be stored.

```
Put p = new Put(((EntityQ)list.get(i)).getKey());
p.add(Bytes.toBytes("Q"),
Bytes.toBytes("status"),Bytes.toBytes(((EntityQ)list.get(i)).getStatus()));
p.add(Bytes.toBytes("Q"), Bytes.toBytes("valid")
,Bytes.toBytes(((EntityQ)list.get(i)).getValid()));
for(int t= 0 ; t < list.get(i).getAttributes().size(); t++)
p.add(Bytes.toBytes("Q"),
Bytes.toBytes(list.get(i).getAttributes().get(t).getDescriptor().getClass()),
Bytes.toBytes(list.get(i).getAttributes().get(t).getValue())); }
table.put(p);
```

.

Test hypothesis

What we expect from this test is that the total latency of an insert operation in testQ is higher than in testR. This can be conceived for two reasons:

- Basically, testQ dataset rows have two more attributes than testR dataset rows.
- In testR we insert the rows as they are, while in testQ the number of attributes per single row can increase, as they can include all the attributes of all the entities involved in the query.

Test 1 results

Each total insert operation into the database has been executed five times.

We show here four tables where are reported, in order, the Operation Type, the number of Rows involved, the number of performed `Put` operation, the average latency for inserting each row and the total latency of the entire process. The first table refers to testR, while all the others refer to testQ insert operations.

R model				
Table	**Op. Type**	**#Rows**	**AVG Latency/row [ms]**	**Total Latency [s]**
P	Put	7	113,571	0,795
GP	Put	130	985,861	128,162
RC	Put	103	100,87	10,39
U_GP	Put	1161	251,319	291,781
U	Put	138	975,138	134,569
All	**Put**	**1539**	367,574	**565,697**

Table 5.1: TestR test 1

Q Model			
Op. Type	**#Rows**	**AVG Latency/row [ms]**	**Total Latency [s]**
Put	1539	484,006	744886,602

Table 5.2: TestQ insert test 1

Q Model			
Op. type	**#Rows**	**AVG Latency/row [ms]**	**Total Latency [s]**
Put	1539	477,124	734294,916

Table 5.3: TestQ insert test 2

Q Model			
Op. type	**#Rows**	**AVG Latency/row [ms]**	**Total Latency [s]**
Put	1531	476,319	729244,273

Table 5.4: TestQ insert test 3

Q Model			
Op. Type	**#Rows**	**AVG Latency/row [ms]**	**Total Latency [s]**
Put	1327	670,734	890064,066

Table 5.5: TestQ insert test 4

Comment

From the results exposed in the above tables we can note two main observations:

- As expected (cfr Test Hypotesis), insert test is directly dependent on latency of the single `Put` Operation;
- We can note from tables 5.1 and 5.2 that the difference between testR and testQ when we insert a simple dataset is given by `status` and `valid` attributes.
- As we can see looking at tables 5.3, 5.4 and 5.5, the total latency of the entire operation increases even though the number of rows decreases.
  This is due to the fact that rows stored are less in number but they have more columns and they collect more data.

5.4.2 Test 2: selection

The second test we are going to perform is based on a selection query that involves *GeoPlace* entity. In particular we want to select all cheaper Restaurants. In particular, we want all the restaurants that have a `"low"` value in `"cost"` attribute.

Aim

We designed this test in order to evaluate the power of modelQ when a Selection query is given at design time.

It can be represented by the following Relational Algebra equation:

$$\sigma_{price='low'}(GP)$$

and its query schema for our specific dataset is represented in figure 5.5.

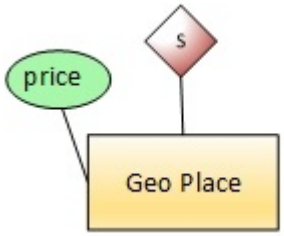


Figure 5.5: Query schema selection test

Dataset structure

The dataset used for this testR is the same as the one used for test 1. It is previosuly described in section 5.4.1, and its schema is shown in figure **??**.

The dataset used for testQ is quite the same as Test 1, but this time it is applied the Selection formula of the model presented in 3.4.3. Rows that answer to the query are stored with bit `valid` equal to "1", with a consequent key that has a starting prefix equal to "000001". All remaining rows are stored with `valid` bit equal to "0".

Operation

For this test case, the operation performed by testR and testQ to obtain the same result are quite different.

In particular, to test model R we need to search for table "GP" and apply a filter on it, in order to select all the values that match with the requested one:

```
Scan s = new Scan();
SingleColumnValueFilter filter = new SingleColumnValueFilter(Bytes.toBytes("GP"),
Bytes.toBytes("price"),CompareOp.EQUAL,Bytes.toBytes("low"));
s.setFilter(filter);
ResultScanner result = table.getScanner(s);
```

Table of testQ is instead already designed to answer to this test, since the key is predisposed too, as we showed in 4.1.3 and resumed in 5.1.

Then the operation is a simple Scan on a limited key-range values, in particular on which ones begin with a string with "000001" prefix:

```
Scan s = new Scan (Bytes.toBytes("000001"), Bytes.toBytes("000002"));
```

As we will see, this will be the same query operation for all the queries that we will perform to test model Q.

In this case the difference between the two queries is simply the difference between a filter on a single value of a specific column, and a Scan operation on a pre-defined row-key range.

Test hypothesis

We know, theoretically, that HBase Scan queries on a well designed key are faster than the same query performed on the entire table with a selection on the expected value ([11]).

In addition, testR operation has to apply the selection on all "GP" rows, while the testQ operation returns only the rows that are required.

Since the fact we do not deal with a huge amount of data, we can expect similar performance values.

Test 2 results

This test has been executed 10 times for each select operation. We do not present here the average latency for each row because it is represented by the Total Latency value, due to the fact that there is only one operation. Table 5.6 shows testR results, while table 5.7 shows the results for testQ.

R Model		
Op. Type	**#Operations**	**Total Latency [ms]**
Scan with filter	1	147,710

Table 5.6: TestR test 2

Q Model		
Op. Type	**#Operations**	**Total Latency [ms]**
Scan on limited range	1	50,8557

Table 5.7: TestQ test 2

Comment

As we said at the beginning of the description of this test, Scan operation for testR needs to locate the correct table, get all rows of that table and finally apply the filter on them.

TestQ is, instead, a simpler Scan operation on a single table, performed only on rows whose key has a starting String value equal to "000001".

We can see in tables 5.6 and 5.7 that the total latency measured by testQ is 3 times smaller than the one of testR. One of the possible reasons of this result can be the number of columns of the table used. As it is structured, "GP" table has 21 different columns. So, latency performance of testR can be highly affected by the fact that it has to locate the correct attribute before applying the filter.

5.4.3 Test 3: one to many relation

The test shown in this subsection involves a One-to-Many relation. We performed a Join operation, involving two different entities.

In this case we want to get all GeoPlace's parkings, showing all attributes for both entities.

Aim

With this test we want to prove the power of model Q on a Join operation that involves two different entities.

This test query can be expressed with the following expression:

$$GP \bowtie P$$

. Our aim is to extract both Parking and GeoPlace information, in order to get all information about the type of parking for each Restaurant. Figure 5.6 shows the query schema provided as input of the model.

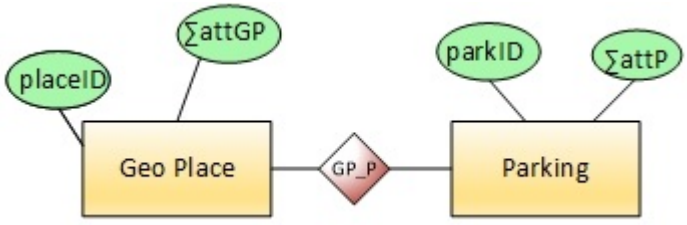


Figure 5.6: Query schema Join o-t-m test

Dataset structure

As we said for test 1 and test 2, the dataset for testR is always the same, described in 5.4.1 (Dataset structure describtion).

The structure of the dataset designed for testQ is the result of model's one-to-many formula described in 3.4.3.

In particular, we can note here that entity "P" has only one attribute over the identifier one. So, "Q" is designed in order to have in the same row the entities of "GP" and "P" that are related by the "GP_P" relation.

All rows with "P" and "GP" related in the same row have a `valid` attribute equal to "1". All remaining rows are stored with `valid` attribute equal to "0".

Operation

As we said different times, NoSQL datastores are not predisposed to perform Join operations between different tables.

To perform this particular operation for testR, we have to combine two searches, one on the first table, and one, with the results obtained, on the second table.

In testR, the operation consists in a Scan, followed by a Get operation:

```
Scan s = new Scan();
s.addColumn(Bytes.toBytes("GP"), Bytes.toBytes("P"));
ResultScanner result = table.getScanner(s);

...
//results stored in a list ...
foreach element in list
Get g = new Get(Bytes.toBytes(list.get(i).getKey()));
Result res = table.get(g);
endfor
```

Operation for testQ is always the same. It searches for all the rows that have key whose value starts with a string equal to "000001". In this case the key range is limited to the ending string "000002".

```
Scan s = new Scan (Bytes.toBytes("000001"), Bytes.toBytes("000002"));
```

Test hypothesis

TestR complete operation is composed of different single operations. TestQ is composed by a single Scan operation on a filtered key-space. This is the first reason that induces us to expect better latencies in testQ. We also expect that testR latency depends on the number of Get operations that are done after the first Scan, while testQ latency is dependent only on the final number of rows returned.

Test 3 results

As test 2, we executed 10 tests for each of the two models presented below. For better completeness, in testR model are added also the partial values of single operations, in this case Scan and Get(s) operations.

R Model		
Op. Type	**#Operations**	**Total Latency [ms]**
Scan	1	148,330
Get	123	5945,738
2 ops	124	12256,547 *

Table 5.8: TestR test 3
* it includes intermediate latency of getting Scan results and extracting the correct value to use for Gets operations.

Q Model		
Op. Type	**#Operations**	**Total Latency [ms]**
Scan	1	50,5467

Table 5.9: TestQ test3

Comment

Unlike the second test, in this case there is a big difference between model R and morel Q. This is given by the fact that testR needs to execute multiple operations. As we can see from tables 5.8 and 5.9, the final performances are very different. TestQ total latency is quite the same as the one measured in Test 2 (table 5.7). This is due to the fact that we performed the same operation.

TestR spends most of the time to extract the correct value from Scan results (see "*" in table 5.8) and use it to get the correspondent "P" data. The total latency of the intermediate operation affects performances as Scan operation returns all the columns of "GP" table, and it takes a lot of time to Get the correct column equal to "P".

This situation opens the possibility to future works, like studying a similar situation, focusing on the study of performances according to the number of each table columns.

In testQ, since the presence of a single operation, there is no time spent to extract information from the result of the previous operation. This fact highly influences total latency, making it 200 times faster.

As we can observe, on 130 rows of table "GP", 123 are related with table "P", so we are talking about a relation factor[3] of 94%. This factor highly affects testR performances, while it do not change very much testQ performances.

Our test results, as we have seen, strictly depend on the chosen dataset. Another possible future analysis can be the same comparison test, studied on different datasets.

5.4.4 Test 4: many to many relation

Last test involves the execution of a complex query. We want to get all only the Regional Cuisines of the Restaurants ("GP") where "informal" dressed Users have been. The query have to retrieve all attributes of all the involved entities.

Aim

It involves three different entities and two different relations:

$$RC \bowtie \Big(GP \bowtie \big((\sigma_{dresscode=informal}(U)) \big) \Big)$$

.

It represents a selection on an attribute of table User, and two Join operations between tables User, GeoPlace and RegionCuisine. Its E-R diagram can be seen in figure 5.7.

[3]The relation factor shows how many items of a table are related with the items of another table, given a relation between the two tables.

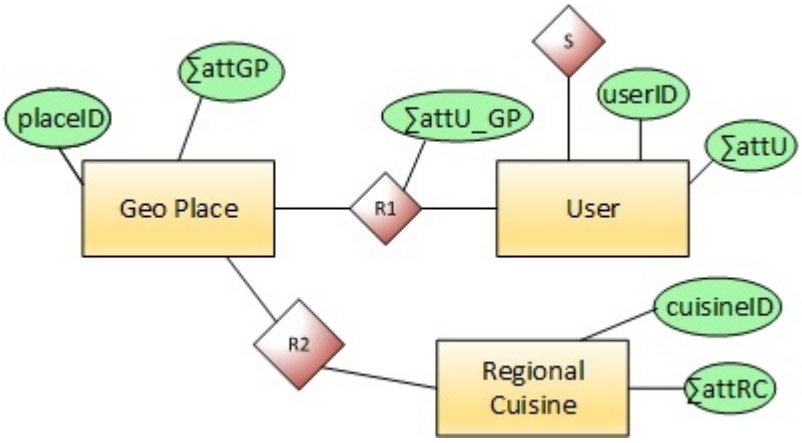


Figure 5.7: E-R schema test 4

Dataset structure

TestR dataset is described in 5.4.1.

As presented by figure 5.3, we designed the row-key of "GP_P" table, allowing to get GeoPlace information, given the identifier of table User. This key is well designed for the query we are going to perform for this test.

If we had designed a different row-key, like attaching the User identifier to the GeoPlace identifier, we would have performed the query starting from Regional_Cuisine table, leaving the Selection operation as last operation. In this case the Selection reduces the number of tuples that are involved in the relation, implying an theoretically faster execution.

Test Q dataset is denormalized in order to have attributes of entities `GP`, `P`, `GP_P` and `RC` in the same row. The rules used to build the model are described in 3.4.3 section.

Rows that answer to the final query are stored with bit `valid` equal to "1", with a consequent key that has a starting prefix equal to "000001". All the other rows, the ones that are from these entities but do not answer the query or the ones from the other entities, are stored with `valid` bit equal to "0".

Operation

In testR, as usual, the operations performed on the datastore are the one expressed by the following piece of code:

```
Scan s = new Scan();
SingleColumnValueFilter filter = new SingleColumnValueFilter(Bytes.toBytes("U"),
Bytes.toBytes("dress_preference"),CompareOp.EQUAL,Bytes.toBytes("informal"));
s.setFilter(filter);
ResultScanner result = table.getScanner(s);
...
//results extracted and put in a list
...
foreach element in list
Scan s1 = new Scan(list.getKey());
s1.addColumn(Bytes.toBytes("U_GP"), Bytes.toBytes("GP"));
ResultScanner result = table.getScanner(s1);
endfor
...
//results in lista1
...
foreach element in lista1
Get g = new Get(Bytes.toBytes(lista1.getKey()));
g.addColumn(Bytes.toBytes("GP"), Bytes.toBytes("RC"));
Result res = table.get(g);
endfor
...

//results in lista2
...
foreach element in lista2
Get g = new Get(Bytes.toBytes(lista2.getKey()));
get.addColumn(Bytes.toBytes("RC"), Bytes.toBytes("name"));
Result res = table.get("G");
endfor
```

As said when the dataset was presented (cfr 5.4.1) , we chose to start from the Selection operation (represented by a filtered Scan), and then perform the two Join operations.

The first one is done on the join-table "U_GP", using key designed ad hoc (cfr Dataset Structure section of this test) in order to get the GeoPlace identifiers. Once got them, we have to perform a Get operation to get the attribute value of the related Regional Cuisine of each GeoPlace . Given this one, we can finally retrieve the attributes of the linked Cuisines.

Each time an operation ends, its results are stored in a list, ready to be used for the sequent one.

For testQ, as it is conceived, the results of a given query, known at design time, are characterized by row with `KEY` value that starts with the string "000001", while all the others are stored with a key value that starts with the String value of "000000". The operation, as usual, is a single Scan operation:

```
Scan s = new Scan (Bytes.toBytes("000001"), Bytes.toBytes("000002"));
```

Test hypothesis

As described in the previous section, testR operation is a very complex operation that involves scanning different tables different times and getting filtered or single values.

TestQ is always the same single limited key-range Scan operation.

For this reason we expect a huge difference between testR and testQ, with clear low latency in the second case.

Test 4 results

The following tables represents the test results for test 4. Again, as we have done for test 3 (5.4.3), is shown the latency of all the intermediate operations. This test has been executed 10 times.

R Model		
Op. type	**#Operations**	**Total Latency [ms]**
Scan	1	50,811
Scan	35	20575,430
Get	325	15887,578
Get	239	11552,473
4 ops	600	50541,614 (*)

Table 5.10: TestR test4

* as done for test 3 (table 5.9), the total latency includes also the intermediate operations needed to extract the results and use them in the following operation.

Q Model		
Op. Type	**#Operations**	**Total Latency [ms]**
Scan	1	145,3147

Table 5.11: TestQ test4

Comment

Test 4, as Test 3, shows the difference between testR and testQ performances when the query involves a single or many relations.

For each operation on each different table, testR has to execute different operations, while testQ needs only to perform a single Scan operation to get all the results.

We deduce the relation factor of the many-to-many relation, given by the number of Get operations done in the third step of testR (cfr table 5.8), divided by the total rows of table "U_GP" (5.2) (325/1161 = 28%).

This value has a big influence on testR query performance, but not on testQ ones.

TestR directly depends on that, as higher is the number of the rows involved, higher will be the number of required operations.

Comparing testQ results from test 3 (table 5.9) and test 4 (table 5.11), we can observe that the total latency is not influenced by the relation factor of GP-U relation, but it is influenced by two other factors: the number of rows involved in the Scan operation and the number of columns per each row involved.

In third test the number of rows with key-value equal to "00001xxxx" was equal to 123, while, for the fourth test, the final rows are only twice, 239. However, total latency of test 4 is 2,9 times bigger than test 3. This is deductible by the fact that the rows stored in table Q of test4 involve a bigger number of columns, (45) than test 3 (20).

5.5 Summary

In this Chapter we presented the tests done and the results obtained. All results are quite in line with what we expected. We summarize the tests in table 5.12.

Test operation	Test	Best solution
Insertion	1	testR
Selection	2	testQ
Join on one-to-many relation	3	testQ
Join on many-to-many relation	4	testQ

Table 5.12: Test Results summary

In the table is shown which test obtained the smallest latency time, per each performed test. We have to point out that this results are obtained using a random chosen dataset and a specific configuration of the datastore, so we know that numbers can change if any of these parameter change.

We also notice that, despite a higher latency on insertion of data in testQ, query performance are really different, in particular when a Join operation is performed. We showed how testQ can reach performance 200 times faster thatn testR, when a query that involves a Join operation is performed. We can conclude that model Q can be a good solution for mapping relational structured data into a NoSQL datastore, in particular if the query involves at least one Join operation. For operations like selection, model R is anyway a good solution.

```
<?xml version="1.0" encoding="UTF-8"?>
<er name="er1" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" >
    <entity id="GP" name="GeoPlace" key="GP.placeID">
        <attribute name="GP.placeID"/><attribute name="GP.latitude"/>
        <attribute name="GP.longitude"/><attribute name="GP.the_geom_meter"/>
        <attribute name="GP.name"/><attribute name="GP.address"/>
        <attribute name="GP.city"/><attribute name="GP.state"/>
        <attribute name="GP.country"/><attribute name="GP.fax"/>
        <attribute name="GP.zip"/><attribute name="GP.alcohol"/>
        <attribute name="GP.smoking_area"/><attribute name="GP.dress_code"/>
        <attribute name="GP.accessibility"/><attribute name="GP.price"/>
        <attribute name="GP.url"/><attribute name="GP.Rambience"/>
        <attribute name="GP.franchise"/><attribute name="GP.area"/>
        <attribute name="GP.other_services"/>
    </entity>
    <entity id="P" name="Parking" key="P.parkID">
        <attribute name="P.parkID"/><attribute name="P.lot"/>
    </entity>
    <entity id="RC" name="RegionalCuisine" key="RC.cusineID">
        <attribute name="RC.cusineID"/><attribute name="RC.name"/>
    </entity>
    <entity id="U" name="User" key="U.userID">
        <attribute name="U.userID"/><attribute name="U.latitude"/>
        <attribute name="U.longitude"/><attribute name="U.the_geom_meter"/>
        <attribute name="U.smoker"/><attribute name="U.drink_level"/>
        <attribute name="U.dress_preference"/><attribute name="U.ambience"/>
        <attribute name="U.transport"/><attribute name="U.marital_status"/>
        <attribute name="U.hijos"/><attribute name="U.birth_year"/>
        <attribute name="U.interest"/><attribute name="U.personality"/>
        <attribute name="U.religion" /><attribute name="U.activity"/>
        <attribute name="U.color"/><attribute name="U.weight"/>
        <attribute name="U.budget"/><attribute name="U.height"/>
    </entity>
    <relationship id="GP_P" name="GeoPlace_Parking">
        <link minCard="0" maxCard="1" id="GP"/><link minCard="0" id="P"/>
    </relationship>
    <relationship id="GP_RC" name="GeoPlace_RCuisine">
        <link minCard="0" maxCard="1" id="GP"/><link minCard="1" id="RC" />
    </relationship>
    <relationship id="U_RC" name="User_RCuisine">
        <link minCard="0" maxCard="1" id="U"/><link minCard="1" id="RC"/>
    </relationship>
    <relationship id="U_GP" name="Payment">
        <link minCard="0"  id="U"/><link minCard="0"  id="GP"/>
        <attribute name="rating"></attribute><attribute name="food_rating"></attribute>
        <attribute name="service_rating" ></attribute>
    </relationship>
</er>
```

Figure 5.2: XML - entity-relational schema

Chapter 6

Conclusion and future work

This final Chapter reports the conclusions about the work exposed in the previous Chapters and it lays the groundwork for all subsequent future work.

6.1 Summary of work done

RDBMS have been the best way of storing data since '70s. However, nowadays requirements have changed, and features like high availability, scalability and fault tolerance are constantly required.

NoSQL databases are able to manage huge quantities of data and guarantee the above features with extremely good performances. Nevertheless, they differ in architectural structures and features with respect to RDBMS, that make it difficult to use them. One of the main problem of adopting a NoSQL database is the absence, in most of them, of structures that can help us to perform operations between different tables.

RDBMS represent a completely structured way of storing data. They provide ad hoc querying facilities to interrogate the system and a structured query language (SQL, Standard Query Language). NoSQL datastores can store unstructured data very well but they do not provide any standard query language. These are the main reason why RDBMS databases are still used nowadays to store data.

This work starts from this point of view, and tries to propose a data mapping model between a relational structure into a NoSQL one, in order to transfer custom data model that targets column base NoSQL databases.

We started our work assuming to have as input a dataset, designed primarily for read queries, as nowadays BigData datasets are. We also assumed that these queries are available at design time, during model development.

Starting from the entity-relation schema, and given a list of all the queries to be performed on such schema, we proposed two approaches to denormalize data from the original dataset, predisposing it on the basis of each query.

The first approach proposes to map all data requested by a single query into a single row of the target NoSQL datastore. This solution has been achieved after considerations about the fact that most NoSQL databases guarantee strong consistency on write operations that manipulate a single row. The second approach provides a mechanism to map all data into multiple rows; it has been designed in such a way that it guarantees that these rows are stored in the same physical space. This second approach has been identified in order to achieve high availability, in contrast with consistency requirements guaranteed by the first approach.

We proposed a model, called model "Q", for the fist approach and we studied its adaptability to all possible Relational Algebra queries. In particular, we showed how this model solves the lack of relations between tables in a column NoSQL database, i.e., HBase.

To test our model we needed to choose among many NoSQL databases. After a deep analysis and comparisons between the different features guaranteed by each of them, our choice fell on HBase.

The main reason which led us to this choice was that this database guarantees strong consistency on single row, and it also offers the possibility to design data row-keys and do node pre-splitting studies, to accomplish to requirements expressed by the second approach.

We analyzed this database and we proposed a way to design key solution for both approaches. We also showed how to configure a complete HBase cluster, understanding its correlation with Hadoop. Furthermore, we built a custom bash script able to set up an HBase cluster very quickly.

This partially allowed us to partially automate the test we performed.

In fact, the final part consisted in executing test of model Q, in order to evaluate its performance, in comparison to the classical relational model, called R, that maps data from a RDBMS compliant dataset to a NoSQL database.

To practically execute this evaluation work, a Java application has been developed. In particular, it takes as input the entity-relations schema, a single query schema and the whole dataset, and, following the model presented for the first approach, it denormalizes the given data into a single entity, ready to be stored in HBase.

After choosing an input dataset, we compared performance between model R and model Q, given different queries, in order to test the validity of our approach.

The first contribution that this thesis offers is the identification of two possible approaches for denormalizing data, from an E-R schema to a columner NoSQL compliant schema.

We provided a mapping model for the first approach, while we introduced the second approach in order to provide a valid alternative to the first one.

The last one can be found in the study of HBase datastore features, its configura-

tion and installation process and the analysis of its pre-splitting policies, proposing a possible key design solution, that allows to provide those availability guarantees that our second approach aims to at providing.

6.2 Future works

This work represents a starting point to analyze possible mapping methods from a relational schema to a columner NoSQL one.

The same mapping method can be also integrated with a study on insert and update queries.

A parallel work can be done studying the method for the second approach, trying to make the same analysis done in this thesis for the first one.

A deeper study of performance of the model proposed can be another possible future development. A deep test execution, maybe in relation to the relationships involved, or the study of the model performance when an expected query is performed, are other possible point of analysis.

A third possible future work consists in studying if the model presented can be mapped into a different NoSQL datastore, maintaining all properties and guaranteeing all needed requirements. Starting from this point, similar performance analysis can be done, comparing the one obtained here with the one obtained in a different database.

Bibliography

[1] Paolo Atzeni. *Database systems: concepts, languages & architectures*. Bookmantraa. com, 1999.

[2] K. Bache and M. Lichman. UCI machine learning repository, 2013.

[3] Dhruba Borthakur. The hadoop distributed file system: Architecture and design. *Hadoop Project Website*, 11:21, 2007.

[4] Eric A Brewer. Towards robust distributed systems. In *PODC*, page 7, 2000.

[5] Mike Burrows. The chubby lock service for loosely-coupled distributed systems. In *Proceedings of the 7th symposium on Operating systems design and implementation*, pages 335–350. USENIX Association, 2006.

[6] 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 Transactions on Computer Systems (TOCS)*, 26(2):4, 2008.

[7] 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*, pages 143–154. ACM, 2010.

[8] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[9] 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. In *ACM SIGOPS Operating Systems Review*, volume 41, pages 205–220. ACM, 2007.

[10] A Feinberg. Project voldemort: Reliable distributed storage. In *Proceedings of the 10th IEEE International Conference on Data Engineering*, 2011.

[11] Lars George. *HBase: the definitive guide*. " O'Reilly Media, Inc.", 2011.

[12] Brad Hedlund. Understanding hadoop clusters and the network. *Studies in Data Center Networking, Virtualization, Computing*, 2010.

[13] Patrick Hunt, Mahadev Konar, Flavio Paiva Junqueira, and Benjamin Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *USENIX Annual Technical Conference*, volume 8, page 9, 2010.

[14] James Manyika, Michael Chui, Brad Brown, Jacques Bughin, Richard Dobbs, Charles Roxburgh, and Angela H Byers. Big data: The next frontier for innovation, competition, and productivity. 2011.

[15] John L Viescas and Jeff Conrad. *Microsoft Office Access 2010 Inside Out*. Microsoft Press, 2007.

[16] Werner Vogels. Amazon dynamodb–a fast and scalable nosql database service designed for internet scale applications. *Retrieved July*, 30:2012, 2012.

[17] Tom White. *Hadoop: the definitive guide: the definitive guide*. " O'Reilly Media, Inc.", 2009.

[18] Jeremy Zawodny. Redis: Lightweight key/value store that goes the extra mile. *Linux Magazine*, 79, 2009.

Appendix A

Model Q examples

A.1 Introduction

This Appendix shows all the complete set of equations introduced in examples 1 and 2 presented in Chapter 3.

A.2 Example 1

This section shows the steps that allows us to generate the final table presented in example 1 (3.25).

The example consists of performing the following query on the tables described in RIF.

A

idA*	att1A	att2A
1	70	a
2	60	b
3	50	c
4	40	d
5	30	e
6	20	f
7	10	g

B

idB*	att1B	att2B	idA
1	one	600	2
2	two	500	4
3	three	400	6
4	four	300	
5	five	200	
6	six	100	

C

idC*	att1C	att2C	idB
1	alfa	e	1
2	beta	d	4
3	gamma	c	5
4	delta	b	
5	epsilon	a	

Figure A.1: A,B and C tables

We continue here the intermediate phases of the process described in section 3.4.4.1.

We start from the following equation:

$$B \bowtie_{R2} C = Q := (B \bowtie_{R2} C)_{valid=1}$$
$$\bigcup$$
$$\pi_{B.idB,\Sigma B.attB}(Q) := \Big[\pi_{idB,\Sigma attB}\big(B - \pi_{idB,\Sigma attB}(B \bowtie_{R2} C)\big)\Big]_{valid=0}$$
$$\bigcup$$
$$\pi_{C.idC,\Sigma C.attC}(Q) := \Big[\pi_{idC,\Sigma attC}\big(C - \pi_{idC,\Sigma attC}(B \bowtie_{R2} C)\big)\Big]_{valid=0}$$

Figure A.2 represents the intermediate tables that can be useful for better understanding the creation of entity Q.

(a) B JOIN C

idB*	att1B	att2B	idA	idC	att1C	att2C
1	one	600	2	1	alfa	e
4	four	300		2	beta	d
5	five	200		3	gamma	c

(b) $\pi_{idB,\Sigma attB}$(B JOIN C)

idB*	att1B	att2B	idA
1	one	600	2
4	four	300	
5	five	200	

(c) $\pi_{idC,\Sigma attC}$(B JOIN C)

idC*	att1C	att2C	idB
1	alfa	e	1
2	beta	d	4
3	gamma	c	5

(d) = B – (b) = B – $\pi_{idB,\Sigma attB}$(B JOIN C)

idB*	att1B	att2B	idA
2	two	500	4
3	three	400	6
6	six	100	

(e) = C – (c) = C – $\pi_{idC,\Sigma attC}$(B JOIN C)

idC*	att1C	att2C	idB
4	delta	b	
5	epsilon	a	

(f) Q := $(a)_1$ U $(d)_0$ U $(e)_0$:= (B JOIN C)$_1$ U (B – (B JOIN C))$_0$ U (C – (B JOIN C))$_0$

KEY*	idB	att1B	att2B	idA	idC	att1C	att2C	valid	status
	1	one	600	2	1	alfa	e	1	
	4	four	300		2	beta	d	1	
	5	five	200		3	gamma	c	1	
	2	two	500	4				0	
	3	three	400	6				0	
	6	six	100					0	
					4	delta	b	0	
					5	epsilon	a	0	

Figure A.2: B join C tables

After this first Join operation, between B and C tables, we can use the resulting entity Q as operator of the second Join operation with table A, producing table Q^*:

$$A \bowtie_{R1} Q = Q* := (A \bowtie_{R1} Q)_{valid=1}$$
$$\bigcup$$
$$\pi_{A.idA,\Sigma A.attA}(Q*) := \Big[\pi_{idA,\Sigma attA}\big(A - \pi_{idA,\Sigma attA}(A \bowtie_{R1} Q)\big)\Big]_{valid=0}$$
$$\bigcup$$
$$\pi_{Q.idQ,\Sigma Q.attQ}(Q*) := \Big[\pi_{idQ,\Sigma attQ}\big(Q - \pi_{idQ,\Sigma attQ}(A \bowtie_{R1} Q)\big)\Big]_{valid=0}$$

We can now replace Q in the equation with its original formula:

$$
\begin{aligned}
A \bowtie_{R1} (B \bowtie_{R2} C) = \\
&Q^* := \Big[A \bowtie_{R1} \Big((B \bowtie_{R2} C)_{valid=1} \\
&\cup \big(B - \pi_{idB,\Sigma attB}(B \bowtie_{R2} C)\big)_{valid=0} \\
&\cup \big(C - \pi_{idC,\Sigma attC}(B \bowtie_{R2} C)\big)_{valid=0} \Big) \Big]_{valid=1} \\
&\bigcup \\
&Q^* := \Big[A - \pi_{idA,\Sigma attA} \Big(A \bowtie_{R1} \Big((B \bowtie_{R2} C)_{valid=1} \\
&\cup \big(B - \pi_{idB,\Sigma attB}(B \bowtie_{R2} C)\big)_{valid=0} \\
&\cup \big(C - \pi_{idC,\Sigma attC}(B \bowtie_{R2} C)\big)_{valid=0} \Big) \Big) \Big]_{valid=0} \\
&\bigcup \\
&Q^* := \Big[\Big((B \bowtie_{R2} C)_{valid=1} \\
&\cup \big(B - \pi_{idB,\Sigma attB}(B \bowtie_{R2} C)\big)_{valid=0} \\
&\cup \big(B - \pi_{idC,\Sigma attC}(B \bowtie_{R2} C)\big)_{valid=0} \Big) \\
&- \pi_{idB,\Sigma attB,idC,\Sigma attC} \Big(A \bowtie_{R1} \Big((B \bowtie_{R1} C)_{valid=1} \\
&\cup \big(B - \pi_{idB,\Sigma attB}(B \bowtie_{R2} C)\big)_{valid=0} \\
&\cup \big(C - \pi_{idC,\Sigma attC}(B \bowtie_{R2} C)\big)_{valid=0} \Big) \Big) \Big]_{valid=0}
\end{aligned}
$$

we now exploit the distributive property of join over union:

$$
\begin{aligned}
A \bowtie_{R1} (B \bowtie_{R2} C) = \\
&(Q^*) := \Bigg[\Big(A \bowtie_{R1} (B \bowtie_{R2} C)_{valid=1} \Big) \\
&\cup \Big(A \bowtie_{R1} \big(B - \pi_{idB,\Sigma attB} (B \bowtie_{R2} C) \big)_{valid=0} \Big) \\
&\cup \Big(A \bowtie_{R1} \big(B - \pi_{idB,\Sigma attB} (B \bowtie_{R2} C) \big)_{valid=0} \Big) \\
&\bigcup \\
&Q^* := \Bigg[A - \pi_{idA,\Sigma attA} \Bigg(\Big(A \bowtie_{R1} (B \bowtie_{R2} C)_{valid=1} \Big) \\
&\cup \Big(A \bowtie_{R1} \big(B - \pi_{idB,\Sigma attB} (B \bowtie_{R2} C) \big)_{valid=0} \Big) \\
&\cup \Big(A \bowtie_{R1} \big(C - \pi_{idC,\Sigma attC} (B \bowtie_{R2} C) \big)_{valid=0} \Big) \Bigg) \Bigg]_{valid=0} \\
&\bigcup \\
&Q^* := \Bigg[\Bigg((B \bowtie_{R2} C)_{valid=1} \\
&\cup \big(B - \pi_{idB,\Sigma attB} (B \bowtie_{R2} C) \big)_{valid=0} \\
&\cup \big(C - \pi_{idC,\Sigma attC} (B \bowtie_{R2} C) \big)_{valid=0} \Bigg) \\
&- \pi_{idB,\Sigma attB,idC,\Sigma attC} \Bigg(\Big(A \bowtie_{R1} (B \bowtie_{R1} C)_{valid=1} \Big) \\
&\cup \Big(A \bowtie_{R1} \big(B - \pi_{idB,\Sigma attB} (B \bowtie_{R2} C) \big)_{valid=0} \Big) \\
&\cup \Big(A \bowtie_{R1} \big(C - \pi_{idC,\Sigma attC} (B \bowtie_{R2} C) \big)_{valid=0} \Big) \Bigg) \Bigg]_{valid=0}
\end{aligned}
$$

separating the equations:

$$
\begin{aligned}
A \bowtie_{R1} (B \bowtie_{R2} C) = \\
& Q^* := \Big[A \bowtie_{R1} (B \bowtie_{R2} C)_{valid=1} \Big]_{valid=1} \\
& \cup \Big[A \bowtie_{R1} \big(B - \pi_{idB,\Sigma attB}(B \bowtie_{R2} C)\big)_{valid=0} \Big]_{valid=1} \\
& \cup \Big[A \bowtie_{R1} \big(C - \pi_{idC,\Sigma attC}(B \bowtie_{R2} C)\big)_{valid=0} \Big]_{valid=1} \\
& \bigcup \\
& Q^* := \Big[A - \pi_{idA,\Sigma attA} \Big(\Big(A \bowtie_{R1} (B \bowtie_{R2} C)_{valid=1} \Big) \\
& \cup \Big(A \bowtie_{R1} \big(B - \pi_{idB,\Sigma attB}(B \bowtie_{R2} C)\big)_{valid=0} \Big) \\
& \cup \Big(A \bowtie_{R1} \big(C - \pi_{idC,\Sigma attC}(B \bowtie_{R2} C)\big)_{valid=0} \Big) \Big) \Big]_{valid=0} \\
& \bigcup \\
& Q^* := \Big[\Big((B \bowtie_{R2} C)_{valid=1} \\
& \cup \big(B - \pi_{idB,\Sigma attB}(B \bowtie_{R2} C)\big)_{valid=0} \\
& \cup \big(B - \pi_{idC,\Sigma attC}(B \bowtie_{R2} C)\big)_{valid=0} \Big) \\
& - \pi_{idB,\Sigma attB,idC,\Sigma attC} \Big(\Big(A \bowtie_{R1} (B \bowtie_{R1} C)_{valid=1} \Big) \\
& \cup \Big(A \bowtie_{R1} \big(B - \pi_{idB,\Sigma attB}(B \bowtie_{R2} C)\big)_{valid=0} \Big) \\
& \cup \Big(A \bowtie_{R1} \big(C - \pi_{idC,\Sigma attC}(B \bowtie_{R2} C)\big)_{valid=0} \Big) \Big) \Big]_{valid=0}
\end{aligned}
$$

Using the assignment operator property, explained in 3.3.2 section, we can reach this final equation, as reported in the Chapter:

$$A \bowtie_{R1} (B \bowtie_{R2} C) =$$

$$\pi_{idA,\Sigma attA}(Q^*) := \Big[A \bowtie_{R1} (B \bowtie_{R2} C) \Big]^{(a)}_{valid=1}$$
$$\cup \Big[A \bowtie_{R1} \big(B - \pi_{idB,\Sigma attB}(B \bowtie_{R2} C)\big) \Big]^{(b)}_{valid=0}$$
$$\cup \Big[A \bowtie_{R1} \big(C - \pi_{idC,\Sigma attC}(B \bowtie_{R2} C)\big) \Big]^{(c)}_{valid=0}$$
$$\bigcup$$
$$\pi_{idA,\Sigma attA}(Q^*) := \Big[A - \pi_{idA,\Sigma attA} \Big(\Big(A \bowtie_{R1} (B \bowtie_{R2} C) \Big)$$
$$\cup \Big(A \bowtie_{R1} \big(B - \pi_{idB,\Sigma attB}(B \bowtie_{R2} C)\big) \Big)$$
$$\cup \Big(A \bowtie_{R1} \big(C - \pi_{idC,\Sigma attC}(B \bowtie_{R2} C)\big) \Big) \Big) \Big]^{(d)}_{valid=0}$$
$$\bigcup$$
$$\pi_{idB,\Sigma attB,idC,\Sigma attC}(Q^*) := \Big[\pi_{idB,\Sigma attB,idC,\Sigma attC} \Big((B \bowtie_{R2} C)$$
$$\cup \big(B - \pi_{idB,\Sigma attB}(B \bowtie_{R2} C)\big)$$
$$\cup \big(B - \pi_{idC,\Sigma attC}(B \bowtie_{R2} C)\big) \Big)$$
$$- \pi_{idB,\Sigma attB,idC,\Sigma attC} \Big(\Big(A \bowtie_{R1} (B \bowtie_{R1} C) \Big)$$
$$\cup \Big(A \bowtie_{R1} \big(B - \pi_{idB,\Sigma attB}(B \bowtie_{R2} C)\big) \Big)$$
$$\cup \Big(A \bowtie_{R1} \big(C - \pi_{idC,\Sigma attC}(B \bowtie_{R2} C)\big) \Big) \Big) \Big]^{(e)}_{valid=0}$$

A.3 Example 2

In example 2 (cfr 3.4.4.2) we showed how the following query is performed on the tables described in figure RIF:

$$Q_A := \big(\sigma_{att1A \geq 40}(A)\big)_{valid=1} \cup \Big(A - \big(\sigma_{att1A \geq 40}(A)\big) \Big)_{valid=0}$$

The following figure presents the final table of the example 2,

KEY*	A.idA	A.att1A	A.att2A	B.idB	B.att1B	B.att2B	B.idA	valid	status
	1	70	a					1	
	2	60	b					1	
	3	50	c					1	
	4	40	d					1	
	5	30	e					0	
	6	20	f					0	
	7	10	g					0	
				1	uno	600	2	1	
				2	due	500	4	1	
				3	tre	400	6	1	
				4	quattro	300		1	
				5	cinque	200		1	
				6	sei	100		0	

Figure A.3: Example 2 final table

Appendix B

HBase auto-configuration script

B.1 Introduction

This Appedix shows how this script is configured and can be used. Instructions are given for Linux-based operating systems. Despite this, the script can be extended to any other operating system.

B.2 Prerequisites

It's very important, before running the script, that the cluster has this few characteristics:

- Each machine does not have a Java Virtual Machine version installed; Hadoop and Hbase does not work fine with all the JVM versions, so it is advised to have not installed a Java Machine in the cluster; the script is prepared for install a correct JVM version on each of the nodes involved;
- SSH protocol has to be correctly installed on each machine;
- Each machine of the cluster can access to any other machine with password less login;
- IPv6 net protocol has to be disabled, because it can cause some delays in script execution.

B.3 Script

We now describe how the HBase auto-installation script modifies all the necessary file settings of this datastore and runs a HBase cluster. To better understand the description below we refer to the source code, at the this link [1] .

[1] https://github.com/zianello/denormalizing_data

Any reference to "user" alludes to the people who want to use this script to install an HBase cluster. The tool is composed of the following folders:

1. `input folder:` this folder contains all files the user must modify; *download* file must be filled with the Hadoop and HBase versions he wants to install and configure. `hadoop_namenodes` , `hadoop_secondarynamenodes` , `hbase_masters` and `slaves` files has to be filled with the *usernames* and *ip-addresses* (username@ip) of each machine the user wants to configure as Hadoop namenode, Hadoop secondarynamenode, HBase Master and Slaves. Only the first machine in `hbase_masters` file will be the effective Master; all the the others will become active Masters if the first goes down. In `Replication` file must be defined the degree of Hadoop replication.

2. `hadoop_conf folder:`it includes all necessary files needed to configure Hadoop;

3. `hbase_conf folder:` it contains `hbase-env.sh` and `hbase-site.xml` files;

4. `main.sh:` this file is the file the user has to start to configure the whole cluster;

5. `configuration.sh:` this script file is used by *main.sh* file and it modifies all the *hadoop_ conf* folder and *hbase_ conf* folder's files with the input parameters included in *input* folder;

6. `conf folder:` this folder is empty and it will be filled with all Hadoop and HBase necessary files, ready to be copied in all the machines of the cluster. This folder contains a .sh file that will do all needed operations on each cluster machine.

B.3.1 How it works

There are four main execution steps:

1. *main.sh* . The user has to execute this file. First of all the script installs a Java Virtual Machine: if the user refuses to install it, the script will finish. If the script goes ahead, *configuration.sh* file will be invoked. This file is responsible of copy all configuration files present in *hadoop_ conf* and *hbase_ conf* folder in *conf* folder. Once copied, it sets all the properties list in 4.2.1 and 4.2.2 with the information given by the user in *input* folder (cfr B.3). Master and NameNode port are setted by default at 54310 and 9000. *conf* folder contains now all necessary set files.

2. For each machine, declared as Namenode, SecondaryNamenode, Datanode, HMaster or RegionServer by the user, the script will follow this few steps:

   - *conf* folder is copied into each machine;

- The script will verify on each machine if a JVM have already been installed. If no, it will install it;
- A local script is started on the machine: it downloads the correct version of Hadoop or HBase, extracts it and copies all necessary files included in *conf* folder into Hadoop and HBase downloaded folders.

3. The Namenode is formatted;
4. Hadoop and HBase instances are started.

B.4 Comments

This script is designed for a basic installation of an HBase cluster; for further settings and properties, we can refer to the Definitive Guide [17].

The following configurations has been successfully set up:
with "H" we mean Hadoop, as with "HB" we refer to HBase:

- Machine 1: H Namenode, H SecondaryNamenode, HB HMaster, Zookeeper
  Machine 2: H Datanode, HB Regionserver
  Machine 3: H Datanode, HB Regionserver
  Hadoop version: 1.2.1 / HBase version: 0.94.19 / Replication factor: 2

- Machine 1: H Namenode, H SecondaryNamenode, HB HMaster, Zookeeper
  Machine 2: H Datanode, HB Regionserver
  Machine 3: H Datanode, HB Regionserver
  Machine 4: H Datanode, HB Regionserver
  Hadoop version: 1.2.1 / HBase version: 0.94.19 / Replication factor: 1

- Machine 1: H Namenode
  Machine 2: HB HMaster, H SecondaryNamenode, Zookeeeper
  Machine 3: H Datanode, HB Regionserver
  Machine 4: H Datanode, HB Regionserver
  Hadoop version: 1.2.1 / HBase version: 0.94.19 / Replication factor: 2

- Machine 1: H Namenode
  Machine 2: HB HMaster, Zookeeeper
  Machine 3: H SecondaryNamenode, H Datanode, HB Regionserver
  Machine 4: H Datanode, HB Regionserver
  Hadoop version: 1.2.1 / HBase version: 0.94.19 / Replication factor: 2

- Machine 1: H Namenode, HB HMaster, Zookeeper
  Machine 2: H Datanode, HB Regionserver
  Machine 3: H SecondaryNamenode

Machine 4: H Datanode, HB Regionserver
Hadoop version: 1.2.1 / HBase version: 0.94.20 / Replication factor: 3

- Machine 1: H Namenode, H SecondaryNamenode, HB HMaster, Zookeeper
  Machine 2: H Datanode, HB Regionserver
  Machine 3: H Datanode, HB Regionserver
  Hadoop version: 2.2.0 / HBase version: 0.96.2 / Replication factor: 2

A cluster with HBase version 0.94.x and Hadoop version 2.0.1 was tested, with no positive result (cfr 4.2.1).

Appendix C

Tool and Tests Java methods

This section shows a list of Java methods. The first section shows methods built in our Java Tool, explained in Chapter 4.
The second section shows methods used in Chapter 5 to perform tests execution.
For each method is explained its main task and what produces.
For a complete overview, the entire set of classes and methods can be found at [1].

C.1 Tool Methods

This section shows main Java methods used in the tool presented in Chapter 4.

C.1.1 ERManager class

This class is the main responsible of extracting the correct information about the E-R structure of the dataset.
The following three methods show how the root element of the E-R query schema is taken as input.
Each of the following three methods extracts information about entities relationships and attributes from the XML file and creates a list of them.

```
public static List<EntityDescriptor> setEntityDescriptor(Element root){}
```

```
public static List<RelationshipDescriptor> setRelationsDescriptor(Element root){}
```

[1] https://github.com/zianello/denormalizing_data

```
public static AttributeDescriptor setAttributeDescriptor(Element rel, Element el){}
```

C.1.2 entityQ Manager

In this section are shown three methods of entityQManager:

public EntityQ createQfromEntity(EntityList en, EntityDescriptor descQ){}

The above method gets the original entit

private static String createKey(String status){}

The method *createKey* is responsible of creating the key value of Q entity, as it has been described in Chapter 4.

private static String updateStatusQ(String newStatus, EntityQ finalE){}

Last method is responsible of updating the Status attribute, given its old value. As shown in Chapter 3, the result is the AND operation between its value and the new value assigned.

C.2 Test methods

This section shows the methods implemented in order to create an HBase connection and to create the necessary tables. Last two classes are classes used to get tests results, as explained in Chapter 5.

public static void StartHbaseR(List<EntityDescriptor> objects, String name) throws IOException{}

This method is used to start a new HBase connection and to create, per each single EntityDescriptor object given as input, a single HBase table (with name "*name*"). This method is used to perform testR.

public static void StartHbaseR(List<EntityDescriptor>, String name){}

This one is the same as the previous, but it is used to create only one table, called Q. This method is used by testQ.

public static void insertRowsR(EntityQList list, String name) throws IOException{}

The method proposed above inserts all entities given as input in a single HBase table called whose name is given by the String *"name"*.

```
public static void measurements(int i ,int latency){}
```

This method is used to save the measurements of a single operation. It receives the latency of the operation (measured monitoring the SystemTime), and the number of operations it measures.

```
public static void resetUnit(){}
```

Last method is used to reset partial measurements of a query. It is used by testR in test 3 and 4 to give partial latencies of the different operations.