Referential Integrity in Cloud NoSQL Databases

by

Harsha Raja

A thesis
submitted to Victoria University of Wellington
in partial fulfilment of the
requirements for the degree of
Master of Engineering
in Software Engineering.

Victoria University of Wellington
2012

Abstract

Cloud computing delivers on-demand access to essential computing services providing benefits such as reduced maintenance, lower costs, global access, and others. One of its important and prominent services is Database as a Service (DaaS) which includes cloud Database Management Systems (DBMSs). Cloud DBMSs commonly adopt the key-value data model and are called Not only SQL (NoSQL) DBMSs. These provide cloud suitable features like scalability, flexibility and robustness, but in order to provide these, features such as referential integrity are often sacrificed. In such cases, referential integrity is left to be dealt with by the applications instead of being handled by the cloud DBMSs. Thus, applications are required to either deal with inconsistency in the data (e.g. dangling references) or to incorporate the necessary logic to ensure that referential integrity is maintained.

This thesis presents an Application Programming Interface (API) that serves as a middle layer between the applications and the cloud DBMS in order to maintain referential integrity. The API provides the necessary Create, Read, Update and Delete (CRUD) operations to be performed on the DBMS while ensuring that the referential integrity constraints are satisfied. These constraints are represented as metadata and four different approaches are provided to store it. Furthermore, the performance of these approaches is measured with different referential integrity constraints and evaluated upon a set of experiments in Apache Cassandra, a prominent cloud NoSQL DBMS. The results showed significant differences between the approaches in terms of performance. However, the final word on which one is better depends on the application demands as each approach presents different trade-offs.

Acknowledgements

First of all, I would like to thank my family for the support they provided me throughout my life and without whose support and encouragement, I would not have finished this thesis.

I am grateful to my supervisor, Pavle Mogin, for his understanding, continuous guidance and patience all through my thesis.

A very special thanks goes to Juan Rada-Vilela, for his invaluable help and support whenever I needed it. His time and patience is greatly appreciated.

Many thanks to Yi-Jing Chung and Jan Larres for all their help throughout the two years of my study.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

Cloud computing is a paradigm that is rapidly shifting the way IT services and tools are being used. The fundamental reason is that it provides essential computing services that can be accessed through the Internet. Amongst these services, cloud providers offer platforms that include hardware equipment and software applications, infrastructures such as servers and other network equipments, data storage and Database Management Systems (DBMSs), and others. All these services come with major benefits such as reduced maintenance costs, resource optimisation, and global access [56]. Moreover, such cloud services and other resources are hosted on the Internet and offered to users for availing according to their needs. Such characteristics made it natural for the cloud to become the backbone of application providers and users, and promoted the creation of novel theories and more applications to enhance the cloud.

Cloud data storage is an important service that has been progressing, evolving and adapting alongside cloud computing. As the number of applications and users keep increasing, so does the demand for scalable, efficient, and consistent data storage mechanisms offered by application providers. These storage mechanisms allow users to store their data on the cloud without having to deal with the hassle of buying and maintaining database servers [48].

The distributed nature of the cloud data storage, requires cloud DBMSs to satisfy essential requirements that are inherent to these environments. Some of these requirements are related to a) scalability by matching the increase and decrease of active users and allowing the storage of large amounts of data, b) robustness by maintaining several copies of data to make data available despite failures, c) flexibility by allowing the storage of unstructured data, and d) consistency of data by making sure data is never stale. These requirements are not necessarily met by traditional Relational Database Management Systems (RDBMSs). For example, scalability in RDBMSs is limited as these are designed to run on a single node. Robustness is lacking in terms of a single point of failure would make data unavailable. Also, rigid schemas for databases reduce the flexibility of storing unstructured data on the cloud. Thus, the prevalent features of traditional RDBMSs become issues when moved to the cloud. Nonetheless, there is still one characteristic that is desirable from RDBMSs: referential integrity in data.

Referential integrity constraints ensure that relationships between data items are preserved. For example, it ensures that a course exists before students can enrol in it. Such relationships are inherent to real-world data and have to be maintained in databases whether it is a traditional RDBMS or a cloud DBMS. Moreover, these constraints ensure that no operations violate the integrity between data items [25]. Despite its importance, referential integrity is currently not a prominent feature in most cloud column-oriented key-value DBMSs [47, 51].

Column-oriented key-value DBMSs, also referred to as cloud Not only SQL (NoSQL) databases, adopts the column-oriented key-value data model which is a widely used model for cloud DBMSs.These DBMSs support many features required in cloud, such as elastic scalability to match varying user demands, data replication, schema-less data storage , and many other features. In these DBMSs, referential integrity constraints are not provided because these were not conceived to maintain data relationships, partly due to the denormalized and decentralized nature of data storage [25]. Instead, the responsibility of maintaining referential integrity is delegated to the application layer. However, such an approach implies a significant overhead for applications as they have incorporate the referential integrity validation themselves. Not to mention that these DBMSs commonly handle large amounts of interconnected, dependant and widely replicated data which makes the validation more difficult. Hence, it becomes a critical problem for applications to handle such data where dependencies have to be correctly maintained and preserved.

Inspired by such problems, this thesis studies the existing modelling of data dependencies in cloud NoSQL DBMSs and contributes by providing four solutions to ensure that referential integrity is effectively maintained. These solutions extend the consistency of data relationships and ensure referential integrity even when it is widely replicated or even spread across data-centers. Also, they are provided as an Application Programming Interface (API) between the application layer and the database layer. This approach reduces the workload of applications, as the responsibility of validating referential integrity is delegated to these solutions.

1.1 Objectives

This thesis focuses particularly on the column-oriented key-value DBMSs on the cloud. Four solutions using different metadata management techniques are suggested to incorporate referential integrity constraints in such DBMSs. These solutions are implemented and tested on Apache Cassandra, a popular and prominent column-oriented key-value DBMS in the cloud.

The performance of these solutions is assessed in terms of response time and throughput. All the proposed solutions extend data integrity and preserves the data dependencies regardless of the scalability and workload of the DBMS.

1.1.1 General Objective

Incorporate validation mechanisms for maintaining referential integrity in column-oriented key-value DBMSs.

1.1.2 Specific Objectives

- Design four solutions to implement referential integrity constraints, using metadata in various ways to store the information about data dependencies.
- Develop an API to implement the four solutions in one of the existing column-oriented key-value DBMSs, namely Cassandra.
- Analyse the performance of the solutions to determine their feasibility and practicality.

1.2 Organization

The remainder of this thesis is structured as follows. Chapter 2 presents cloud computing and describes the column-oriented key-value data model, its challenges and the general architecture of Cassandra on which the proposed solutions are implemented and tested. Chapter 3 describes the design of the proposed solutions along with the motivation for the design. Chapter 4 describes the implementation of these solutions in Apache Cassandra. Chapter 5 details the experimental design used to measure the performance of the solutions. Chapter 6 presents the analysis of the results from the experiments. Finally, Chapter 7 presents the conclusions and further ideas to extend this research.

Chapter 2

Background

This chapter presents an overview of some of the significant topics relevant to this thesis. Section 2.1 describes cloud computing and the various services it provides. Section 2.2 presents cloud databases as one of the key services provided in the cloud. Section 2.3 describes the prevailing data models prominently used by cloud databases. Section 2.4 gives a detailed description of the column-oriented key-value data model, which is one of the popular and widely used data models in cloud databases. Section 2.5 presents some of the challenges existing in this key-value model. Section 2.6 addresses one of the crucial challenges of referential integrity in column-oriented key-value cloud databases. Section2.7 introduces the architectural concepts of Cassandra, the column-oriented key-value cloud Database Management System (DBMS) used in this thesis.

2.1 Cloud Computing

Cloud computing is a major paradigm that is rapidly shifting the way Information Technology (IT) services and tools are being used in the industry. It is perceived that cloud computing would help extend the capabilities of many IT and online services without the need for costly infrastructure.

Similar to remote computing where other machines or computers are accessed from the local machine through a network, cloud computing leverages network connections to provide various services to the users. It also brings with it the virtualization of applications and services, where it appears to users as if the applications are running on the user′s machine rather than a remote cloud machine [14]. This removes the need for installing the actual software by the users. Thus, both expert and naive users need not worry about the technical details and configurations to use these cloud services.

Cloud computing is generally based on a subscription model where users pay as per their usage, which is very similar to utility services like electricity, gas or water etc. The coalescence of virtualization, where applications are separated from the infrastructure is what makes cloud computing easy to use. Users do not have to invest in software applications as they can access such applications on the cloud. Users pay only for the services they use. For example, they pay only for the amount of storage their cloud database uses or pay only for the bandwidth consumed by the servers they rent from the cloud providers. Applications and databases are stored in large server farms or data centres owned by companies like Google, IBM etc.

The architecture of cloud computing services has users who avail cloud services as the front-end. A user is any hardware or software application that relies on cloud computing to perform its work. Notice that ′users′ represent the end-users, like database administrators or programmers or anyone who benefits from cloud computing services while ′client′ refers to any software applications or Application Programming Interfaces (APIs) that are used to perform cloud computing. The back end of the cloud architecture includes the cloud servers, databases, and computers etc. , which are abstracted from users. All the components like the servers, applications, the data storages work together through a web service to provide the users with the cloud services.

The overall structure of cloud computing and its various services have been generalised into layers [11, 49, 50].

Software as a Service (SaaS) is the service provided by the cloud providers where users do not have to install the software applications.

Platform as a Service (PaaS) is the service where a hardware or software platform is provided to users. A platform could be an operating system, programming environment, hardware, run-time libraries etc.

Infrastructure as a Service (IaaS) is the service where users can use the expensive hardware like network equipments, servers etc.

Database as a Service (DaaS) is a cloud storage service that represents the storage facilities, like DBMSs which are provided as cloud services for which users pay only for the storage space they use [39, 57].

DaaS involves hosting cloud databases in the cloud which offer data management, data retrieval, and other database services. Due to the increasing number of users deploying and using web applications on cloud, cloud databases form a crucial part to store the increasing amounts of data on the cloud. Many companies like Amazon, Google, IBM, and Microsoft provide DaaS and offer varying levels of services [39]. The next section gives a description about cloud databases and its key features.

2.2 Cloud Databases

Most cloud applications store, process and provide large amounts of data like the user information, application data or some stored data which maybe accessed by the users. Storage of such data during all times is essential for the cloud applications to operate correctly [37]. Traditionally, users store data in files or databases residing on dedicated database servers or on local disks, but the requirements for data storage on the cloud are very different and need a distributed approach in data storage, where data is spread across several machines.

Generally, DBMSs on the cloud (also known as cloud databases) are replicated so that multiple copies of data are available to cater to many users who access the same data at the same time [15]. This also helps in cases of server crashes or network failures, as copies of the data are available. Since data is replicated on several machines, these databases are distributed.

Being distributed, these DBMSs are spread across several machines that commonly belong to data centers owned by hosting companies like Google, Amazon etc. These data centres house many servers, computers and telecommunication infrastructure, including back up and security facilities and users can rent or buy the storage space they need. Within such data centres, data is stored on remote machines, which can be any server within the same or a different data centre. Thus, when users connect to cloud databases through the Internet, they remain unaware of the exact location of their stored data and are guided to their databases by APIs of the cloud DBMS [57].

Cloud databases have to be scalable across these servers so that data is available to any user at any given point in time. Scalability in the context of cloud storage refers to the ability of dynamically incorporating changes to the number of users or storage space, without affecting the functioning of the databases or the availability of data to the users. In other words, when more machines are added to increase storage capacity, or when more users access the same data, cloud databases should cope with the increased workload and yet maintain the same throughput.

In order to scale in such a distributed environment, most cloud DBMSs split data into distinct individual parts and save these parts on different nodes in the data centre across several databases. Hogan [36] claims that such data partitioning in cloud databases increases complexity as a database is spread across several servers and querying the database involves complex Joins and more time. As a result of the data partitioning, nodes have a subset of data or rows from each table in the database which moves the databases and the user applications farther apart, thus increasing latency [23].

In general, cloud DBMSs require more features than the traditional DBMSs for an efficient data management on the cloud and are found to be less efficient than traditional DBMSs because of the dynamic scalability required to support a changing user-base and [2, 26, 30, 36, 47, 51, 52]. Most traditional Relational Database Management Systems (RDBMSs) give data a structure, that adheres to a schema. This is mainly achieved through the process of normalisation where each table in a database is evaluated according to its functional dependencies and primary keys, to reduce redundancy and minimise integrity anomalies [25]. Normalisation causes databases to have smaller and structured tables by removing duplicate data from large and badly organised tables and by imposing constraints on the data. Tables are normalised to at-least First Normal Form (1-NF) ensuring data is organised and less redundant. Redundancy is further reduced by bringing the database schema into at-least Third Normal Form (3-NF) (or Boyce Codd Normal Form (BCNF)). Throughout the chapters normalization refers to making databases at least in 1-NF.

Unlike traditional DBMSs, cloud DBMSs are simple in their structure with minimum querying support and have a simple API for database administration. These have been made scalable to support the diverse and large number of users who store structured data and to support various applications that users use. Most such cloud DBMSs are non relational and follow a different data model, which is explained in Section 2.3. Such Cloud DBMSs are loosely termed as cloud Not only SQL (NoSQL) DBMSs and do not provide support for efficient querying or query languages such as SQL. Unlike RDBMSs, cloud NoSQL DBMSs do not aim to be ACID-compliant [26, 31, 51, 52]. ACID stands for the properties Atomicity, Consistency, Isolation and Durability, which ensure the completeness and reliability of a database operation. In general, unless operations are not ACID compliant in RDBMSs, it is not considered valid. But ACID compatibility in cloud NoSQL DBMSs is a bottleneck as it does not suit the distributed nature of the cloud environment [51, 55].

Cloud NoSQL DBMSs require to have high throughput, high availability and also require to be elastically scalable to increasing resources or users. This requires cloud NoSQL DBMSs to part with some traditional RDBMS functionalities (like JOINS) and ACID operations mainly because of its distributed nature [55]. Due to this distributed nature across different environments, cloud NoSQLs DBMSs are prone to node failures. Node failures and the elasticity prevailing in cloud environments affect consistency of data, which adversely affect the 'C' of ACID properties, i. e., Consistency. Moreover, network and data partitioning play a major role in affecting consistency and availability of data. A partition takes place when a node fails or there is a network failure at some point in the network. Such partitions pose problems in cloud NoSQL DBMSs as cloud databases rely on more than one server. Hence, these DBMSs aim for partition tolerance, which is the ability to continue their operations despite node failures and partitions. To achieve these features it is commonly found that cloud NoSQL DBMSs sacrifice data consistency. Commonly, most web applications aim to have their data available at anytime, as many users could access the data at the same time and in a business model, applications lose valuable customers if they are not kept satisfied with the services in terms of speed, availability and consistency. Hence, cloud NoSQL DBMSs aim to achieve properties that are different from the traditional ACID properties. These DBMSs aim to achieve Consistency, Availability and Partition-tolerance (CAP) properties as stated in the CAP theorem [29, 46, 55]. These properties and the CAP theorem are explained next.

2.2.1 CAP Theorem

In distributed environments or web-based applications, the three main system requirements necessary for designing and deploying applications are: Consistency, Availability and Partition tolerance [8, 29, 46, 55]. The

CAP theorem, proposed by Brewer [8], claims that it is not possible for a distributed system to achieve all these three properties at the same given time.

- Consistency: When a request is made to access data, a system is called consistent if it provides the correct and latest version of the data [4, 10, 34, 46, 55]. For example, in the case of an online shopping website, consistency ensures that the stock of items is always correct. When a user attempts to buy an item and the same item is being bought by another user, the system will have to ensure that both the users get the most recent stock details available. So if there is only a single item remaining, then the second user is informed that his request can not be completed as there is no stock available. This means that the data is consistent and users do not get stale data.
- Availability: A system is considered highly available when all parts of the system are always available, despite any failures or problems. It is expected that all requests will be addressed any given point of time [4, 10, 46, 55]. In the previous example, this means that even when the website is busy, with many users accessing it, it is expected to have all user requests addressed and to be up and running always.
- Partition-tolerance: Generally, distributed services are run on several machines across different networks and these services are prone to network partitions [8, 29, 46]. Network partitions happen when there is a failure of a segment or component of a network such that nodes cannot communicate with each other. A system is considered partition-tolerant when despite such partitions, it continues to provide its services and address user requests.

The CAP theorem states that at a given point of time, only two of these properties can be achieved or satisfied by any application. This means that distributed applications, such as cloud NoSQL DBMSs, have to make trade-offs on one of the properties always. Such trade-offs are always considered from the design stages in most distributed applications. Similarly, most cloud NoSQL DBMSs have chosen its priorities and the trade-offs that suits it the best. For example, Cassandra focuses on 'A' and 'P' while Bigtable focuses on 'C' and 'A' [13].

What such trade-offs mean in relation to the CAP theorem is examined below [1, 4, 9, 34, 46]:

Case 1: Achieving 'C' and 'A' properties: This means that when an application aims to achieve consistency and availability, it will be less partition tolerant. When data is partitioned, there is more time involved in accessing the data from the various points in the distributed network. Moreover, failures mean more time delays. Thus, in order to achieve high consistency and availability the application depends on fewer nodes. However, when applications that are not partition tolerant face a partition, it becomes unreliable. It could either give inconsistent data or become unavailable or both.

Case 2: Achieving 'A' and 'P' properties: Commonly, this is what most cloud NoSQL DBMSs aim to achieve and in this case less attention is paid to consistency of data [55]. A system lacking consistency is thus mostly available even during partitions, but may give stale data or incorrect data occasionally to the users. This suits most business models as it ensures that data is always accessible to users. For example, in the online shopping example, when data is not available or the request fails, users can get anxious whether they lost their money during the transaction. In order to avoid this, users are presented with data as soon as possible, despite being stale, since users will be able to see the data rather than being left unsure.

Case 3: Achieving 'C' and 'P' properties: This means that while a system is consistent and tolerant to partitions or failures, it may not always be available and running. Such a system provides correct data while tolerating network failures but may not be accessible during failures preventing operations to be performed on data. This leads to a less reliable system, where data is correct but unavailable and inaccessible during network failures.

Interestingly, while cloud NoSQL DBMSs do not comply with ACID properties, the CAP theorem has lead to a new set of properties called BASE and is considered as an alternative of ACID properties in distributed and scalable systems [44]. BASE refers to the properties Basically Available, Soft-state and Eventually consistent. This means that data is basically available, although, at some point not all data will be available. Soft-state indicates that data could be lost if not properly maintained, i.e., data has to be refreshed and version-checked for it to remain saved. Eventually consistent, as mentioned previously, is a weak form of consistency where in a cluster of nodes, nodes do not get the updates immediately. BASE could be understood as being closer to `Case 2` mentioned above, where consistency takes a back seat. But this leads to conflicts where a new update or a new read request is made before all nodes get the latest update. In order to resolve such conflicts there are some types of repairs used by cloud NoSQL DBMSs, such as read and write repairs [53]. When a read or write operation takes place, such repairs check for inconsistency in data before correctly updating the data. Some cloud NoSQL DBMSs also rely on APIs to work around such issues.

All these characteristics make cloud NoSQL DBMSs very different from traditional DBMSs that are used outside cloud networks. As mentioned previously, the underlying data model of the cloud NoSQL DBMSs is fundamentally different from the relational data model of RDBMSs and this is explored in the following sections.

2.3 Cloud Data Models

Data models describe the structure of a database and give the users information on how a database can be used or implemented. On the cloud, different types of data models exist. The selection of a data model for a cloud database depends on the problem the cloud database is specialised to address or a feature it is incorporating. Some of the current popular data models on the cloud are:

- Key Value data model
- Document data model
- Relational data model

Both key-value and document data models store data in key-value pairs, where in databases using the document data model,documents are stored with a key and this key is used to retrieve the document [45]. In databases using the key-value data model, data is stored with a key which is used to retrieve the data. On the other hand, relation data model on the cloud adopts the traditional relational data model and store data in rows in tables [40].

In general, cloud DBMSs are non-relational and most cloud DBMSs adopt the key-value data model to maintain the data replication, consistency and scalability that are part of cloud data storage [2, 52]). The key-value databases, document databases and other databases that support non-relational data models on the cloud are loosely termed as NoSQL databases. NoSQL DBMSs are considered the next generation cloud DBMSs that aim to provide non-relational distributed DBMSs with open-source content and development for the cloud [52]. Many NoSQL DBMSs, that are inherently key-value DBMSs, have evolved by adopting various features from other popular cloud NoSQL DBMSs. For example, Cassandra adopts the column oriented data model of Google's Bigtable [13] while Riak ( ) is influenced by Amazon's Dynamo [22]. This thesis focuses on the column-oriented key-value data model and is explained in Section 2.4.

Although RDBMSs on the cloud are not widely used, there exist some cloud capable RDBMSs like Amazon Relational Data Service, Microsoft

SQL Azure etc. Just like the traditional relational model, relational model on the cloud also supports relations or tables with rows and columns to store structured data and adheres to a schema [12, 40]. These RDBMSs provide users with database administration facilities and APIs to perform operations on stored data, like updating, inserting, deleting data etc. However, the replication of data is restrained due to the relational nature of such RDBMSs and affects its scalability and performance.

2.4 Key-Value Data Model

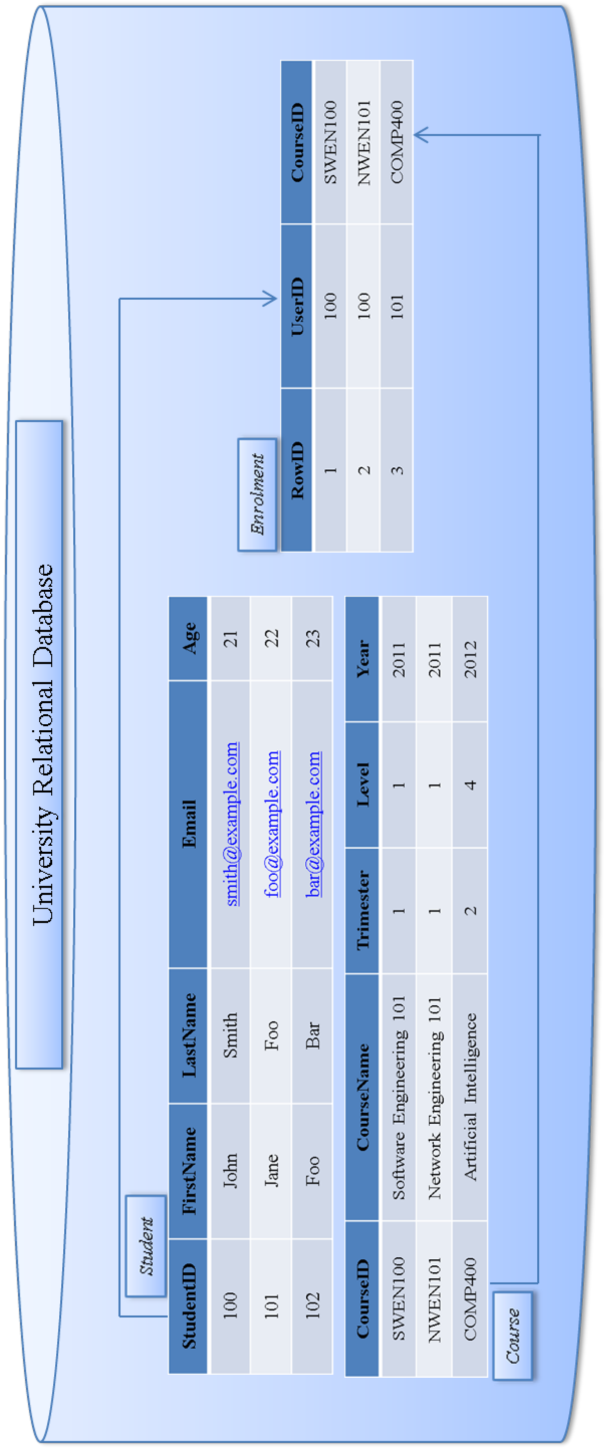
In basic terms, the key-value data model represents data as a key-value tuple consisting of a key, a value and a timestamp. A key is a unique string commonly encoded as UTF-8. A value is the actual data that has to be saved and it is associated with a key that is used to retrieve the value from a key-value database. The value is commonly of the string data type. This is similar to the way data is stored in a map. A timestamp is a 64-bit integer that records the time at which the value was inserted or updated in any way.

Generally, the key-value data model on cloud implements the column-oriented approach, which is adopted from Google's Bigtable [13]. The data model explored in this section is the column-oriented key-value data model adopted by Cassandra. This type of data model is fundamentally different from the relational data model. It sacrifices ACID properties as well as normalisation in order to achieve high scalability, fault tolerance, data partitioning among others. To understand this new type of data model and cloud DBMSs that adopt this model, comparisons are drawn to RDBMSs that adopt the relational model. For this purpose a simple example of a University database is used throughout the chapters, where it is assumed that students enrol into different courses. This example is illustrated below.

When the University database is saved in an RDBMS, a schema will be applied. This example assumes that the details of the students are saved in a table called `Student` and the course details in the `Course` table. The Student-Course relationship is maintained in a separate table called `Enrolment` which has foreign keys for both `Student` and `Course` tables. This can be seen in Figure 2.1.

This shows how the University database example is deployed as a Relational Database (RDB). When data in the University example is modelled using the column-oriented key-value data model, the way it is stored is different. Although key-value DBMSs are schema-less, column-oriented key-value DBMSs are not entirely schema-less and hold some information about the databases as metadata , as seen in Cassandra [18]. Such DBMSs allow applications to model the way data is organised in a traditional RDBMS, whilst bringing more flexibility by denormalising data and imposing no rigid structures or schema requirements [18, 33, 38]. Therefore, it allows applications to add data in the way they want and change their schema (if needed), without adhering to a rigid schema unlike the traditional RDBMSs.

The building blocks of column-oriented key-value DBMSs are the columns, the Super Columns, the Column Family and the Key Space. Using the University example, these terminologies are explained below. Appropriate analogies are drawn with the RDB University, as seen in Figure 2.1, to better understand these column-oriented key-value concepts. Since the focus is on Cassandra's data model, these concepts are explained in the way Cassandra deploys them. The example used to describe the Cassandra



Student

StudentID	FirstName	LastName	Email	Age
100	John	Smith	smith@example.com	21
101	Jane	Foo	foo@example.com	22
102	Foo	Bar	bar@example.com	23

Course

CourseID	CourseName	Trimester	Level	Year
SWEN100	Software Engineering 101	1	1	2011
NWEN101	Network Engineering 101	1	1	2011
COMP400	Artificial Intelligence	2	4	2012

Enrolment

RowID	UserID	CourseID
1	100	SWEN100
2	100	NWEN101
3	101	COMP400

Figure 2.1: University example as a Relational database

data model adopts a simple and flexible schema that allows some structure in the way data is stored.

As previously mentioned, cloud NoSQL DBMSs are generally specialised to address specific problems like partition-tolerance, high availability among others and for this some trade-off are made when these are developed. Some of the challenges and problems present in such DBMSs are discussed in the following section.

2.4.1 Columns

A column is the basic unit of data in this data model. It is a tuple containing a column name, a value and a timestamp (Figure 2.2).

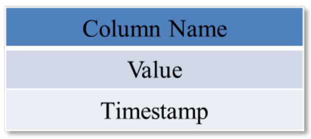
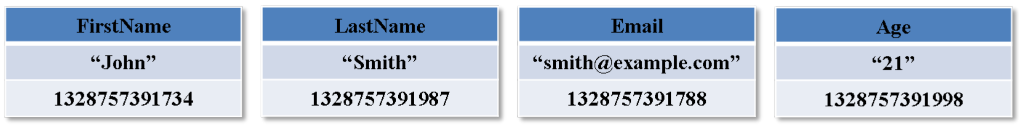


Figure 2.2: A column in Cassandra

The column names are labels and it is mandatory that a column has a name. Column names and values are stored as Bytes Type, Long Type, Ascii Type, binary values Lexical UUID Type, Time UUID Type or as UTF8 serialized strings [18, 33]. Timstamps are used to store the time of the latest update made to the column and are thus used for conflict resolutions. The timestamp values are commonly stored as microseconds, but could be in any format that the application chooses. However, timestamp formats have to be consistent across the database so that is the same format across all columns.

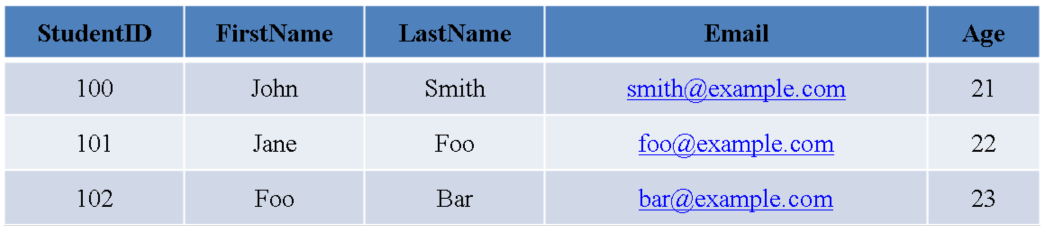
Cassandra allows indexes to be created on column names. These are called Secondary indexes and are of type `Keys` in Cassandra. When such secondary indexes are used, efficient queries can be specified using equality predicates, and can be made on ranges of columns too. The latter ones are called range queries.

A column name can be considered analogous to an attribute name in a table in any traditional RDBMS. To illustrate this analogy, Figures 2.3 and 2.4 show the differences between the representation of values in `Student` in Cassandra and in an RDBMS. It can be seen from these figures that a column in the column-oriented key-value data model is similar to a single value in a row of a relational table. For example, the data `'John'` in the relational table `Student` can be considered equivalent to a single column in Cassandra.



FirstName	LastName	Email	Age
"John"	"Smith"	"smith@example.com"	"21"
1328757391734	1328757391987	1328757391788	1328757391998

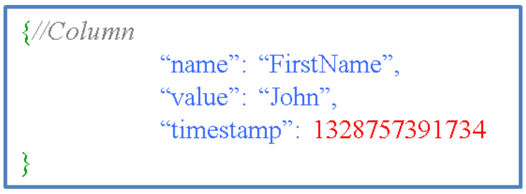
Figure 2.3: Columns in Cassandra



StudentID	FirstName	LastName	Email	Age
100	John	Smith	smith@example.com	21
101	Jane	Foo	foo@example.com	22
102	Foo	Bar	bar@example.com	23

Figure 2.4: Relational Table - Student

The JSON notation for columns in Cassandra is shown in Figure 2.5.



```
{//Column
        "name": "FirstName",
        "value": "John",
        "timestamp": 1328757391734
}
```

Figure 2.5: JSON notation for a column

2.4.2 Super Columns

A super column is a different kind of a column where the values are an array of regular columns (Figure 2.6). It consists of a super column name and an ordered map of columns. The columns within the values of a super column are grouped together using a common look-up value, which is commonly referred to as the `RowKey`. In other words, a super column is a nested key-value pair of columns. The outer key-value pair forms the super column while the inner nested key-value pairs are the columns. Unlike regular columns, super columns do not have timestamps for its key-value pairs.

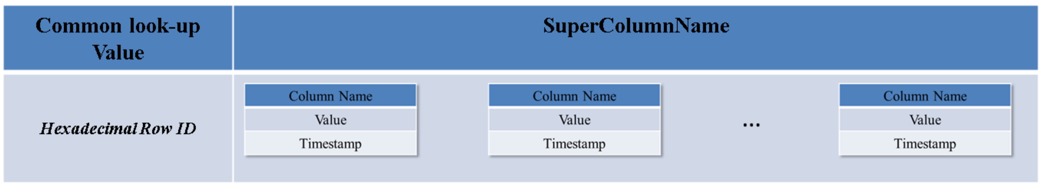


Figure 2.6: A Super Column

A super column can be considered roughly similar to a whole record in a relational table in an RDB. For example, the super column for a student, as seen in Figure 2.7, is analogous to a single record in the relational table `Student` (Figure 2.4).

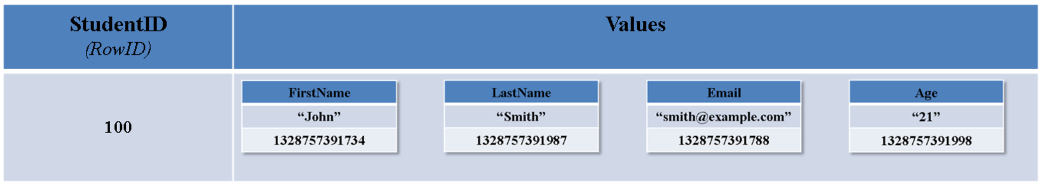


Figure 2.7: A Super Column for Student '`John`' in Cassandra

The JSON notation for a super column is shown in Figure 2.8.

```
{//SuperColumn
            "name": 100,
            "value": {//Columns
                        "FirstName": {//Column
                                              "name": "FirstName",
                                              "value": "John",
                                              "timestamp": 1328757391734
                                    }

                        "LastName": {//Column
                                              "name": "LastName",
                                              "value": "Smith",
                                              "timestamp": 1328757391987
                                    }
                        "Email": {//Column
                                              "name": "Email",
                                              "value": "smith@example.com",
                                              "timestamp": 1328757391788
                                    }

                        "Age": {//Column
                                              "name": "Age",
                                              "value": 21,
                                              "timestamp": 1328757391998
                                    }

                      }
}
```

Figure 2.8: JSON notation for a super column

2.4.3 Column Family

A column family contains columns or super columns that are grouped together using a unique row key. It is a set of key-value pairs, where the key is the row key and the value is a map of column names (Figure 2.9). The row key groups the columns together, just as in super columns.

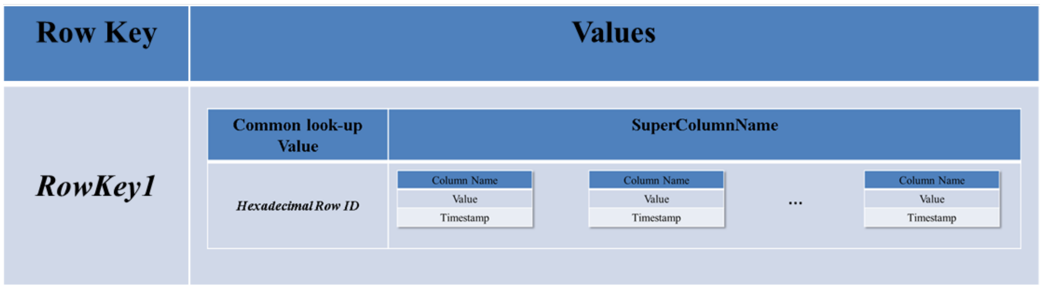


Figure 2.9: Column Family in Cassandra

Applications can define column families and metadata about the columns. It is commonly practised to have columns that are related or accessed together to be grouped in the same column family. Column families require that some attributes are always defined, like name, column type and others. It also has optional attributes that can be defined if the application requires so. Some of the optional attributes are number of keys cached, comments, read repairs, column metadata among others.

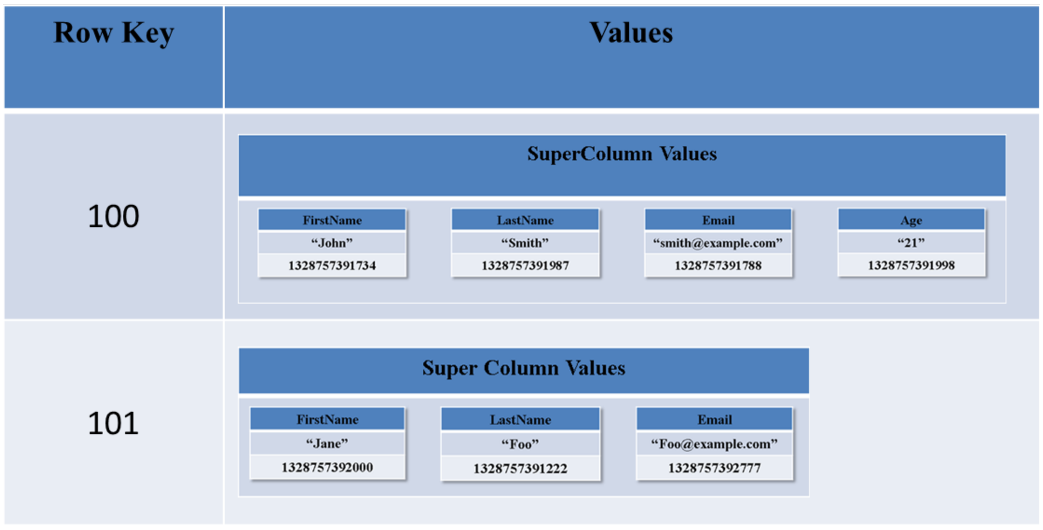
Column families can have rows that are identified by their unique row keys. This is similar to a table in an RDB, as seen for table `Student` in Figure 2.4, where every row in the table has the same number of columns and primary keys are used to identify a row. An example of a column family is shown in Figure 2.10. Unlike relational tables in an RDB, column families do not require all the rows to define the same number of columns [18, 33].

The JSON notation for a single row of a column family in Cassandra is shown in Figure 2.11

2.4.4 Keyspace

A keyspace is a container to hold the data that the application uses. Keyspaces have one or more column families, although it is not strictly required that a keyspace should always have column families. Any relationships existing between column families in a keyspace are not preserved.

A keyspace can be considered similar to a database in traditional rela-



Row Key	Values
100	SuperColumn Values: FirstName “John” 1328757391734; LastName “Smith” 1328757391987; Email “smith@example.com” 1328757391788; Age “21” 1328757391998
101	Super Column Values: FirstName “Jane” 1328757392000; LastName “Foo” 1328757391222; Email “Foo@example.com” 1328757392777

Figure 2.10: Column Family `User` in Cassandra

tional databases, without any relationships. An example of the keyspace University is shown in Figure 2.12.

Keyspaces require that some attributes are defined, like a user defined name, replication strategy and others. Some optional elements that can be defined are the details of the column families in the keyspace and other options for replication of data.

```
{//ColumnFamily
        “name”: “Student”,
        “value”: {//SuperColumns
                100 : {//SuperColumn
                        name: 100,
                        value: {//Columns
                                “FirstName”: {//Column
                                                “name”: “FirstName”,
                                                “value”: “John”,
                                                “timestamp”: 1328757391734
                                         }

                                “LastName”: {//Column
                                                “name”: “LastName”,
                                                “value”: “Smith”,
                                                “timestamp”: 1328757391987
                                         }
                                “Email”:  {//Column
                                                “name”: “Email”,
                                                “value”: “smith@example.com”,
                                                “timestamp”: 1328757391788
                                        }

                                “Age”: {//Column
                                                “name”: “Age”,
                                                “value”: 21,
                                                “timestamp”: 1328757391998
                                       }

                               }
                          }
                }
}
```

Figure 2.11: JSON notation for a column family in Cassandra

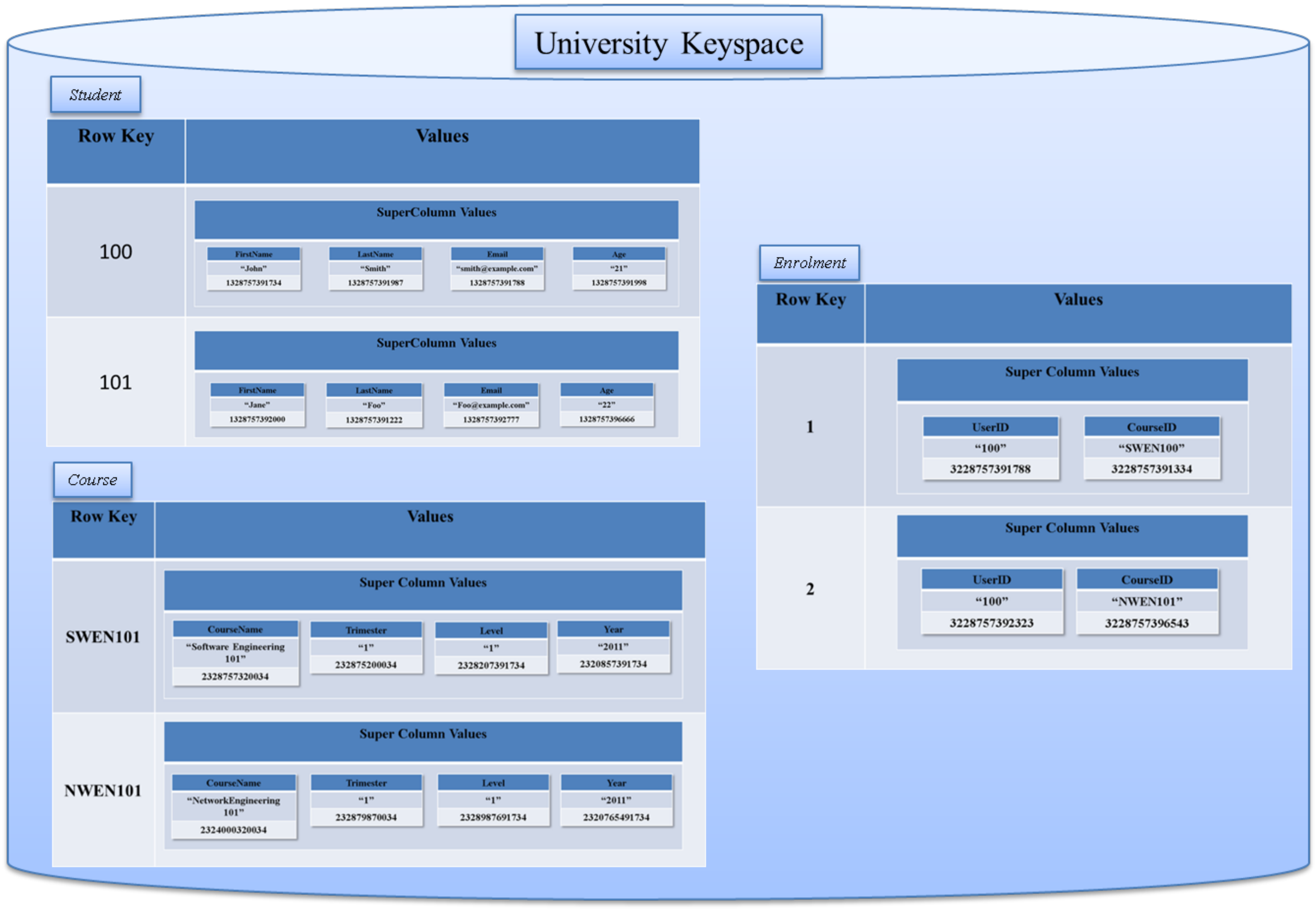


Figure 2.12: A keyspace in Cassandra

2.5 Challenges in the Key-Value Data Model

Fundamentally, the key-value data model is different from the relational model in many ways. While the relational data model aims at giving data a structure and providing data integrity, the key-value data model just store data as Binary Large Objectss (blobs) or string values and generally do not maintain relationships between data. In the column-oriented key-value model, the key-value association and the grouping of columns in column families can be considered as the minimum relationship that is maintained.

According to Bell and Brockhausen [3], data dependencies are the most common types of semantic constraints in relational databases and these determine the database design. Data dependencies are the various relationships that may exist between data entities in a database. For example, in the University database, a student can enrol into more than one course and this means that there is a many-to-many relationship between `Student` and `Course` since one course can have many students enrolled in it.

As seen in Section 2.4, the `Enrolment` table contains the `StudentID` and the `CourseID` as foreign keys, thus showing the dependency or relationship between students and courses (Figure 2.1). In the `University` RDB, any attempt to delete a course from the `Course` table is prevented by a constraint, unless the dependency itself is removed first. In RDBMSs, such constraints are the referential integrity constraints , which ensure that references between data entities are valid, consistent and intact [6, 21]. Normalisation, as well as modelling real world data and relationships enforce such dependencies in the schema and this causes integrity constraints like referential integrity constraints, to be imposed on data entities.

When such constraints are not imposed, the database is prone to dangling dependencies. Consider the case of foreign key references between `Course` and `Enrolment` in the University database. If a course is deleted from the `Course` table, without removing its dependencies in `Enrolment`, the latter will contain active references to the deleted course. Another example of a dangling reference occurs during insertion of data, where a new student is entered in the `Enrolment` table, with a `CourseID`, that does not exist in the `Course` table (i.e., wrong `CourseID`). A dangling reference occurs because this inserted student refers to a nonexistent course. Such problems violate data integrity and cause inconsistent data to be stored in databases. In order to ensure that users get consistent and valid information, applications have to implement mechanisms to check or prevent dangling references. However, if referential integrity constraints are applied as in NoSQL DBMSs, as in RDBMSs, operations on data that adversely affect referential integrity will not be permitted.

As previously mentioned, NoSQL DBMSs do not normalise data and nor are any relationships maintained. However, relationships or dependencies between data are common when real world data is stored in databases. For example, in the real world, a course can be taught by more than one lecturer or a student with an Art major is restricted entry into Chemistry courses etc. These relationships and constraints have to be preserved upon storage in cloud NoSQL database systems as well. As mentioned in Section 2.2, cloud databases (both relational and NoSQL, have to replicate data across several machines and need to be scalable to match the needs of the applications. The replicated and distributed nature makes maintaining data dependencies complex and unfeasible in terms of speed and efficiency. In cloud NoSQL DBMSs, this effectively means that the relationship between `Enrolment`, `Student` and `Course` will not be strictly enforced and deleting a course in cloud NoSQL DBMSs is allowed because of the absence of constraints. As mentioned before, this means that students could still be enrolled in deleted courses as there are no constraints to prevent such deletions or changes in cloud NoSQL DBMSs.

Commonly, developers impose such constraints and reference integrity checks on NoSQL data at the application side. Another way to implement such checks is to impose these constraints at the persistence layer of the application server. Both these ways eventually have to handle all the processing and managing of these constraint checks for all the widely spread data in NoSQL DBMSs. However, this could mean immense workload on the application or the application server, especially if the data volume is large in the NoSQL database or if it is has many replicas that have to be checked.

This is a serious problem when data is interconnected and dependant on other data entities as is commonly the case. For example, consider a banking application that uses cloud NoSQL DBMSs where its data is interconnected and spread across several nodes. Any debit or credit transactions made to a customer's account will have to be replicated across all the nodes and correctly persisted. In such applications, many constraints will exist for transfer of funds between user accounts and such constraints need to be validated correctly. If a user has multiple accounts, the relationship between the accounts have to be maintained. When such constraints are not validated correctly, it leads to incorrect account balances and wrong updates in the user accounts. On the other hand, when such applications use an RDBMS, referential integrity constraints are imposed to maintain the relationships between the accounts. In such cases, the referential integrity constraints are defined when tables are created and validations are triggered whenever any operations are performed on the data.

Although such problems affect most applications using cloud NoSQL DBMS, its impact is application dependant. For instance, a banking system as mentioned above could be gravely affected because of dangling references while in a simple game application such problems can be trivial. Motivated by such problems of data dependencies, this thesis studies the existing modelling of data dependencies in cloud NoSQL DBMSs and contributes by proposing four solutions to effectively maintain and validate referential integrity. The following section describes referential integrity and the rules that have to be imposed within a database system to validate referential integrity.

2.6 Referential Integrity in Key-Value Model

Referential integrity is a fundamental property of data within databases, which ensures that data dependencies between tables are maintained correctly in the database [6, 21, 25, 28]. These dependencies are generally a part of the business rule and are enforced using referential integrity constraints to ensure proper data integrity. These constraints have been a relational feature in traditional RDBMSs and are imposed due to the way the RDBMSs enforce normalisation. Such constraints are defined on the tables in a database, and have to be mandatorily satisfied at all times in order to ensure that users or applications do not enter incorrect or inconsistent data into the databases.

Generally in RDBMSs, referential integrity constraints ensure that the value of foreign keys in a table matches the values of primary keys in another table. That is, referential integrity is enforced by the combination of a primary (or unique) key and a foreign key such that every foreign key matches the primary key [6, 25, 28, 42]. In the University example, every foreign key in the `Enrolment` table must match one of the primary keys in the `Student` and `Course` tables. Hence, if any foreign key refers to a non-existing primary key, the referential integrity constraint is violated. For example, if '`StudID100`' is a foreign key for a student in the `Enrolment` table, but '`StudID100`' does not exist as a primary key in the `Student` table, it is a violation of referential integrity. Notice that the table containing the foreign key is the referencing table (or child table), while the table with the primary or unique key is the referenced table (or parent table). For example, `Enrolment` is the referencing table while `Student` and `Course` are the referenced tables. Foreign keys are also known as referencing keys and the primary keys as referenced keys.

Referential integrity constraints also describe the data manipulation that is allowed on the referenced values. Some of the widely associated rules are:

- `Restrict` or `No delete`: which prevents any update or deletion of data that has references.
- `Set to NULL`: which sets all foreign keys to NULL values, on updating or deleting the referenced key.
- `Set to Default`: which sets all the foreign keys to a default value, on updating or deleting the referenced key.
- `Cascade`: which updates or deletes all the associated dependant values accordingly, when the referenced data is updated or deleted.
- `No Action`: which performs checks only at the end of a statement and is similar to `Restrict`

Existing DBMSs may not always support all of the above rules. Some DBMSs may have the `Cascade` rule by default like Oracle, while some may have the `Restrict` rule by default.

Generally, in RDBMSs the database manager enforces a set of rules to prevent any data operation, like insert, update or delete, to change data in such a way that referential integrity is not violated as seen in Figure 2.13.

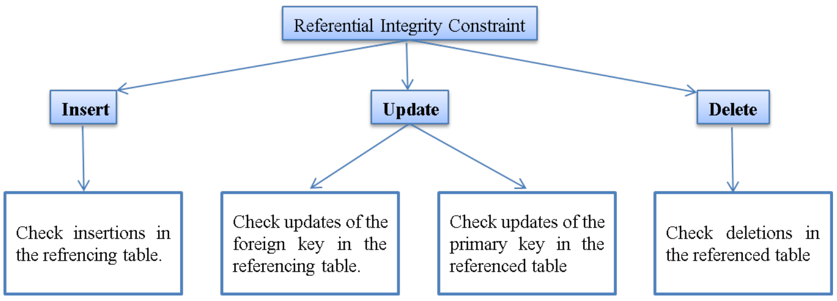

Referential Integrity Constraint
Insert
Update
Delete
Check insertions in the refrencing table.
Check updates of the foreign key in the referencing table.
Check updates of the primary key in the referenced table
Check deletions in the referenced table

Figure 2.13: Referential Integrity Rules

2.6.1 Insert rule

An insert operation triggers a referential integrity validation when data is being inserted into a referencing table, i. e. , the child table. In such an event, prior to entering the values in the referencing table, it is checked if the foreign keys exist in the referenced table. For example, in the University RDB, when a row is inserted in the `Enrolment` table with foreign key values for `StudentID` and `CourseID`, a check is triggered to verify whether these foreign keys exist in the `Student` and `Course` tables as primary keys. If the foreign keys do not exist in the referenced tables, then the insert operation is not allowed.

2.6.2 Update rule

When data is updated either in the referencing table or the referenced table, a referential integrity validation is required. When any primary key is updated in the referenced table, then it is verified whether this key is a foreign key in any of the referencing tables. If a dependency is found to exist in the referencing tables, then the applicable data manipulation rule is checked. For instance, if it is a `Cascade` rule, then the associated foreign keys in the referencing table are updated prior to updating the primary key in the referenced table. Consider the University RDB, if the primary key '`SWEN100`' for a course is updated to '`SWEN101`', then all the records in `Enrolment` that have '`SWEN100`' as a foreign key are updated to '`SWEN101`', if it has a `Cascade` rule.

When any foreign key is updated in a referencing table, then a referential integrity validation has to be performed. It is ensured that the new updated value exists as a primary key in the referenced table. For example, in the `Enrolment` table, if `CourseID` in a row is updated to a new value, then it is verified that the new value is an existing primary key in the `Course` table. If the new value does not exist, the update is not allowed generally.

2.6.3 Delete rule

A delete operation triggers a referential integrity validation when data is deleted from the referenced table. When data that is marked for deletion is found to have dependencies in other referencing tables, the data manipulation rule applicable for this operation is checked. That is, if the rule is `Cascade`, then the depending values in the referencing table have to be removed prior to deleting values from the referenced tables. For example, when a student record is deleted from the `Student` table, a check is performed to see if the `StudentID` is a foreign key in any other table. Therefore, `Enrolment` is checked and when the `StudentID` is found as a foreign key, the appropriate action is performed depending on the data manipulation rule. If it is `Cascade`, the enrolment details for the `StudentID` are removed from `Enrolment` and then the student record is deleted from `Student`.

2.7 Apache Cassandra

Cassandra is a distributed data storage system initially developed by Facebook for satisfying the needs of large web applications that handle large volumes of data [33]. Its development has been undertaken by Apache and it is currently used by many large web applications and large organisations like Facebook, Twitter, Cisco, Digg, Reddit, and others [20].

Cassandra is based on the column-oriented key-value data model and stores data as columns, super columns, column families and keyspaces, all of which are explained in Section 2.4. Being a distributed system, Cassandra can run on multiple machines and provides the option to work across different machines and across multiple data centers, even if these are geographically distributed [33]. These machines are configured to operate together and run as a single cluster, where these machines form a ring of nodes [17, 33]. Such nodes are connected to each other and each node is aware of all their peers in the cluster (Figure 2.14).

The nodes in a cluster communicate with each other to send their state information at regular time intervals, so that other nodes in the ring know their status [33, 38]. Such communication between the nodes support

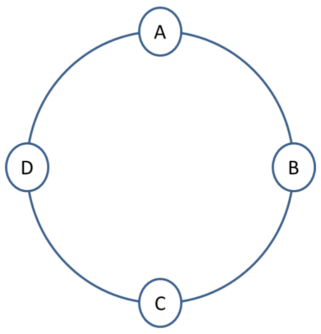


Figure 2.14: A cluster of nodes in Cassandra

failure detection in Cassandra as when a node fails, it stops responding to messages from other active nodes and in this way the rest of the nodes know of its inactive state. In the event of a node failure, operations sent to it are not lost since another active node ensures these are performed [38].

In such a cluster, every node applies the same architectural features fundamental to Cassandra, namely, load balancing, replicating and partitioning data, failure detection mechanisms, among others. Some of the key architectural concepts of Cassandra are explained next.

2.7.1 Architecture

Cassandra adopts many of its architectural concepts from other popular distributed key-value data storage systems on the cloud, like Google's Bigtable and Amazon's Dynamo [16, 22]. Over time, these adopted concepts evolved and developed new features, some of which became specific to Cassandra's architecture. Some of these concepts are, peer-peer distribution model, data partitioning, eventual consistency, among others. These concepts gave Cassandra features such as elastic scalability, fault tolerance, high availability and high performance.

Peer-Peer Distribution Model

Cassandra is a decentralised system where all the nodes are considered equal or identical (i.e. nodes are peers) in sharing responsibilities and performing operations, without any master or slave nodes [20, 33]. This model provides high data availability since failure of a node does not affect the service of the cluster because other nodes carry out the same operation.

Data Partitioning

Cassandra partitions data between the nodes in a cluster so that data items from overloaded or failed nodes are assigned to other nodes or new nodes. For this, Cassandra uses consistent hashing where data items are hashed on its key. After hashing the key, the data items are assigned to the node whose position in the ring is larger than the hashed value of the key [33]. Data partitioning makes Cassandra elastically scalable since the load is balanced and distributed in the cluster irrespective of addition or removal of nodes [33, 38].

Replication strategy

In order to ensure high data availability irrespective of failures, Cassandra uses a replication strategy where every data item is replicated across a number of nodes. Applications can set the level of replication to suit its requirements, that is, the replication factor is set to the number of nodes on which the application wants to create replicas [33, 38]. The replication factor tells the cluster how many copies to create of a single data item. Setting the replication factor to a large number will help in higher consistency of data items, but replicating data items to a large number of nodes can adversely affect the performance.

Once data items are partitioned and assigned to a node, these are replicated onto other nodes and a list of the nodes responsible for storing the data items are maintained. Thus, every node in the cluster knows which nodes are responsible for a data item [33, 38]. Such a replication strategy makes data highly available since data items can be accessed from any node in a cluster regardless of node failures.

Eventual Consistency

In any strongly consistent DBMSs, data items immediately reflect the new values upon an insert or update operation. However, Cassandra uses the eventual consistency model where replicas do not agree to the most recent value immediately but will do so eventually. This is because the new values are propagated to all the replicas in a cluster in an asynchronous way [4, 16, 32, 54].

2.7.2 Write and Read Operations

In order to write data into Cassandra column families, a write request is sent to a random node in the cluster, which acts as a proxy node and replicates the data in the cluster [17, 33, 38]. The number of nodes on which data is to be replicated can be changed to suit the application requirements. Moreover, these nodes can be in the same data centre and other data centres.

When a read request is issued to a node, it acts as a proxy node and forwards the request to all the other nodes in the cluster. These nodes return their copy of the data item to the proxy node and the proxy node checks the versions of the replicas and sends the latest replica to the user [19]. If the replicas received from a node are not consistent with other replicas, a read repair is performed on the node with the outdated replicas. This means that the nodes with outdated replicas are sent a write operation with the latest data. Thus, data consistency is maintained whenever conflicting versions of data items are found.

When rows or columns are deleted in Cassandra, the data within these rows or columns is not removed immediately from the disk [17, 33]. Instead, data is deleted after a time period that is configurable by applications. This is called a tombstone delete in Cassandra, where the columns that are to be deleted are only marked for deletion and empty values are written into such columns. Once the configured time period expires, the data is physically removed from the disk. However, the row keys of the deleted columns continue to persist. Such a tombstone delete is useful when a failed node is active again as this node can update its replicas correctly to show the deletions.

2.8 Summary

This chapter presented the background about the underlying concepts in cloud computing and cloud databases. It is clear that cloud computing is gaining prevalence due to its many benefits like high data availability, cheap storage, and others. With an increase in the number of users migrating to cloud computing, cloud data storage is gaining prominence as well, for easy and simple data storage. This has paved the way for the existence of many different data models and databases on the cloud. Amongst the many data models, the key-value data model has been most widely used on the cloud as it is more adapted to the cloud environment due to its support for replication and scalability and other cloud related features [51, 52].

This chapter also discussed a few architectural concepts of Cassandra that form its foundation , providing it with BASE properties and many important features like high data availability, failure management, fault tolerance and scalability among others. The architectural concepts and operations of Cassandra are designed to make it a highly available and scalable DBMS. Cassandra is used to implement the four solutions designed to impose referential integrity validations in cloud NoSQL DBMSs. The design of these solutions and the approaches used to implement such validations are explained in the following chapter.

Chapter 3

Design of Referential Integrity Constraints in NoSQL databases

Traditionally, referential integrity constraints are imposed on data items of a database to maintain foreign key relationships. These relationships are maintained by correctly identifying and preserving the data dependencies existing between the data items. Most popular traditional Relational Database Management Systems (RDBMSs) preserve such dependency information in their `System` tables or data dictionaries. These tables store the necessary information which is required to maintain valid dependencies. The information stored in such tables include table names, primary and foreign keys, among others. This can be seen in popular RDBMSs like MS SQL Server, PostgreSQL, Oracle, and so on.

For example, in MS SQL Server 2000, `sysforeignkeys` is a `System` table which stores the information of all foreign keys of every table in a database, and `sysreferences` stores the mappings of foreign keys to the referenced primary key columns [41]. Information in these `System` tables consist of the names of tables and its constraints, unique identifiers of referenced and referencing columns and others. In PostgreSQL, such information is presented to users as views but it is stored in base tables which contain the dependency information of data items in a database. The view `table_constraints` show the information about all the constraints in every table owned by the current user [43]. Similarly, Oracle uses a `SYSTEM` meta-database to hold such constraint information. In general, `System` tables or views with information about the existing dependencies are looked up by these RDBMSs whenever referential integrity checks are triggered [41].

The solutions presented in this thesis save the dependency information as metadata. This metadata contains relevant information about primary keys of column families and foreign key relationships in keyspaces. Thus, metadata is accessed whenever an operation is performed on the data and referential integrity needs to be validated. These solutions are implemented using an experimental Application Programming Interface (API) which is discussed in Chapter 4.

This chapter presents the design of four solutions that implement referential integrity constraints in a cloud Not only SQL (NoSQL) Database Management System (DBMS). Section 3.1 describes the metadata used by the solutions to store the dependency information. Sections 3.2, 3.3, 3.4, 3.5 present the design and motivation of the four solutions. Section 3.6 summarises the design of the four solutions.

3.1 Metadata

Metadata in DBMSs provide information about the data stored within the databases. It may contain details related to schemas, constraints, primary and foreign keys, and so on. As previously mentioned, most traditional RDBMSs maintain such metadata within their `System` tables or data dictionaries. In Apache Cassandra, the DBMS of interest, metadata is stored in a keyspace named `System` and it contains information about the cluster and its nodes along with information related to the keyspaces, column families, and so on [33]. Even when Cassandra has a `System` keyspace to store metadata, it is read-only and therefore it cannot be modified to store additional metadata about referential integrity constraints. Hence, for preserving the metadata, each of the solutions implement a different strategy in which metadata is associated with actual data. Solutions 1 and 2 use embedded metadata, that is, metadata is created with the actual data; while solutions 3 and 4 associate metadata separately from the actual data. Notice that, the structure of the metadata is kept the same across all the solutions even when the way of storing and associating this metadata is different in each.

The role of metadata in the solutions is primarily to hold the necessary information required to maintain referential integrity. The metadata contains information about primary keys, foreign keys, referenced and referencing column family details, constraints, and others. The constraints considered in the solutions can be either Primary Key (PK) or Foreign Key (FK) constraints. PK constraints specify which column is the primary key of a column family. FK constraints (or referential integrity constraints) determine the foreign key relationship between two column families, that is, the column of a column family which is dependent on the primary key column of another column family. Hence, for each column family with a primary key, the metadata contains one PK constraint and as many FK constraints as foreign key relationships the column family has.

The structure of the metadata is shown in Table 3.1. This structure contains information about a University keyspace example in which a simple schema is applied for the keyspace. In this example, the details of the students are saved in the `Student` column family and the course details in the `Course` column family. The enrolment details of students are saved in the `Enrolment` column family by associating students to courses and hence having foreign key relationships to both `Student` and `Course` column families. All the column families have unique primary keys and their PK constraints are saved in the metadata as presented in Table 3.1 while the foreign key relationships between `Enrolment`, `Student` and `Course` are saved as FK constraints. For instance, consider in Table 3.1

the PK constraint `CONST100`, for the `Student` column family, and the FK constraint `CONST400` for the foreign key relationship between `Enrolment` and `Student`. Notice that only single column primary keys are considered in the solutions.

Constraint Name	**Keyspace**	**Constraint Type**	**Column Family**	**RKeyspace**	**RConstraint Name**	**RColumn**	**DeleteRule**
CONST100	University	P	Student	University		StudentId	
CONST200	University	P	Course	University		CourseId	
CONST300	University	P	Enrolment	University		RowId	
CONST400	University	R	Enrolment	University	CONST100	StudentId	CASCADE
CONST500	University	R	Enrolment	University	CONST200	CourseId	NODELETE
CONST600	University	F	Course	University	CONST500	CourseId	NODELETE
CONST700	University	F	Student	University	CONST400	StudentId	CASCADE

Table 3.1: Metadata for the Solutions

Specifically, the structure of the metadata contains:

- `ConstraintName`: is the name assigned for every constraint and it uniquely identifies an existing PK or FK constraint in the metadata. For example, `CONST100` and `CONST400` are `ConstraintNames`.
- `Keyspace`:represents the name of the Keyspace the constraint belongs to.
- `ConstraintType`: denotes the type of the constraint and the possible values are ′`P`′, ′`R`′ and ′`F`′. A PK constraint is referred by ′`P`′, while ′`R`′ and ′`F`′ are two representations of FK constraints. ′`R`′ represents the referential integrity constraint (or FK constraint) a child entity has on a parent primary key, and ′`F`′ represents the existing dependencies on a parent entity . For example, `CONST400` shows that the parent entity for `Enrolment` is `Student` by looking up `RConstraintName`. `CONST700` shows that parent entity `Student` has child dependencies on it by following `CONST400`. Notice that, the constraint type ′`F`′ is

primarily used to locate the child dependencies for a parent when it is deleted or updated.

- `ColumnFamily`: refers to the column family this constraint applies to. For example, the PK constraint `CONST100` applies on column family `Student`, and the FK constraint `CONST400` applies on `Enrolment`.

- `RKeyspace`: is the name of the keyspace on which this constraint is applied. In the example, all the constraints are applied in the keyspace `University`.

- `RConstraintName`: represents the constraint that is referenced. For the constraint type '`R`', this represents the referenced PK constraint; and for the constraint type '`F`', it shows the child dependencies for a parent entity. In the example, the FK constraint `CONST400` references the PK constraint `CONST100`, which means that `Enrolment` has a foreign key relationship with `Student`. In `CONST700` this field indicates that FK constraint `CONST400` exists for `Student`. Notice that this field is left blank in a PK constraint since it has no references to other keys.

- `RColumn`: indicates the primary key column on which this constraint is applicable. For PK constraints, this holds the name of the primary key column. For FK constraints, this field denotes the referenced column. This example shows that the PK constraint `CONST100` is applied on the primary key column `StudentId` of `Student` column family . The FK constraint `CONST400` shows that the referenced column is `StudentId`, indicating that `Enrolment` references primary key column `StudentId` of `Student`.

- `DeleteRule`: stores the type of data manipulation rule applicable on this constraint for a delete operation. Notice that for the sake of simplicity, this rule is also used for update operations. The possible rules considered in this thesis are `Cascade` and `NoDelete`, other rules such as `Null` or `Default` are out of the scope of this thesis. Notice that, this field is not applicable for PK constraints since data manipulation rules are associated with constraints that hold dependency information like the FK constraints.

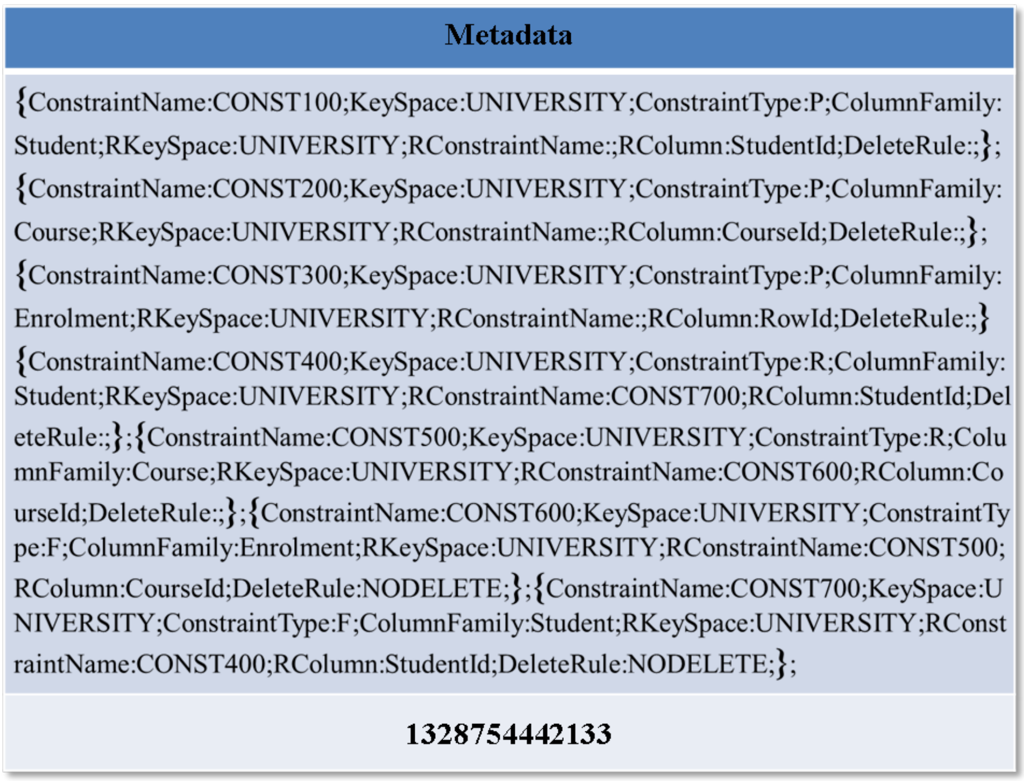
In the solutions, metadata is accessed whenever referential integrity validations are triggered by Create, Read, Update and Delete (CRUD)[1] operations performed on a column family. Thus, the relevant FK constraints to perform such validations are accessed from the metadata. Notice that, each solution stores metadata in a distinct way and provides specific methods to access and process the metadata to support the validation. The logic for validating the referential integrity is consistent across all the solutions. The way these solutions store metadata and the motivation behind the design of its metadata storage is presented in the following sections.

3.2 Solution 1: Metadata in Super Columns

In this solution, metadata is embedded with the actual data by storing it within each super column. That is, each super column of a column family stores the metadata in its `Metadata` column (Figure 3.1). Since metadata is common in a keyspace, all the super columns in every column family contains the same value in the `Metadata` column. Figure 3.2 presents how metadata is stored in every super column of the example `Student` column family in the University keyspace.

Metadata contains all the parts as described in Section 3.1 and each super column stores all the constraints belonging to the keyspace in its `Metadata` column. Since all the constraints are stored together, it is possible to retrieve all the relevant constraints of a super column from its `Metadata` column.

[1]Notice that `Read` does not trigger any referential integrity validations, but the CRUD acronym is still used all throughout this thesis for the sake of familiarity.



Metadata

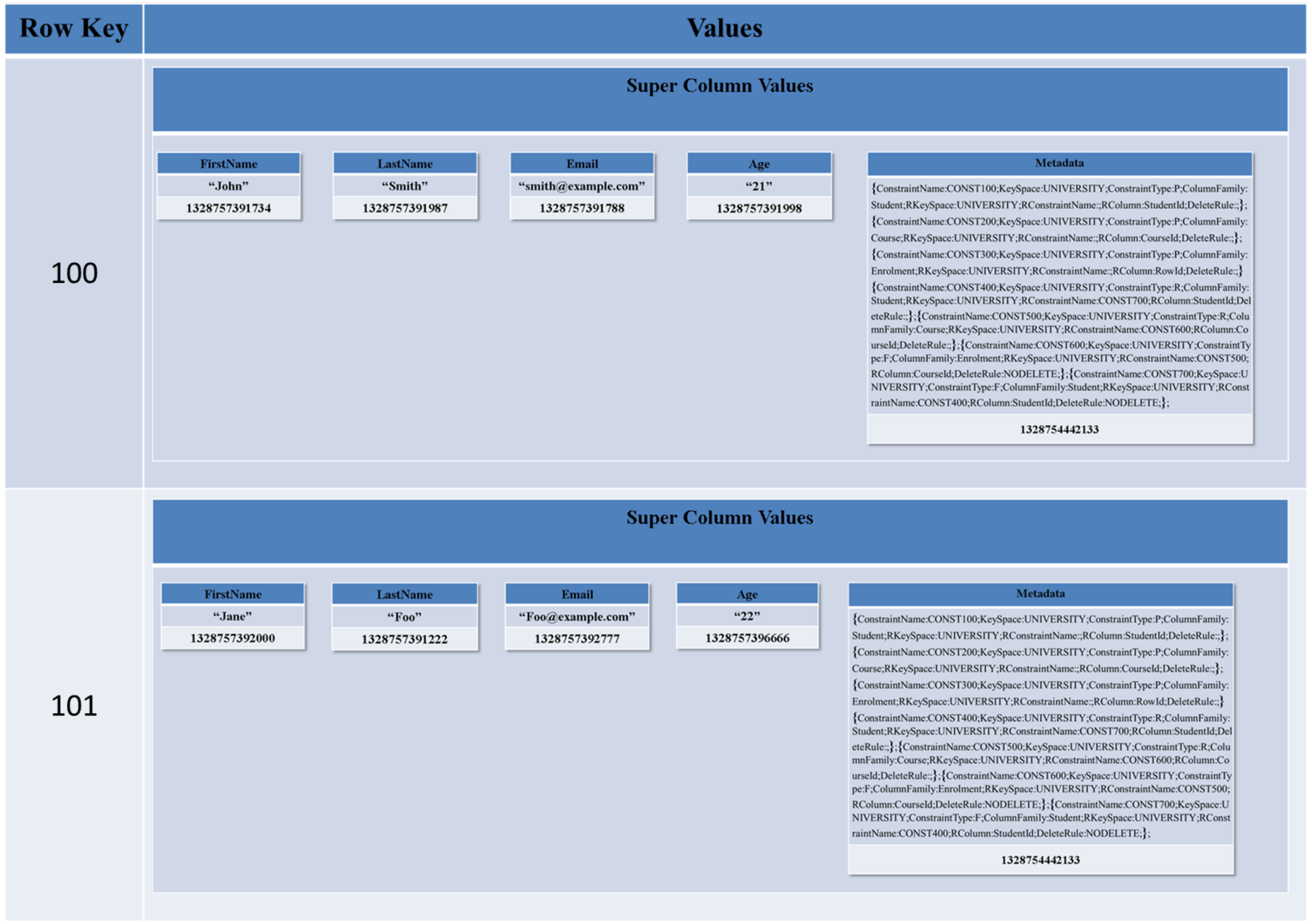
{ConstraintName:CONST100;KeySpace:UNIVERSITY;ConstraintType:P;ColumnFamily:
Student;RKeySpace:UNIVERSITY;RConstraintName:;RColumn:StudentId;DeleteRule:;};
{ConstraintName:CONST200;KeySpace:UNIVERSITY;ConstraintType:P;ColumnFamily:
Course;RKeySpace:UNIVERSITY;RConstraintName:;RColumn:CourseId;DeleteRule:;};
{ConstraintName:CONST300;KeySpace:UNIVERSITY;ConstraintType:P;ColumnFamily:
Enrolment;RKeySpace:UNIVERSITY;RConstraintName:;RColumn:RowId;DeleteRule:;}
{ConstraintName:CONST400;KeySpace:UNIVERSITY;ConstraintType:R;ColumnFamily:
Student;RKeySpace:UNIVERSITY;RConstraintName:CONST700;RColumn:StudentId;Del
eteRule:;};{ConstraintName:CONST500;KeySpace:UNIVERSITY;ConstraintType:R;Colu
mnFamily:Course;RKeySpace:UNIVERSITY;RConstraintName:CONST600;RColumn:Co
urseId;DeleteRule:;};{ConstraintName:CONST600;KeySpace:UNIVERSITY;ConstraintTy
pe:F;ColumnFamily:Enrolment;RKeySpace:UNIVERSITY;RConstraintName:CONST500;
RColumn:CourseId;DeleteRule:NODELETE;};{ConstraintName:CONST700;KeySpace:U
NIVERSITY;ConstraintType:F;ColumnFamily:Student;RKeySpace:UNIVERSITY;RConst
raintName:CONST400;RColumn:StudentId;DeleteRule:NODELETE;};

1328754442133

Figure 3.1: Metadata Column in Solution 1

Since all the constraints of a column family are saved together in the `Metadata` column, it is essential for each constraint to be easily identifiable and accessible. Thus, special characters are used within the metadata to separate the constraints and to identify its different parts, as shown in Figure 3.1. These characters are curly brackets ('{', '}') and semi-colon and colon (';', ':'). These special characters are used as follows.

- Each constraint is enclosed within curly brackets and the constraints are separated from each other with a ';'. For example, `CONST100` and `CONST200` are enclosed in curly braces and separated by ';'. Thus, '};' marks the end of every constraint in the metadata.



```
{ConstraintName:CONST100;KeySpace:UNIVERSITY;ConstraintType:P;ColumnFamily:
Student;RKeySpace:UNIVERSITY;RConstraintName:;RColumn:StudentId;DeleteRule:;};
{ConstraintName:CONST200;KeySpace:UNIVERSITY;ConstraintType:P;ColumnFamily:
Course;RKeySpace:UNIVERSITY;RConstraintName:;RColumn:CourseId;DeleteRule:;};
{ConstraintName:CONST300;KeySpace:UNIVERSITY;ConstraintType:P;ColumnFamily:
Enrolment;RKeySpace:UNIVERSITY;RConstraintName:;RColumn:RowId;DeleteRule:;}
{ConstraintName:CONST400;KeySpace:UNIVERSITY;ConstraintType:R;ColumnFamily:
Student;RKeySpace:UNIVERSITY;RConstraintName:CONST700;RColumn:StudentId;Del
eteRule:;};{ConstraintName:CONST500;KeySpace:UNIVERSITY;ConstraintType:R;Colu
mnFamily:Course;RKeySpace:UNIVERSITY;RConstraintName:CONST600;RColumn:Co
urseId;DeleteRule:;};{ConstraintName:CONST600;KeySpace:UNIVERSITY;ConstraintTy
pe:F;ColumnFamily:Enrolment;RKeySpace:UNIVERSITY;RConstraintName:CONST500;
RColumn:CourseId;DeleteRule:NODELETE;};{ConstraintName:CONST700;KeySpace:U
NIVERSITY;ConstraintType:F;ColumnFamily:Student;RKeySpace:UNIVERSITY;RConst
raintName:CONST400;RColumn:StudentId;DeleteRule:NODELETE;};
1328754442133
```

```
{ConstraintName:CONST100;KeySpace:UNIVERSITY;ConstraintType:P;ColumnFamily:
Student;RKeySpace:UNIVERSITY;RConstraintName:;RColumn:StudentId;DeleteRule:;};
{ConstraintName:CONST200;KeySpace:UNIVERSITY;ConstraintType:P;ColumnFamily:
Course;RKeySpace:UNIVERSITY;RConstraintName:;RColumn:CourseId;DeleteRule:;};
{ConstraintName:CONST300;KeySpace:UNIVERSITY;ConstraintType:P;ColumnFamily:
Enrolment;RKeySpace:UNIVERSITY;RConstraintName:;RColumn:RowId;DeleteRule:;}
{ConstraintName:CONST400;KeySpace:UNIVERSITY;ConstraintType:R;ColumnFamily:
Student;RKeySpace:UNIVERSITY;RConstraintName:CONST700;RColumn:StudentId;Del
eteRule:;};{ConstraintName:CONST500;KeySpace:UNIVERSITY;ConstraintType:R;Colu
mnFamily:Course;RKeySpace:UNIVERSITY;RConstraintName:CONST600;RColumn:Co
urseId;DeleteRule:;};{ConstraintName:CONST600;KeySpace:UNIVERSITY;ConstraintTy
pe:F;ColumnFamily:Enrolment;RKeySpace:UNIVERSITY;RConstraintName:CONST500;
RColumn:CourseId;DeleteRule:NODELETE;};{ConstraintName:CONST700;KeySpace:U
NIVERSITY;ConstraintType:F;ColumnFamily:Student;RKeySpace:UNIVERSITY;RConst
raintName:CONST400;RColumn:StudentId;DeleteRule:NODELETE;};
1328754442133
```

Figure 3.2: Metadata in Solution 1

- The different parts in a constraint are separated by the special character `';'`. For example, the `ConstraintName` and `Keyspace` and other parts in the constraints `CONST100` and `CONST200` are separated with a `';'`.
- Each part and its respective value are separated by the special character `':'`. For example, `ConstraintName` is separated from its value `CONST100` with a `':'`.

The special characters help in identifying the values of every constraint in the metadata information for this solution. Thus, the metadata is extracted from each super column and processed by specific methods within the solution so that relevant constraints are used for validating referential integrity.

This design was inspired by the experiments done by Hackl et al. [30] on a popular NoSQL DBMS named Tokyo Cabinet. Tokyo Cabinet is similar to Cassandra as data is stored in key-value pairs but it does not involve data types or columns and column families as in Cassandra [30, 35]. As a part of their experiments to manage metadata for huge file systems, they adopted an approach to store metadata as a part of the value in a key-value pair. In their approach, this value is associated with a unique key and the different parts of the metadata are separated by semicolons. Their results showed that such a metadata storage provided high speed metadata access where valuable information was integrated with the actual data.

Solution 1 derives this method of saving metadata using special characters and integrating it with the actual data as value in a key-value pair in Cassandra. The metadata in this solution contains all the constraints pertinent to a keyspace and uses the special characters to distinguish the relevant constraints and its various parts.

3.3 Solution 2: Metadata as a Top Row

In Solution 2, metadata is embedded with the actual data and exists within the same column family as the actual data, which is similar to Solution 1. However, in this solution, metadata is saved only once in the column family as a top row. Specifically, it is stored in the first super column in a column family) with the unique `RowId` '`-1`'. This top row has only the `Metadata` column which contains the metadata information as its value while the rest of the super columns in the same column family have different columns containing the actual data. This design is possible since Cassandra allows rows to have different number of columns within a column family. The `Metadata` column is similar to the Solution 1 as shown in Figure 3.1. Thus, for each column family, the metadata exists only once as a single row and is common for all its super columns. In the University example, the `Student` column family has the metadata stored as a top row as shown in Figure 3.3.

In this solution, metadata for each column family contains all the constraints belonging to the keyspace. The metadata contains the same special characters '{', '}', '`;`' and '`:`' to distinguish all the constraints and its different parts and values, as seen in Solution 1. For example, the metadata for `Student` contains all the constraints as listed in Table 3.1 as its metadata in the top row as shown in Figure 3.3.

The motivation behind this solution is to overcome the redundancy of metadata storage in Solution 1 where metadata is stored in every super column of a column family and replicated across the cluster along with the column family. Solution 2 reduces this redundancy and centralises the metadata as a top row within the column family. Thus, when metadata is large, lesser space is consumed since it is not replicated as widely as in Solution 1. Furthermore, this solution ensures that, when changes are made to the metadata, the actual data is not accessed and only the column family is accessed to fetch the top row containing the metadata.

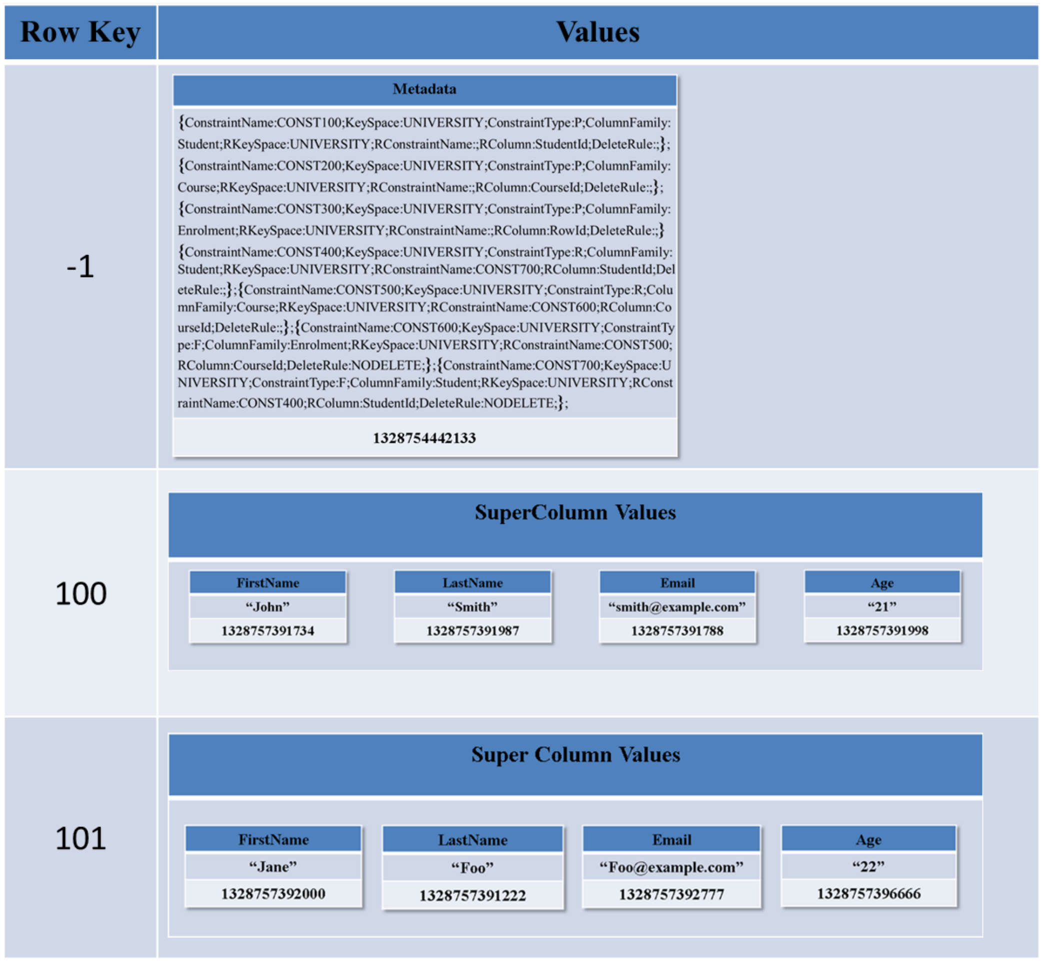

Row Key
Values
-1
Metadata
{ConstraintName:CONST100;KeySpace:UNIVERSITY;ConstraintType:P;ColumnFamily:Student;RKeySpace:UNIVERSITY;RConstraintName:;RColumn:StudentId;DeleteRule:;};{ConstraintName:CONST200;KeySpace:UNIVERSITY;ConstraintType:P;ColumnFamily:Course;RKeySpace:UNIVERSITY;RConstraintName:;RColumn:CourseId;DeleteRule:;};{ConstraintName:CONST300;KeySpace:UNIVERSITY;ConstraintType:P;ColumnFamily:Enrolment;RKeySpace:UNIVERSITY;RConstraintName:;RColumn:RowId;DeleteRule:;}{ConstraintName:CONST400;KeySpace:UNIVERSITY;ConstraintType:R;ColumnFamily:Student;RKeySpace:UNIVERSITY;RConstraintName:CONST700;RColumn:StudentId;DeleteRule:;};{ConstraintName:CONST500;KeySpace:UNIVERSITY;ConstraintType:R;ColumnFamily:Course;RKeySpace:UNIVERSITY;RConstraintName:CONST600;RColumn:CourseId;DeleteRule:;};{ConstraintName:CONST600;KeySpace:UNIVERSITY;ConstraintType:F;ColumnFamily:Enrolment;RKeySpace:UNIVERSITY;RConstraintName:CONST500;RColumn:CourseId;DeleteRule:NODELETE;};{ConstraintName:CONST700;KeySpace:UNIVERSITY;ConstraintType:F;ColumnFamily:Student;RKeySpace:UNIVERSITY;RConstraintName:CONST400;RColumn:StudentId;DeleteRule:NODELETE;};
1328754442133
100
SuperColumn Values
FirstName
"John"
1328757391734
LastName
"Smith"
1328757391987
Email
"smith@example.com"
1328757391788
Age
"21"
1328757391998
101
Super Column Values
FirstName
"Jane"
1328757392000
LastName
"Foo"
1328757391222
Email
"Foo@example.com"
1328757392777
Age
"22"
1328757396666

Figure 3.3: Metadata storage in Solution 2

3.4 Solution 3: Metadata Column Family

In Solution 3, metadata for all the column families in a keyspace is stored in a separate column family called `Metadata`. In this approach, the metadata is decoupled from the actual data and stored in a centralised way where all the PK and FK constraints of all the column families within a keyspace are saved in a single location. The other column families contain only the actual data and do not store any metadata. Using this approach, all the existing constraints are saved as super columns in the `Metadata` column family. Figure 3.4 shows an example of the `Metadata` column family with some of the constraints of the University keyspace.

The different parts of the constraints are saved as separate columns in the `Metadata` column family. Thus, no special characters are required to identify the various parts as seen in Solutions 1 and 2. When an operation is invoked on a column family in the keyspace referential integrity validations are triggered. For these validations, it is necessary to connect to `Metadata` column family and retrieves the relevant constraints for the column family on which the operation is invoked. Thus, the different parts of the constraints are accessed by identifying the correct columns in the `Metadata` column family and necessary values are retrieved to do the validation.

This approach is similar to the way dependency information is stored in traditional RDBMSs, where metadata holds information about tables, its dependencies and its many other properties. Commonly, such metadata is maintained in `System` tables, separated from the tables containing the actual data, as seen in this approach.

The design to decouple metadata from the actual data is inspired from the potential challenges in Solutions 1 and 2 where a column family with several constraints will have a large value in the `Metadata` column, hence making it cumbersome to maintain such metadata within a single string. Moreover, in these solutions metadata has to be changed at every place it is repeated in the event of any alterations to the metadata. Consider Solution 1, where the `Metadata` column in every super column of every column family has to be updated every time a constraint is added, removed or changed; similarly, in Solution 2 where the top row has to be updated for all the column families.

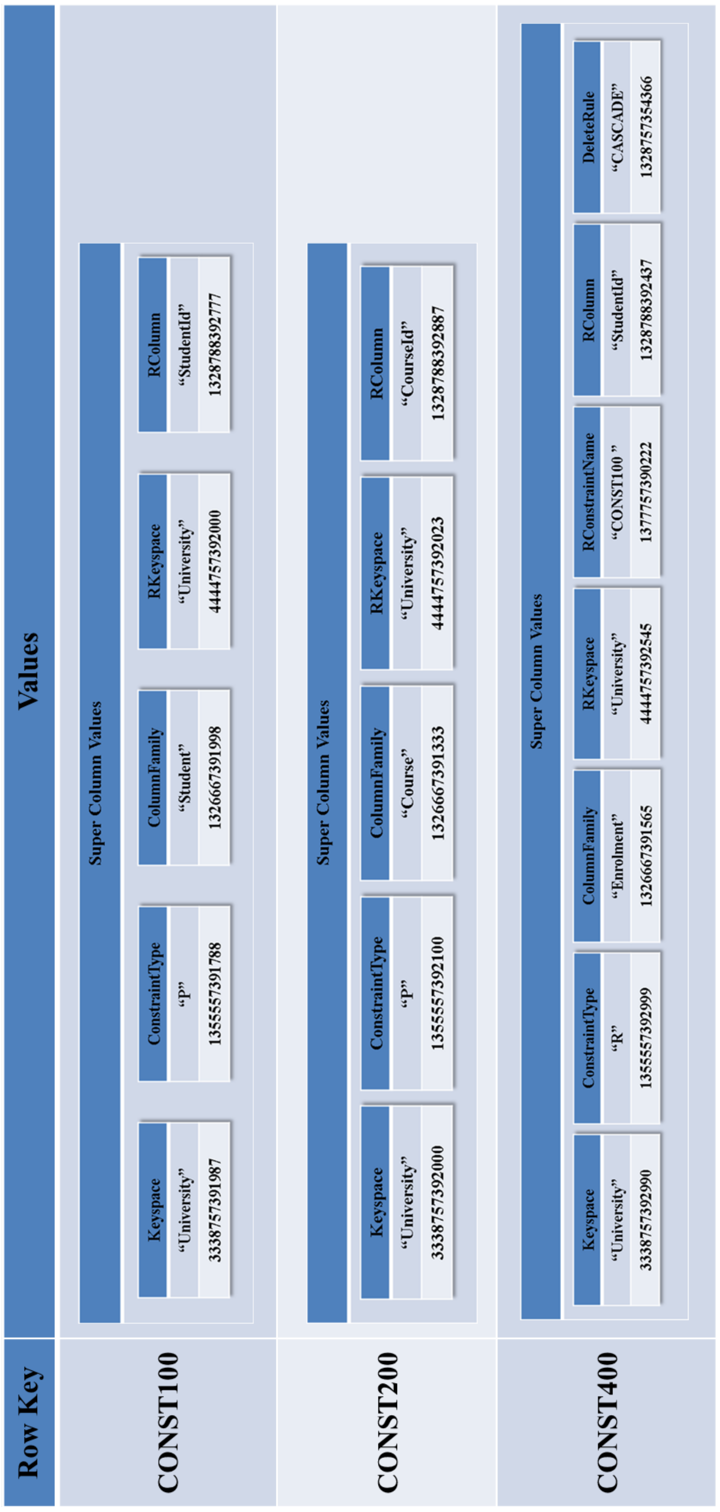


Figure 3.4: Metadata Column Family in Solution 3

Decoupling the metadata from the actual data allows accessing and retrieving the various parts of the constraints by fetching the respective column names. More importantly, adding, removing or changing constraints require access only to the centralised metadata. In such cases, changes affect only the `Metadata` column family and access to the actual data is not needed to perform such changes.

3.5 Solution 4: Metadata Cluster

In Solution 4, metadata is stored in a separate column family similar to Solution 3. However, in this solution, the `Metadata` column family is located in a separate Cassandra cluster instead of within the same cluster. Since such a cluster does not require many nodes, metadata is not as widely replicated as in the previous solutions since the replication of `Metadata` is only within the metadata cluster. Figure 3.5 shows an example of how the University keyspace is saved in a separate cluster (nodes `A`, `B`, `C`, `D`) and the metadata is saved in the separate cluster called Metadata cluster (nodes `L`, `M`, `N`, `O`). In this example, `Metadata` is inserted into `MetadataCluster`, while the column families `Student`, `Course` and `Enrolment` are stored into another cluster (`KeyspaceCluster`). .

In this case, it is necessary to connect to the Metadata cluster and to the cluster containing the keyspace to perform any operations on the actual data. Whenever a CRUD operation is invoked on a column family, the relevant metadata must be accessed from the `MetadataCluster`. However, since such external connection requires more time, a cache is implemented to re-use it for future operations on this column family. The advantages of a cache is that if the metadata cluster becomes unresponsive or not active, the metadata can still be retrieved from the cache to continue performing validations despite such disruptions. Having metadata cached is effective since metadata is not expected to be as frequently changed as the actual data. This also saves operational time by not having to connect to the

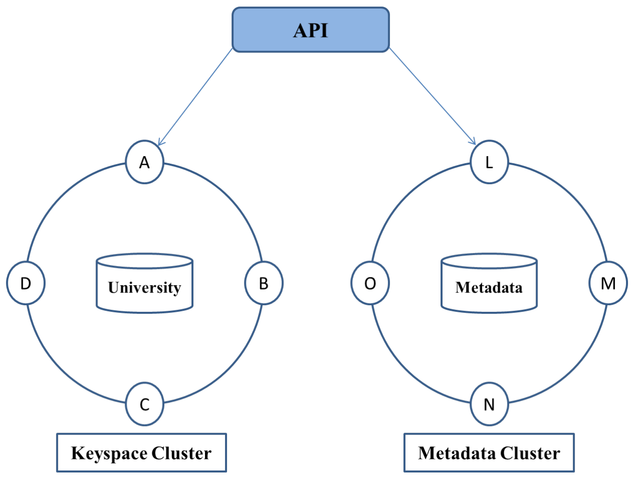


Figure 3.5: Metadata Cluster in Solution 4

`Metadata` column family each time metadata is accessed.

This approach is inspired from the way most distributed systems save metadata in Metadata Server clusters [5, 27, 58]. For better scalability and efficient access of metadata, these clusters are often separately maintained in large distributed environments where master and subordinate metadata servers handle various responsibilities within the cluster. In Solution 4 such delegations of tasks are not required since all nodes in a Cassandra cluster have the same responsibilities and do not have a master-slave configuration. Thus, Solution 4 adopts the design of having a cluster of dedicated nodes to save metadata information and to have a central location to preserve and maintain metadata for any number of keyspaces.

3.6 Summary

The main difference in the design of all the solutions is the way each of the solutions store metadata. Solution 1 stores metadata with the data in every super column providing fast access to the metadata but increasing its redundancy. Solution 2 has a similar approach as Solution 1 but stores metadata in a single super column in every column family. This approach is useful when the metadata is large since it consumes less space when compared to Solution 1. Both Solutions 1 and 2 use special characters in the metadata in order to distinguish the constraints relevant to an entity. Solution 3 separates the metadata from the actual data and stores all the constraints together for all column families of a keyspace in a centralised way. This is useful when metadata has to be altered because it simplifies the management of metadata. Solution 4 has a similar approach as Solution 3, but saves the `Metadata` column family in a separate cluster on different nodes. It also caches the metadata to save operational time and reduce database connections to a different cluster.

The next chapter presents the experimental API which provides methods for retrieving and processing the metadata as well as handling all the CRUD operations and referential integrity validations.

Chapter 4

Implementation of Referential Integrity Constraints in NoSQL DBMSs

The four solutions presented in this thesis store the referential integrity constraints of column families as metadata, and each solution stores it in its own unique way. Such metadata is accessed whenever referential integrity validations are triggered, that is, when Create, Read, Update and Delete (CRUD)[1] operations are invoked on column families. The validations require to access the relevant Foreign Key (FK) constraints of a column family and its associated Primary Key (PK) constraints from its metadata and processed in order to extract the values held in the constraints. The design of the four solutions presented in the previous chapter was shown as an abstract representation of the way metadata is stored.

The metadata storage is different in every solution, and retrieving it and processing it is different in each solution as well. Specific methods are designed in all the solutions to retrieve and process the metadata, and these methods along with all the solutions are incorporated into a single

[1]Notice that `Read` does not trigger any referential integrity validations, but the CRUD acronym is used for the sake of familiarity.

experimental Application Programming Interface (API). The experimental API implements the design of the solutions and provides handlers to execute the CRUD operations and perform referential integrity validations.

This chapter describes the experimental API and the implementation details of the four solutions. Section 4.1 presents the experimental API and describes its most relevant components. Section 4.2 describes the approaches taken by the four solutions to retrieve and process metadata. Lastly, Section 4.3 presents a summary of the chapter.

4.1 Experimental API

The experimental API[2] was designed generically in order to ensure that it can be used by applications to maintain dependencies within their keyspaces irrespective of keyspace schemas or structures of column families. However, applications using this API still have to supply the list of referential integrity constraints as these are not automatically deduced from their implementation. Instead, the constraints have to be introduced according to the solution as it is explained in detail in Section 4.2.

This API validates the referential integrity based on the metadata provided for the application and its column families. It provides the implementation of all the four solutions as well as the required components to successfully maintain referential integrity in Cassandra.

The class diagram of the API is presented in Figure 4.1 alongside with the classes that belong to the University keyspace example. The design of the API follows the Entity Relationship (ER) model and the main components are the entities, entity managers, and validation handlers, all of which are described in the next sections. Notice that for the sake of clarity and brevity, the class diagram only contains the relevant methods of the classes, favoring a simpler explanation of the functioning of the API.

[2]The source code is available at `http://code.google.com/p/harsha-api/`.

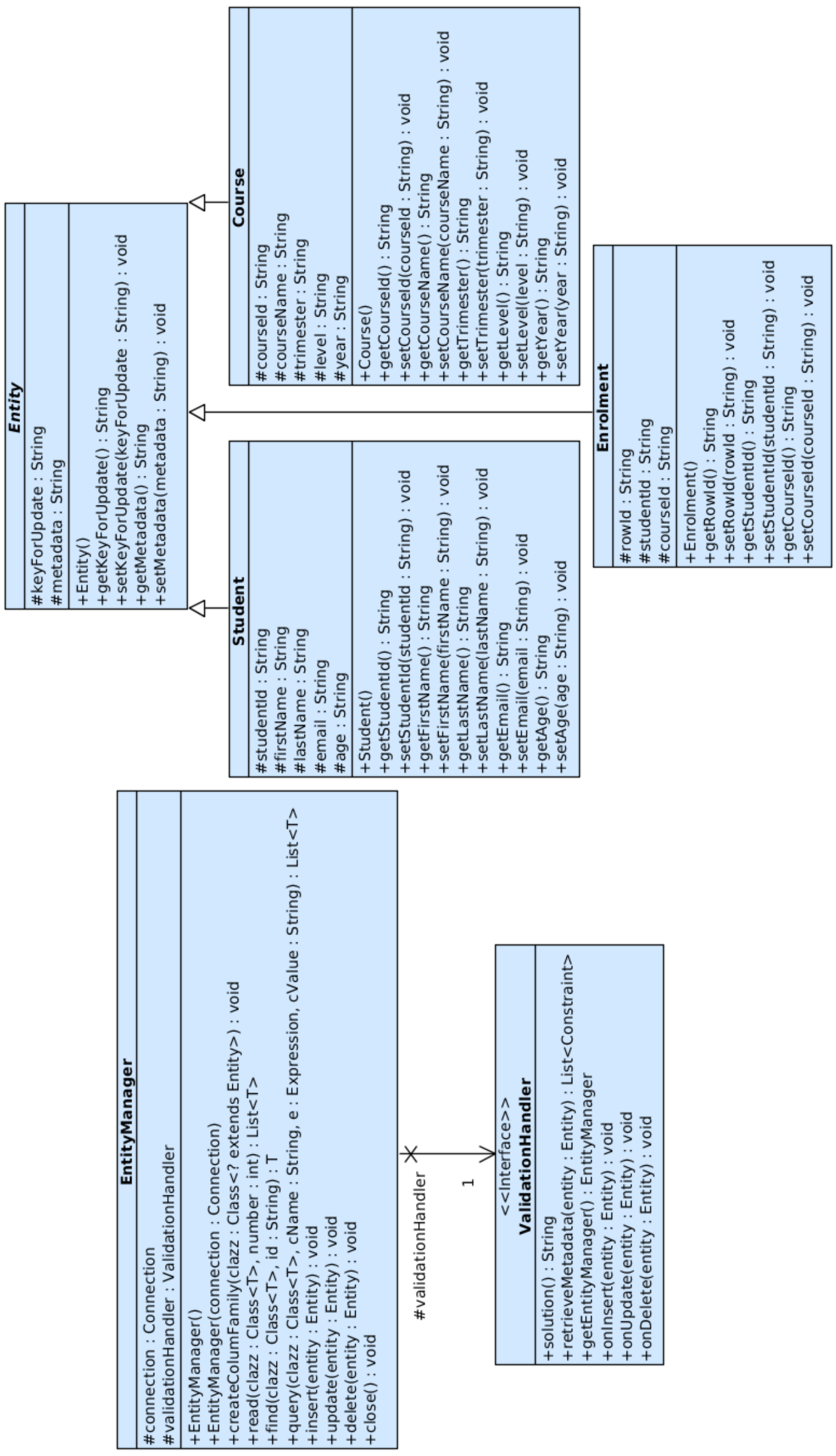

Entity
#keyForUpdate : String
#metadata : String
+Entity()
+getKeyForUpdate() : String
+setKeyForUpdate(keyForUpdate : String) : void
+getMetadata() : String
+setMetadata(metadata : String) : void
Course
#courseId : String
#courseName : String
#trimester : String
#level : String
#year : String
+Course()
+getCourseId() : String
+setCourseId(courseId : String) : void
+getCourseName() : String
+setCourseName(courseName : String) : void
+getTrimester() : String
+setTrimester(trimester : String) : void
+getLevel() : String
+setLevel(level : String) : void
+getYear() : String
+setYear(year : String) : void
Student
#studentId : String
#firstName : String
#lastName : String
#email : String
#age : String
+Student()
+getStudentId() : String
+setStudentId(studentId : String) : void
+getFirstName() : String
+setFirstName(firstName : String) : void
+getLastName() : String
+setLastName(lastName : String) : void
+getEmail() : String
+setEmail(email : String) : void
+getAge() : String
+setAge(age : String) : void
Enrolment
#rowId : String
#studentId : String
#courseId : String
+Enrolment()
+getRowId() : String
+setRowId(rowId : String) : void
+getStudentId() : String
+setStudentId(studentId : String) : void
+getCourseId() : String
+setCourseId(courseId : String) : void
EntityManager
#connection : Connection
#validationHandler : ValidationHandler
+EntityManager()
+EntityManager(connection : Connection)
+createColumFamily(clazz : Class<? extends Entity>) : void
+read(clazz : Class<T>, number : int) : List<T>
+find(clazz : Class<T>, id : String) : T
+query(clazz : Class<T>, cName : String, e : Expression, cValue : String) : List<T>
+insert(entity : Entity) : void
+update(entity : Entity) : void
+delete(entity : Entity) : void
+close() : void
#validationHandler
1
<<Interface>>
ValidationHandler
+solution() : String
+retrieveMetadata(entity : Entity) : List<Constraint>
+getEntityManager() : EntityManager
+onInsert(entity : Entity) : void
+onUpdate(entity : Entity) : void
+onDelete(entity : Entity) : void

Figure 4.1: Class Diagram for the API

4.1.1 Entities

An `Entity` class contains attributes (with respective getters and setters) that map to columns within a specific column family. As such, the contents of a column family can be represented by a list of entity objects. All entities in the API extend from the class `Entity` to aid the API towards their management. Particularly, the attributes that `Entity` contains are `columnFamily` which determines the column family to which the entity maps, and `keyForUpdate` which shall contain the new value in case the primary key of the entity is to be updated.

For example, considering the University keyspace example, `Enrolment` is an entity class that maps to the `Enrolment` column family, thus containing the attributes `RowId`, `CourseId` and `StudentId` which represent its respective columns. As such, an instance of `Enrolment` contains the values of one super column. Likewise, `Student` and `Course` are entity classes and their instances map to super columns in their respective column families. Thus, the CRUD operations that are performed on these entities and are handled by the `EntityManager` class, explained next.

4.1.2 EntityManager

The `EntityManager` class implements all the CRUD operations to be performed on the entities. In order to perform these operations, the `EntityManager` interacts with the respective keyspace the entity belongs. Moreover, it ensures to trigger the referential integrity validation process whenever a CRUD operation requires it. The validation is triggered by invoking the respective methods on the `ValidationHandler` object which is contained within the `EntityManager`.

The `EntityManager`, before performing any operation, requires a connection to the keyspace. This connection is established using a third-party API named Hector [24]. Hector encapsulates the driver-level interface provided by Cassandra (known as Thrift) and simplifies the interaction

with it. Regarding the CRUD operations, Hector provides a `Mutator` class that encapsulates the necessary procedures to create, update, and delete ,
and different classes to query for data (e.g. `SliceQuery`).

Notice that the `EntityManager` is able to generically deal with any entity that derives from the `Entity` class. This can be seen in the class diagram where `T` is a generic type which extends `Entity` and is used across the CRUD methods. The `EntityManager` is able to deal with any entity by using reflection, a Java feature that performs an introspection of a class to retrieve its attributes, methods, annotations, among others. Moreover, the methods retrieved from such a class can be invoked on an object in run-time. Thus, the `EntityManager` uses reflection to invoke the respective getter and setter methods of an entity in order to load it with data from the column family or to write the data into a column family.

Create

The `create` (or `insert`) operation stores an entity in its respective column family. This operation triggers a referential integrity validation whenever an entity is inserted. Such validation is performed by the `onInsert` method of `ValidationHandler`. Finally, if the `ValidationHandler` allows it, the `EntityManager` passes the entity details including row key and column family as parameters to the `addInsertion` method of the Hector `Mutator` object to perform the insertion. This operation is detailed in Algorithm 4.1

Read

The `read` operation retrieves entities from the column family mapped by the entity class. This API provides three methods for retrieving entities: `find`, `query` and `read`. The `find` method retrieves a single entity given the class and the value of its primary key. The `query` method retrieves a list of entities from a column family given the class and a conditional expression $(< \,,\ \leq \,,\ = \,,\ \geq \,,\ >)$ on a column name and and its column value. The `read`

```
Input: Entity e to insert
begin
    invoke onInsert on validationHandler passing entity e;
    if no exception is thrown then
        retrieve the attributes of e;
        foreach attribute do
            invoke addInsertion on Mutator passing attribute;
        invoke execute on Mutator;
end
```

Algorithm 4.1: Insert algorithm in `EntityManager`

method retrieves the list of entities contained in the column family. These methods are detailed in Algorithms 4.2, 4.3 and 4.4, respectively.

Notice that, these operations do not prompt any referential integrity validation since entities are only read and their state is not changed, unlike in the other operations.

```
Input: Class<T extends Entity> clazz, String value
Result: <T extends Entity>
begin
    retrieve attributes of clazz;
    create SliceQuery of clazz where PK = value;
    if no result is found then
        return null;
    else
        create instance of clazz;
        load instance with result;
        return instance;
end
```

Algorithm 4.2: Find algorithm in `EntityManager`

```
Input: Class<T extends Entity> clazz,
       String columnName, Expression expression, String
       columnValue
Result: List of <T extends Entity>
begin
    retrieve attributes of clazz;
    create IndexedSlicesQuery of clazz where
        columnName expression columnValue;
    foreach result found do
        create instance of clazz;
        load instance with result;
        add instance to list;
    return list
end
```

Algorithm 4.3: Query algorithm in `EntityManager`

```
Input: Class<T extends Entity> clazz
Result: List of <T extends Entity>
begin
    retrieve attributes of clazz;
    create RangeSlicesQuery of clazz
    foreach result found do
        create instance of clazz;
        load instance with result;
        add instance to list;
    return list
end
```

Algorithm 4.4: Read algorithm in `EntityManager`

Update

The `update` operation changes the columns of an existing entity. If the changes to be performed are on columns which are not the primary key, then an `insert` operation is performed as it uses the primary key value of the entity to locate and replace the column values. Otherwise, if the primary key value is to be updated, then more actions need to be performed.

Firstly, the entity with new primary key value has to be inserted, then all the children entities have to be located and their foreign keys updated to reflect the new primary key value, and finally the old entity has to be removed.

This operation triggers referential integrity constraints which are validated by the method `onUpdate` of the `ValidationHandler` when the primary key value is to be updated, otherwise, the respective validations are performed when the entity is inserted.

```
Input: Entity e to update
begin
    if primary key value of e is not changed then
        invoke insert on this EntityManager passing e;
        return
    insert entity with keyForUpdate as primary key;
    invoke onUpdate on validationHandler passing entity e;
    if no exception is thrown then
        invoke delete from this EntityManager passing e;
    else
        delete entity with keyForUpdate as primary key;
end
```

Algorithm 4.5: Update algorithm in `EntityManager`

Delete

The `Delete` operation removes an entity from its respective column family. As mentioned before, primary key values will never cease to exist in Cassandra due to the tombstone delete, hence, this operation empties the values of the columns represented within the entity to be deleted. Even when primary key values exist within the column families for deleted entities, these are ignored on `read` operations.

The `Delete` operation triggers referential integrity validations every time an entity is deleted. This validation is performed by the `onDelete` method of the `ValidationHandler`. Finally, if the `ValidationHandler` allows it, the `EntityManager` passes the necessary information to the `delete` method of the Hector `Mutator` object. This operation is detailed in Algorithm 4.6.

```
Input: Entity e to delete
begin
    invoke onDelete on validationHandler passing entity e;
    if no exception is thrown then
        retrieve attributes of e;
        invoke delete on Mutator passing primary key and column family
        of e;
end
```

Algorithm 4.6: Delete algorithm in `EntityManager`

4.1.3 ValidationHandler

The `ValidationHandler` is used by the `EntityManager` every time an operation triggers referential integrity validations on any entity. It contains the logic that checks if an entity has dependencies, verifies whether the `insert`, `update` or `delete` operations performed on an entity violate referential integrity constraints, and it also applies referential integrity rules when operations are cascaded or updates nor deletes are allowed.

The `ValidationHandler` is just an interface that has to be implemented to deal with each solution. However, since the validations are the same across solutions, it is sufficient to provide a generic class that implements the `ValidationHandler` for CRUD operations. Thus, it is left for each solution to extend the generic `ValidationHandler` and implement the `retrieveMetadata` method accordingly.

Validation: `onInsert`

This validation is triggered every time the `EntityManager` is asked to insert an entity. It occurs before the insertion of the actual data as the validation has to check whether the entity has foreign keys to other column families. This is determined by retrieving the constraints relevant to the entity, and if a constraint indicates that the entity has foreign keys, this validation must ensure that they match the primary keys of the respective column families they reference. This validation is detailed in Algorithm 4.7.

```
Input: Entity e to be inserted
begin
    retrieve the metadata of e;
    foreach constraint of type R ∈ metadata do
        retrieve referenced constraint RConstraintName;
        determine the entity class that maps to the parent column family;
        use EntityManager to find parent entity;
        if parent entity does not exist then
            throw Exception;
end
```

Algorithm 4.7: Validation `onInsert`

For example, when an `Enrolment` entity is to be inserted, the `ValidationHandler` identifies the parent column families by looking up at the FK constraints of the entity. In this case, there is only `CONST400` which identifies `Student` as a parent column family. Hence, the `ValidationHandler` makes sure there is a `Student` entity whose primary key value matches the foreign key value of the `Enrolment` entity to be inserted. If such a parent entity exists, then the `EntityManager` is allowed to `insert` the entity.

Validation: `onUpdate`

This validation is triggered every time the `EntityManager` is asked to update an entity which primary key value has changed. Recall that if the changes of the entity to be updated do not involve the primary key, then an `insert` takes place instead and hence the validation is performed by `onInsert`. Otherwise, if the primary key value of the entity is changed, the new entity is inserted, the foreign keys of the dependencies are updated to the new primary key value, and the old entity is deleted. Clearly, the referential integrity rules apply in this validation. These are assumed to be contained within the `DeleteRule`[3] field of the constraints relevant to the children entities. Thus, the validation finds all the child dependencies of the entity and, if the `DeleteRule` is `CASCADE`, updates their corresponding foreign keys to the new primary key value. Otherwise, if the `DeleteRule` is `NODELETE`, the update is only allowed if there are no child dependencies. Notice that this operation assumes the entity with new primary key value has already been inserted. This validation is detailed in Algorithm 4.8.

```
Input: Entity e to be updated
begin
    retrieve the metadata of e;
    foreach constraint of type F ∈ metadata do
        retrieve referenced constraint RConstraintName;
        if DeleteRule is CASCADE then
            determine the entity class that maps to the child column family;
            use EntityManager to query for the children of e;
            add children to list;
        else if DeleteRule is NODELETE then
            determine the entity class that maps to the child column family;
            use EntityManager to query for the children of e;
            if e has children then
                throw Exception;
    foreach child ∈ list do
        use EntityManager to update child;
end
```

Algorithm 4.8: Validation `onUpdate`

For example, in the University keyspace, if the primary key of a `Course` entity is to be updated, the `ValidationHandler` locates the FK constraints which reference `Course`. In this case, only `CONST600` references `Course`, which follows up to `CONST500` from which the child entity can be retrieved. Since the `DeleteRule` is `NODELETE`, it is checked if there are

[3]Notice that for the sake of simplicity, this rule is also used for update operations

any `Enrolment` entities which reference the course to be updated. If there are child entities, an exception is thrown and the update of the `Course` entity is not allowed. Otherwise, if no `Enrolment` entities refer to such a course, the `EntityManager` is allowed to update its primary key value.

Validation: `onDelete`

This validation is triggered every time the `EntityManager` is asked to delete an entity. This validation ensures that referential integrity rules are applied when the entity is referenced by other entities. That is, the entity to be deleted is parent of other entities. The referential integrity rule to apply can be `CASCADE` or `NODELETE`. In the former case, child dependencies on the entity are deleted, while deletion in the latter case is only allowed when the entity has no child dependencies. This validation is detailed in Algorithm 4.9.

For example, in the University keyspace, if a `Student` entity is requested to be deleted, the `ValidationHandler` locates the FK constraints which reference `Student`. In this case, only `CONST700` references `Student`, which follows up to `CONST400` from which the child entity can be retrieved. Since the `DeleteRule` is `Cascade`, the child entities are deleted from `Enrolment` and no exception is thrown, hence allowing the `EntityManager` to delete the entity from its `Student` column family.

```
Input: Entity e to be deleted
begin
    retrieve the metadata of e;
    foreach constraint of type F ∈ metadata do
        retrieve referenced constraint RConstraintName;
        if DeleteRule is CASCADE then
            determine the entity class that maps to the child column family;
            use EntityManager to query for the children of e;
            add children to list;
        else if DeleteRule is NODELETE then
            determine the entity class that maps to the child column family;
            use EntityManager to query for the children of e;
            if e has children then
                throw Exception;
    foreach child ∈ list do
        use EntityManager to delete child;
end
```

Algorithm 4.9: Validation `onDelete`

4.2 Metadata Retrieval Approaches in Solutions

In every solution, every time referential integrity validations are triggered by operations performed on any entity, the `ValidationHandler` requires access to its metadata. Even when metadata storage is different in each solution, all of them adopt one of the following two methods for retrieving and processing the metadata. One method handles metadata as an entity and the other handles metadata as text. These approaches and their implementation are explained next.

4.2.1 Metadata as an Entity

Solutions 3 and 4 store metadata in separate column families, either in the same cluster or in a different one (respectively). As such, in the API, the metadata column family is mapped by the entity class `Metadata` which contains the necessary information for each constraint. These constraints are inserted by the `EntityManager` the same way as other entities are inserted, only without performing referential integrity validations. Notice that the constraints required by an application must be explicitly provided to the API by inserting them upon initialization of the keyspace .

The `Metadata` entity class stores the various parts of a constraint as its attributes and provides their respective getter and setter methods. Since all the constraints in a keyspace are stored in the `Metadata` column family, a single metadata entity refers to a single constraint. Thus, a list of `Metadata` entities contains all the relevant constraints for an entity. The class diagram of the `Metadata` entity class is shown in Figure 4.2.

The `ValidationHandler` retrieves the list of `Metadata` entities relevant to the entity upon validation. In this case, it does so by using the `EntityManager` to read the constraints from `Metadata`. The `ValidationHandler` iterates through the list of `Metadata` entities and uses the respective getter methods in order to retrieve the different attributes of the metadata to complete the validation. Notice that, in Solution 4, the list of metadata is maintained in cache in order to re-use it for future validations and to avoid additional connections to the metadata cluster each time validation needs to be performed.

4.2.2 Metadata as Text

Solutions 1 and 2 store metadata as a string of text. Solution 1 stores the metadata within each entity in its `Metadata` column, and Solution 2 stores the metadata in the top row of the respective column families. Notice that, Solution 2 performs an additional search to locate the top row

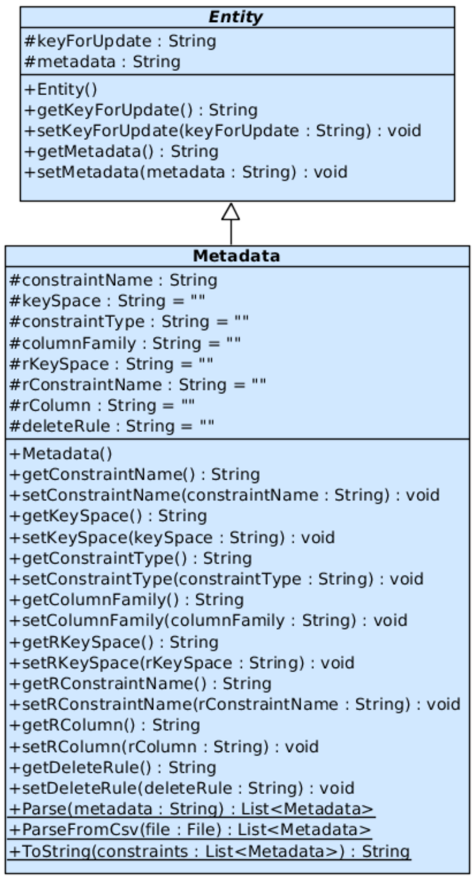

Entity
#keyForUpdate : String
#metadata : String
+Entity()
+getKeyForUpdate() : String
+setKeyForUpdate(keyForUpdate : String) : void
+getMetadata() : String
+setMetadata(metadata : String) : void
Metadata
#constraintName : String
#keySpace : String = ""
#constraintType : String = ""
#columnFamily : String = ""
#rKeySpace : String = ""
#rConstraintName : String = ""
#rColumn : String = ""
#deleteRule : String = ""
+Metadata()
+getConstraintName() : String
+setConstraintName(constraintName : String) : void
+getKeySpace() : String
+setKeySpace(keySpace : String) : void
+getConstraintType() : String
+setConstraintType(constraintType : String) : void
+getColumnFamily() : String
+setColumnFamily(columnFamily : String) : void
+getRKeySpace() : String
+setRKeySpace(rKeySpace : String) : void
+getRConstraintName() : String
+setRConstraintName(rConstraintName : String) : void
+getRColumn() : String
+setRColumn(rColumn : String) : void
+getDeleteRule() : String
+setDeleteRule(deleteRule : String) : void
+Parse(metadata : String) : List<Metadata>
+ParseFromCsv(file : File) : List<Metadata>
+ToString(constraints : List<Metadata>) : String

Figure 4.2: Metadata Entity Class

(`RowId='-1'`) where the metadata is stored, and then loads it within the entity.

The string of metadata within each entity contains all the constraints separated using special characters as explained in Section 3.2. These special characters serve as delimiters to parse and extract the information about the relevant constraints. This information is then loaded into the attributes of an instance of the `Metadata` entity class and, from then on, metadata is handled as an entity as explained in the previous section. Notice that the parsing algorithms for metadata as string are provided as static members of `Metadata` class.

4.3 Summary

This chapter presented the implementation details of the experimental API and the way the solutions retrieve and process metadata. The experimental API is composed of a class `Entity` instances of which map to single super columns in column families, a class `EntityManager` instances of which perform all the CRUD operations on the entities, and a `ValidationHandler` interface for which a generic implementation is provided and is in charge of checking and ensuring that referential integrity is maintained in the keyspace at all times.

In order for the `ValidationHandler` to ensure referential integrity, it must retrieve the metadata associated to the entity upon validation. The metadata retrieval is different in each solution, yet its handling is either from a string of text or as an entity directly. On the one hand, Solution 1 and 2 store the metadata as text, the former within the entities while the latter in the top row of the column family. Hence, in both solutions, the metadata needs to be parsed and the necessary information of the constraints needs to be extracted to then load these it into a list of `Metadata` entities such that its handling is simplified. On the other hand, Solution 3 and 4 store the metadata in column families, the former within the same cluster while the latter uses a dedicated metadata cluster. Since metadata has its own column family, each constraint is naturally mapped into a `Metadata` entity. Thus, metadata can be retrieved by using the `EntityManager` and handling the `Metadata` as any other entity. Notice that all these classes, while providing the fundamental operations on generic entities, they can be derived to suit custom application requirements.

The next chapter presents the experimental design used to evaluate the performance of each solution implemented within the API.

Chapter 5

Experimental Design

The implementation of the four solutions introduces referential integrity constraints and validations in Cassandra, which are not provided by this Database Management System (DBMS) at the moment of writing. In order to evaluate the performance of these four solutions, experiments are conducted by using the implemented Application Programming Interface (API) described in Section 4.1. The goal of the experiments is to determine how each solution affects the performance of Cassandra in Create, Read, Update and Delete (CRUD) operations, specifically in those where referential integrity validations are triggered (namely Create, Update and Delete).

The experimentation is performed on the example application presented across chapters: the University keyspace. In such an application, different constraints are added and allow to test the performance under such different application requirements. The performance of the solutions provided for ensuring referential integrity is measured based on response time and throughput.

This chapter is structured as follows. Section 5.1 describes the example application used for the experiments. Section 5.2 provides the details of the nodes used in the Cassandra cluster. Section 5.3 describes the experimental setup to evaluate the performance of the solutions. Section 5.4 presents the performance indicators considered for measuring the results from the experiments. Finally, Section 5.5 presents a summary of the chapter.

5.1 Example application

The API designed and implemented in the previous chapters is validated and tested by performing CRUD operations on an example application specifically designed for this purpose. This application has been referred to as the `University` keyspace in previous chapters, and it contains different constraints in order to assess the performance of the API and the solutions on each of them. This application stores the details of students and courses along with the enrolment details of the students. The class diagram for the University keyspace is shown in Figure 5.1, and each entity is saved into its respective column family in a Cassandra cluster.

- `Student` stores the following attributes of students: `StudentId` (primary key), `FirstName`, `LastName`, `Email` and `Age`.
- `Course` stores the following attributes of courses: `CourseId` (primary key), `CourseName`, `Trimester`, `Level` and `Year`.
- `Enrolment` stores the relationship between students and courses, that is, it stores the courses each student is enrolled into. The attributes for `Enrolment` are `RowId` (primary key), `StudentId` and `CourseId`, where `StudentId` and `CourseId` are foreign keys.

The list of constraints created for the University keyspace can be seen in Table 5.1. Constraints `CONST100`, `CONST200` and `CONST300` are the respective Primary Key (PK) constraints for the three column families. `CONST400` and `CONST500` are the Foreign Key (FK) constraints of `Enrolment` which specify the respective parent column family for its foreign keys. `CONST600` and `CONST700` show the child column family for `Course` and `Student` respectively.

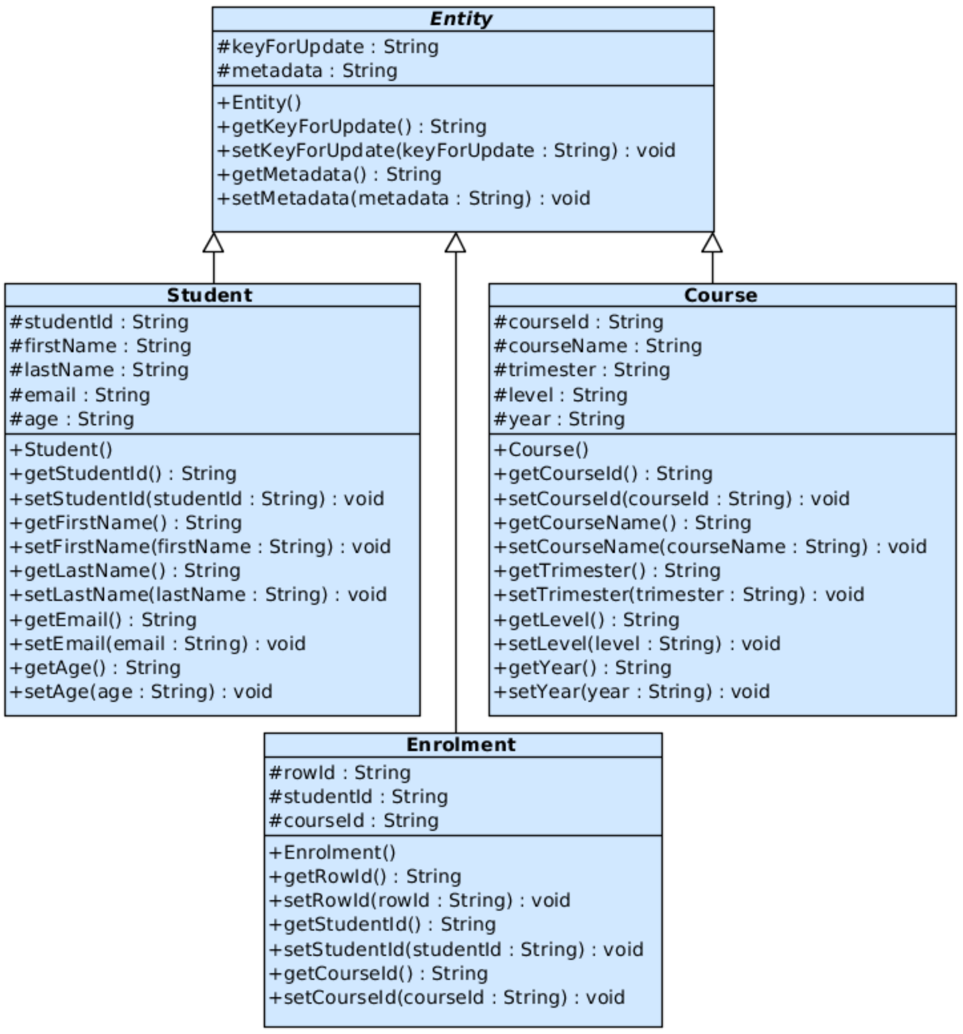


Figure 5.1: Class diagram for University

5.2 Cassandra cluster

Cassandra is deployed in an homogeneous cluster conformed by 10 nodes. That is, all 10 nodes have the same characteristics in software and hardware. These nodes emulate a cloud environment in which each node saves the data on the local disks of the machines. Notice that, for Solution 4, an

Table 5.1: Metadata

Constraint Name	Keyspace	Constraint Type	Column Family	RKeyspace	RConstraint Name	RColumn	DeleteRule
CONST100	University	P	Student	University		StudentId	
CONST200	University	P	Course	University		CourseId	
CONST300	University	P	Enrolment	University		RowId	
CONST400	University	R	Enrolment	University	CONST100	StudentId	CASCADE
CONST500	University	R	Enrolment	University	CONST200	CourseId	NODELETE
CONST600	University	F	Course	University	CONST500	CourseId	NODELETE
CONST700	University	F	Student	University	CONST400	StudentId	CASCADE

additional node is used to emulate an external cluster dedicated to provide metadata of the entities upon request. The following are the characteristics of these nodes.

- Hardware:
  - 2.8 GHz Intel(R) Core(TM) 2 Quad Processor
  - 16 GB main memory (RAM)
  - 1000 GB SATA hard disk
  - 1000 Mbit/s, 802.11n Networking options
- Software:
  - Operating system: Linux 3.2.4-1-ARCH i686 (64-bit)
  - Java JDK 1.6.0_31 (Java 6)
  - Cassandra version 0.8.4
  - Hector version 0.8.0-2

The nodes used in the cluster are part of the Engineering and Computer Science grid system of Victoria University of Wellington. Notice that, such a cluster is not a controlled environment and it is not possible to use it as a dedicated cluster as it can be used for other grid jobs or by students. Nonetheless, the experiments can be performed over night during weekends when the external usage of these nodes is minimal.

Some values in the configuration files on each node are changed before starting the cluster of nodes. For every node, the `listen_address` and `rpc_address` are set to the hostname. The nodes are added to the cluster in a sequential order. One of the nodes is chosen as the first node and is made a host node (a. k. a. seed node). This node becomes the contact point for the following node to join the cluster. The hosts for any given node are specified in its configuration file under the `seeds` option. For the first node, this option is set to its loopback address "`127.0.0.1`" since no other nodes have joined the cluster yet. For nodes that are not seed nodes, this option contains the hostnames that it can contact to learn about the cluster. In the experiments, except for the first node, the remaining nodes have two neighboring hosts as seeds.

The seed node has its `auto_bootstrap` option set to `true` to allow other nodes to migrate data from it when data is partitioned or when other nodes join the cluster. For nodes that are not seed nodes, this option is set to `false`. This is because all the nodes are started prior to the experiments and do not have data to partition yet.

All the remaining settings in the Cassandra configuration file are set to the default values for all the nodes. The directories for saving the data, commit logs and saved caches are saved on the local disk of each node in its temporary folder (`/local/tmp`).

5.3 Experimental setup

The experimentation consists of performing CRUD operations upon artificial data created for the University example application. Specifically, the operations of interest are `insert`, `update` and `delete` as these are the ones that trigger referential integrity validations. Notice that, since the experiments are not performed in a controlled environment, all the operations are repeated 100 times such that the effect of external factors (e.g. network latency, parallel processes running in nodes, etc.) is minimized.

The artificial data upon which operations are performed is made up of 500 students, 500 courses, and 5000 enrolments. These numbers were chosen such that the experiments could be performed in a reasonable amount of time. The format of the artificial data is:

- `Student` has a unit-increasing `StudentId` which is merged into the fields `FirstName` and `LastName` as "First Name (StudentId)" and "Last Name (StudentId)". `Email` is composed in a similar way as "First.Last@email.(StudentId).com" and `Age` is a random number between 18 and 60.

- `Course` has a unit-increasing `CourseId` which is appended to the prefix "COMP". It also has a composed `CourseName` as "Engineering (CourseId)". `Trimester`, `Level` and `Year` are randomly generated numbers.

- `Enrolment` contains a unit-increasing `RowId` and the respective foreign keys of student and course, which are `StudentId` and `CourseId`.

The order of the operations to be performed on the data in each run is as follows. The `insert` operation inserts all the entities for `Student`, `Course` and `Enrolment`. The `update` operation performs changes on the primary keys of `Student` and `Course` entities, and on the foreign keys of `Enrolment` (the one relative to courses, specifically). Finally, the `delete` operation removes all the `Student`, `Course` and `Enrolment` entities. Notice that the primary keys in every column family are different in each run (create, update, delete), in order to avoid introducing biases to the results as product of the tombstone delete paradigm that Cassandra utilizes. That is, since Cassandra does not completely remove the primary keys of the inserted entities (tombstone delete), reinsertion using the same primary key might yield faster times as the key already exists. After each run, all the column families (`Student`, `Course`, and `Enrolment`) are emptied and ready for the next run. The details of the `insert`, `update` and `delete` operations are explained further in the following sections.

5.3.1 Insert

The `insert` operation inserts all the `Student`, `Course` and `Enrolment` entities in that precise order due to the nature of the referential integrity constraints presented in Table 5.1. In the `Student` and `Course` column families, the `insert` operation on these entities do not require referential integrity constraints to be satisfied as these entities do not contain foreign keys. Contrarily, `insert` on `Enrolment` triggers foreign key validation checks on both `Student` and `Course` column families.

5.3.2 Update

The `update` operation is performed after the creation of all entities. First, an attempt is made to update the primary key of each `Course` entity. This operation triggers referential integrity validations that result in exceptions thrown as the `DeleteRule` [1] for all `Course` entities is `NoDelete` and enrolments referencing the courses are still present in `Enrolment`. Hence, the times recorded for updating the `Course` column family represent the time required to identify a constraint violation and throw the respective exceptions.

[1]Notice that for the sake of simplicity, this rule is also used for update operations.

Next, the `Enrolment` column family is updated. In this case, the `CourseId` for each `Enrolment` entity is changed to a different existing value, ensuring that the distribution of `Student` and `Course` entities remains the same. The update on the `Enrolment` column family triggers referential integrity validation checks to ensure that the course to which every `Enrolment` entity is being updated actually exists in `Course` column family.

Finally, the primary key for each `Student` entity is updated to a new integer value that has never existed in the column family. Thus, given the `DeleteRule` for `Student` (i.e. `Cascade`), this operation triggers a cascaded update on the `Enrolment` column family by respectively updating the student foreign key (`StudentId`) in all its existing `Enrolment` entities.

5.3.3 Delete

The deletion of entities occurs first on the `Enrolment` column family, where all of its records are deleted without requiring referential integrity checks as this is a child entity. The times are recorded for each `delete` operation and then all of the entities are reinserted with the same primary keys in order to assess the cascaded `delete` of `Student` entities next.

Secondly, all the `Student` entities are deleted from the `Student` column family. Hence, given the `Cascade DeleteRule` of these entities, the `ValidationHandler` ensures to delete first all of the child entities before deleting a `Student` entity. Thus, the times recorded for this operation also include the time required for performing a cascaded `delete` on the student dependencies in `Enrolment`. Notice that the dependencies exist at this point as they will have been reinserted into `Enrolment` in the previous step.

Finally, all the `Course` entities are deleted. Despite the courses having a `NoDelete` rule, notice that at this point the `Enrolment` column family is empty, so courses can be deleted as there are no child dependencies.

Thus, the times recorded for this operation measure referential integrity validation as well as the `delete` operation of the `Course` entity. After this final operation, all column families are emptied but all the primary keys still exist due to Cassandra's tombstone delete. However, the whole keyspace is ready for the next batch of operations as the primary keys of all column families will be different.

5.4 Performance Indicators

Response time and throughput are the indicators used to gauge the performance of the four solutions under the implemented API and the respective referential integrity constraints specified by the example application. Response time refers to the time a DBMS takes to process an operation and produce results to the end user [7]. Throughput refers to the number of operations that can be processed by the DBMS in a unit of time.

In the experiments, the response time is computed by dividing the total execution time of an operation for a set of entities by the number of entities. In other words, the response time measures the average amount of time required to perform a single operation on one entity. On the other hand, the throughput is the inverse of the response time and is computed accordingly by dividing the number of entities by the total execution time of an operation for a set of entities. In other words, the throughput measures the number of operations that are performed in a unit of time. The following equations define the response time r and throughput t,

$$r = \frac{1}{n}\sum_{i=1}^{n} o_i \qquad\qquad t = n/\sum_{i=1}^{n} o_i$$

where o_i is the time for an operation over entity i, and n is the number of entities.

Notice that external variables such as network latency, simultaneous processes in the operating systems of each node, and other variables are not considered for the analysis of results. Even when they are present, it is expected that results will not be significantly biased by them. Nonetheless, the experiments will be performed at night time over weekends as this is the time when the cluster is least used, thus reducing the presence of such variables and hence their impact on the results.

5.5 Summary

This chapter presented the experimental design to evaluate the performance of each solution and the API itself on the `University` application, which is used as an example in previous chapters as well. This application contains different constraints that makes it useful for assessing the performance of the API as well as that of the solutions since the constraints will trigger different referential integrity validations in order to maintain integrity within the keyspace. The validations are triggered on different CRUD operations performed upon artificial data intentionally created for the application. All the operations are performed several times such that the effect of external factors (e.g. network latency) is mitigated. The response time of each operation is recorded and used as a performance indicator together with throughput, thus providing guidelines to help assess the trade-offs between the different solutions proposed.

The next chapter presents the results obtained from the experimentation as well as their discussion.

Chapter 6

Results and Discussions

The performance of the solutions is measured in terms of response time and throughput while validating referential integrity in the experiments. Response e time and throughput are common Database Management System (DBMS) performance indicators. The response time indicates the time taken for an operation to be completed while throughput measures the number of operations that can be completed in a unit of time. The performance of the operations when referential integrity validations are not enforced is also measured and considered as the baseline with which the solutions are compared. Such a comparison determines the difference in performance when such additional validations are enforced using the Application Programming Interface (API) and provides a guideline to asses the performance impact of each solution.

The results from the experiments are analysed and discussed in this chapter. Section 6.1 presents an overview of the performance of the four solutions. Section 6.2 presents the performance without referential integrity validations. Sections 6.3, 6.4 and 6.5 compare the results of all the solutions for the `insert`, `update` and `delete` operations (respectively). Section 6.6 presents an overall comparison of the operations. Finally, Section 6.7 presents a summary of this chapter.

6.1 Overview of Results

The experiments were performed to evaluate the response time and throughput of the solutions in order to determine the impact of the metadata storage and referential integrity validations on the performance of Cassandra. Notice that the performance is measured and analysed for only the operations that trigger referential integrity validations. That is, the response time and throughput are measured for the `insert`, `update` and `delete` operations across the solutions.

The response time measures the average amount of time required to perform a single operation on one entity. Conversely, the throughput is the inverse of the response time and measures the number of operations that are performed in a unit of time. Tables 6.1 and 6.3 present the mean and standard deviation of the average response time and the throughput for all the solutions. Notice that the solution with the lowest response time and highest throughput has a better performance than rest, while the solution with the highest response time and lowest throughput has the worst performance. In other words, the better performing solution is the one that executes operations using the least amount of time and hence completes more operations in a unit of time.

As seen in these tables, Solution 4 performs the best amongst all since it has better response times and throughput for all the operations on every entity. Notice that Solution 4 performs similar to to baseline only when inserting `Course` and `Student` and when deleting `Enrolment` entities as in these cases, there are no referential integrity constraints to be satisfied. Conversely, Solution 3 performs the worst as it has the worse response times and throughput for all the operations. Regarding Solutions 1 and 2, they perform similarly, although, Solution 1 is faster with slightly smaller response times and higher throughput that Solution 2.

This can be further seen in the ratio of the response time and that of the throughput presented in Tables 6.2 and 6.4. The former shows the ratio of

Table 6.1: Response time in milliseconds per entity

		Baseline	Solution1	Solution2	Solution3	Solution4
	s	0.366 (0.08)	0.568 (0.03)	0.820 (0.09)	2.108 (0.05)	**0.364 (0.02)**
insert	**c**	0.352 (0.05)	0.547 (0.04)	0.803 (0.05)	2.092 (0.06)	**0.351 (0.01)**
	e	0.305 (0.01)	1.239 (0.04)	1.405 (0.02)	3.484 (0.05)	**0.936 (0.01)**
	s	0.730 (0.16)	19.144 (0.39)	20.840 (0.43)	46.394 (0.73)	**14.997 (0.37)**
update	**c**	0.759 (0.06)	5.810 (0.50)	5.991 (0.27)	10.419 (0.30)	**4.751 (0.28)**
	e	0.404 (0.03)	1.353 (0.03)	1.500 (0.02)	3.579 (0.05)	**1.031 (0.01)**
	s	0.314 (0.03)	7.425 (0.44)	10.533 (0.41)	26.023 (0.55)	**5.638 (0.37)**
delete	**c**	0.287 (0.05)	2.037 (0.08)	2.367 (0.09)	3.958 (0.12)	**1.964 (0.09)**
	e	0.290 (0.03)	0.410 (0.02)	0.744 (0.02)	2.132 (0.04)	**0.299 (0.02)**

Table 6.2: Response time ratio with respect to Baseline

		Solution1	Solution2	Solution3	Solution4
	s	1.55	2.24	5.76	**0.99**
insert	**c**	1.56	2.28	5.95	**1.00**
	e	4.06	4.61	11.42	**3.07**
	s	26.21	28.53	63.51	**20.53**
update	**c**	7.66	7.90	13.74	**6.26**
	e	3.35	3.71	8.86	**2.55**
	s	23.61	33.50	82.76	**17.93**
delete	**c**	7.09	8.24	13.78	**6.84**
	e	1.41	2.57	7.36	**1.03**

Lower values mean faster response

the response time of each solution when compared to that of the baseline, and it indicates the factor by which a solution is slower than the baseline; while the latter shows the ratio of the throughput with respect to that of the baseline. The differences in the performance of the solutions is caused by the ways these store and handle metadata. Recall that Solutions 1 and 2 store metadata along with the actual data, where Solution 1 stores it in every super column and Solution 2 stores it as the top super column of a column family. On the other hand, Solutions 3 and 4 store metadata separately from the actual data in a `Metadata` column family, but such a

Table 6.3: Throughput in entities per second

		Baseline	**Solution1**	**Solution2**	**Solution3**	**Solution4**
insert	**s**	2790 (291)	1764 (85)	1228 (82)	475 (12)	**2755 (125)**
	c	2880 (264)	1837 (112)	1250 (69)	478 (13)	**2856 (96)**
	e	3282 (116)	807 (22)	712 (11)	287 (4)	**1069 (15)**
update	**s**	1394 (121)	52 (1)	48 (1)	22 (0)	**67 (2)**
	c	1325 (97)	173 (14)	167 (7)	96 (3)	**211 (12)**
	e	2483 (119)	739 (15)	667 (11)	279 (4)	**970 (12)**
delete	**s**	3198 (205)	135 (8)	95 (4)	38 (1)	**178 (11)**
	c	3574 (567)	492 (19)	423 (15)	253 (7)	**510 (21)**
	e	3470 (206)	2443 (95)	1346 (44)	469 (9)	**3351 (167)**

Table 6.4: Throughput ratio with respect to Baseline

		Solution1	**Solution2**	**Solution3**	**Solution4**
insert	**s**	0.63	0.44	0.17	**0.99**
	c	0.64	0.43	0.17	**0.99**
	e	0.25	0.22	0.09	**0.33**
update	**s**	0.04	0.03	0.02	**0.05**
	c	0.13	0.13	0.07	**0.16**
	e	0.30	0.27	0.11	**0.39**
delete	**s**	0.04	0.03	0.01	**0.06**
	c	0.14	0.12	0.07	**0.14**
	e	0.70	0.39	0.14	**0.97**

Higher values mean more throughput

column family is in a separate cluster in Solution 4.

From these results, it can be seen that Solution 4 is faster than the other solutions when performing the validations since it caches the list of constraints and avoids connecting to the external cluster to access the `Metadata` column family each time operations are invoked on entities. Therefore, to locate the relevant Foreign Key (FK) and Primary Key (PK) constraints of an entity, the constraints stored in the cache memory are re-used. Performance is improved significantly just by caching the `Metadata` column family as it reduces the number of accesses to the column family.

On the other hand, Solution 3 is the slowest because it accesses the metadata from `Metadata` column family every time it is required. That is, for each operation on an entity, the constraints relative to the entity are retrieved from `Metadata` and then, `Metadata` is accessed again to retrieve the referencing constraints. Thus, in order to complete each validation, `Metadata` is accessed more than once. Unlike Solution 4, metadata is not cached for re-use thus costing multiple access to the `Metadata` column family.

Meanwhile, Solutions 1 and 2 have approximately similar response times as both the solutions store the whole list of constraints with the actual data and requires no additional accesses to retrieve the relevant constraints of an entity. Note that, Solution 1 performs slightly better than Solution 2 because the former has the constraints stored within each entity while the latter requires an additional search operation to identify the top row of a column family to locate the constraints. Both solutions are faster than Solution 3 mainly because these have the whole list of constraints along with the actual data. However, they are slower than Solution 4 as these have to access the keyspace to retrieve the constraints from each entity and do not use a cache.

The standard deviation of each operation is presented within parentheses in Tables 6.1 and 6.3. The standard deviation measures the dispersion of the response time and throughput of an operation from the mean. Note that in the experiments, an operation is executed over a set of entities 100 times, which means that one run of the experiment produces 100 values for an operation. Thus, the standard deviation measures the dispersion of the 100 values for every operation with respect to the mean. A low standard deviation means that the values of the response time and throughput of an operation are concentrated around the mean values. Conversely, as the standard deviation increases the response time and throughput values of the operations are more spread.

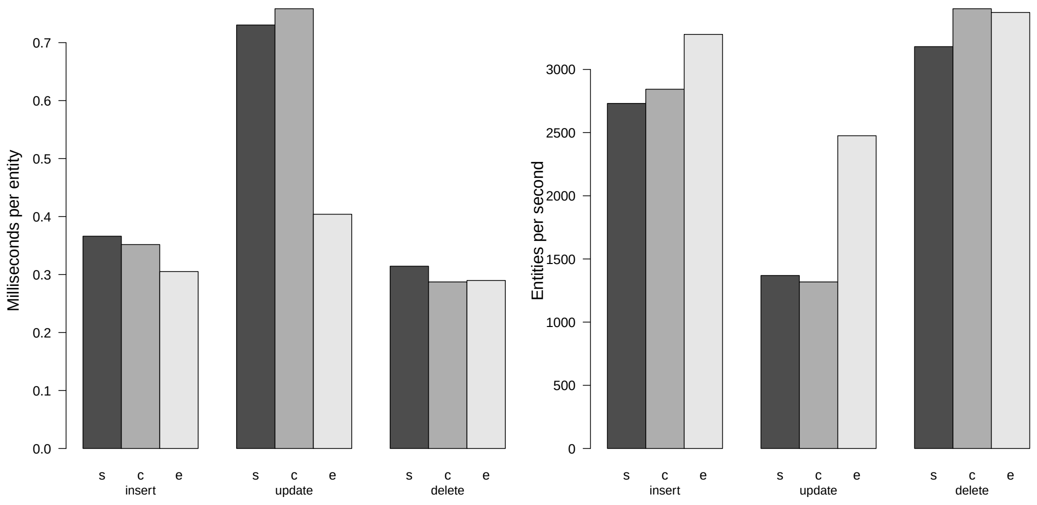
Note that the experiments are run 100 times as external factors such as network latency affect the performance of the operations and the solutions, and thus these 100 values are expected to be different. As can be seen, the network latency is generally a factor that affects the performance of the baseline and the solutions as a network connection is always required to perform operations. This can be deduced by the higher standard deviation present in the results of the solutions which require more accesses to the keyspace. For instance, Solution 4 generally has a low standard deviation as metadata is retrieved once from the keyspace and then it is cached, thus requiring no additional information from the cluster; whereas Solution 3 generally has a high standard deviation as it accesses `Metadata` multiple times in every operation. Solution 2 retrieves all the metadata in one additional access to the keyspace, so its standard deviation is generally lower than Solution 3 and mostly higher than that in Solution 1 which retrieves metadata alongside with the entities.

The performance of the solutions in each operation performed on the entities is discussed in detail in the following sections. Notice that, in all the tables and figures, the entities `Student`, `Course` and `Enrolment` are referred to as '`s`', '`c`' and '`e`' for the sake of brevity.

6.2 Baseline Experiment

This experiment was designed to measure the performance of the Create, Read, Update and Delete (CRUD) operations when no referential integrity validations are triggered. That is, a) no checks are performed to ensure that parent entities exist before inserting or updating child entities; and b) no checks are performed to ensure that child entities exist before deleting parent entities or updating their primary key values. Thus, this experiment is taken as a baseline to assess the impact in performance of the CRUD operations when referential integrity validations are incorporated and the list of constraints on the entities is stored in different locations and handled according to each of the solutions.

The results from this experiment can be seen in Figure 6.1. Specifically, Figure 6.1(a) presents the average response time of the operations on a single entity in the three column families. Similarly, Figure 6.1(b) presents the throughput of such operations. These results show that the performance of the `insert` operation is rather similar between the entities, like the `delete` operation. On the `update` operation, the performance for updating `Student` and `Course` is rather similar, but drastically better when it comes to updating `Enrolment`.



(a) Response time (b) Throughput

Figure 6.1: Performance of Baseline

The performance of the `insert` operation is expected to be similar across solutions since no referential integrity validations are performed. However, the figure shows a slightly better average performance when inserting in `Enrolment`. This subtle difference might be due to the smaller number of columns as well as the smaller size of the contents that `Enrolment` entities have when compared to `Course` and even more when compared to `Student`. Also, external factors such as network latency are expected to affect the performance slightly.

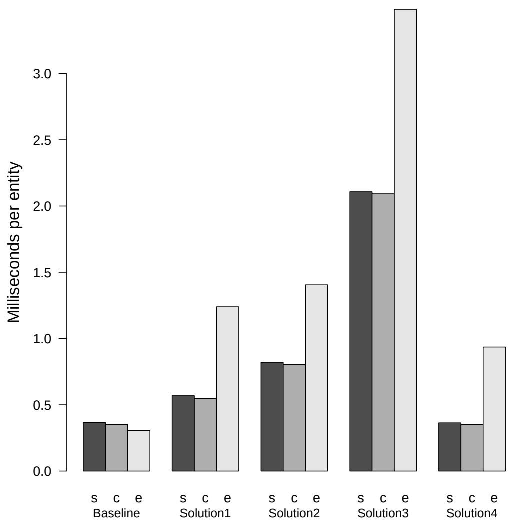
The performance of `delete` operations is similar across entities as well, and quite similar to the performance of the `insert` operations. This is because the tombstone delete paradigm in Cassandra does not allow a complete removal of the super columns, but rather it keeps the row keys and writes empty values to the columns to mark the super column as deleted. Thus, it is expected as well to perform similar to the `insert` operations.

Finally, the performance of the `update` operation is worse as it takes more time to complete because it involves `insert` and `delete` operations in the cases of `Course` and `Student`. However, the `update` operation in `Enrolment` is much better as it does not change the primary keys of these entities, instead, since only the foreign keys are changed. Thus, this operation on `Enrolment` acts as an `insert` operation.

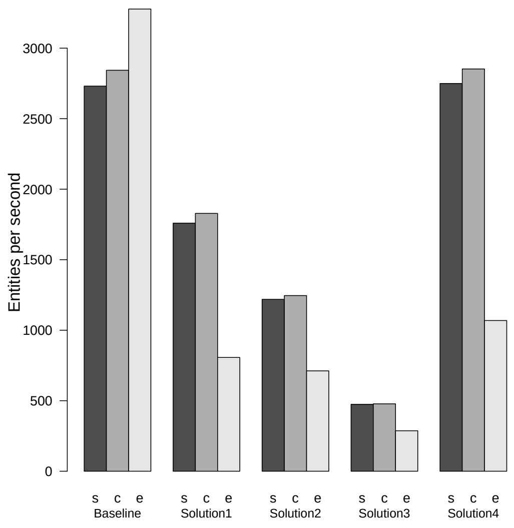
6.3 Insert

Across all the solutions in the experiments, `insert` triggers a validation when `Enrolment` entities are inserted as it is a child entity containing foreign keys referencing to `Student` and `Course`. On the other hand, `Student` and `Course` entities have no referential integrity constraints to be checked as these do not contain any references to other column families.

The average response time and throughput for completing an `insert` on a single entity for all the solutions is presented in Figure 6.2. Specifically, Figure 6.2(a) shows the average time consumed by the baseline and each solution to complete an `insert` operation, and Figure 6.2(b) shows the the number of `insert` operations that can be completed in one second.



(a) Response time for Insert operation



(b) Throughput for Insert operation

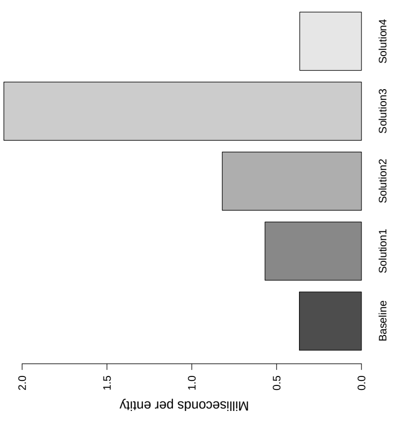
Figure 6.2: Performance of Solutions in Insert

These results show that `insert` on a single entity of `Student` and `Course` take approximately the same time to complete, and this is consistent across solutions. Inserting `Student` and `Course` entities into their respective column families is faster than `insert` in `Enrolment` because these are parent column families that have no referencing constraints. Thus, the `insert` operation on these entities involves only accessing the relevant FK constraints from the metadata in order to determine whether it is a parent or child entity. On the other hand, `insert` on `Enrolment` takes the most time in all the solutions as these entities have existing FK constraints indicating that they reference a parent entity which requires to retrieve additional constraints. Moreover, its validation involves not only identifying its relevant constraints but also accessing its parent column families (`Student` and `Course`) to ensure that foreign keys match primary keys. The results highlight the difference in response time when foreign key validations are required in the case of `Enrolment`. Note that these observations stand true across all the solutions.

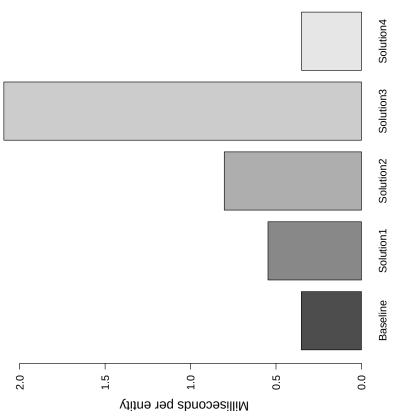
More detailed information about the performance of each solution when `insert` operations are performed is presented in Figures 6.3 and 6.4. These figures show the average response time and throughput for the `insert` operation on each entity individually. It can be seen that Solution 4 takes the least time to complete an `insert` on all the entities while Solution 3 takes the most time. Solution 4 takes the least time since it caches all the metadata thus avoiding multiple accesses to the `Metadata` column family, whereas Solution 3 requires accessing `Metadata` each time a constraint is required. Regarding Solutions 1 and 2, both perform similarly although Solution 2 takes slightly more time than Solution 1 due to its additional search operation to locate the top row. Both the solutions are slightly slower than Solution 4 as constraints from these solutions are retrieved from the column family and not from a cache. However, both solutions are faster than Solution 3 since retrieving constraints require no additional connections to access the metadata.

When compared to the baseline, it is clear that the referential integrity validations as well as metadata access caused the increased response time for `insert` in all the solutions. Since the validations are the same for all solutions, the performance differences in the solutions are due to the different ways of accessing and processing the metadata. From Table 6.2, Solutions 1 and 2 are almost 4 times slower than the baseline, while Solution 3 is more than 11 times slower, and Solution 4 is almost 3 times slower than the baseline.

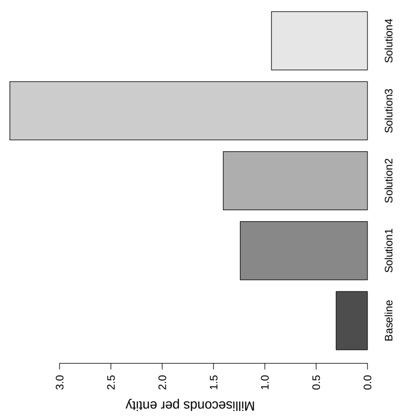
Notice that when no referential integrity constraints need to be satisfied (e.g. `Student` and `Course`), Solutions 1 and 2 are nearly 2 times slower than the baseline, Solution 3 more than 5 times slower, and Solution 4 almost similar to the baseline. Such differences are due to the computational cost incurred while retrieving the metadata.



(a) Insert on Student

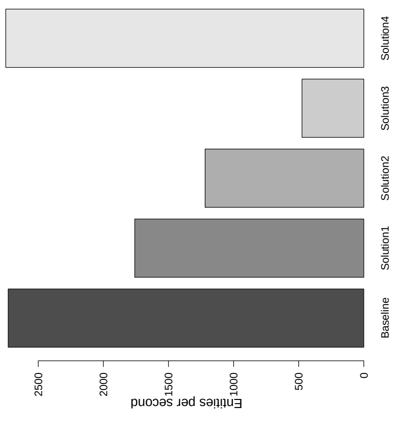


(b) Insert on Course

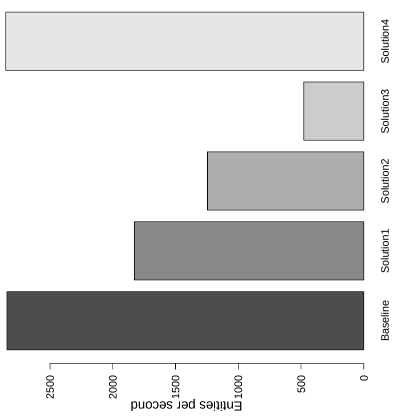


(c) Insert on Enrolment

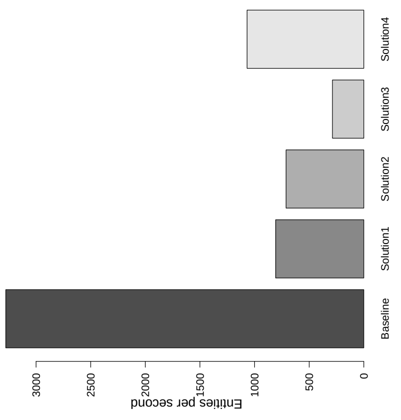
Figure 6.3: Response time inserting entities



(a) Insert on Student



(b) Insert on Course

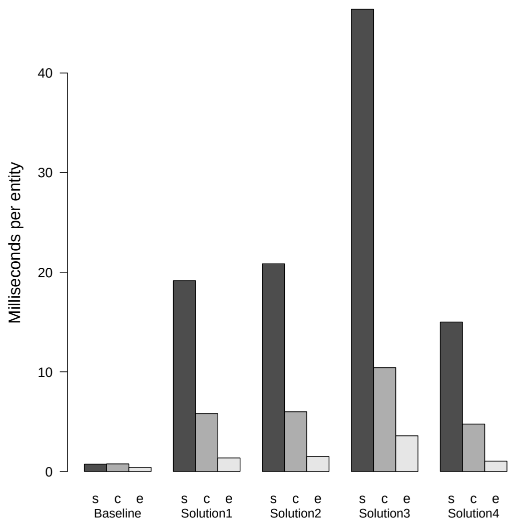


(c) Insert on Enrolment

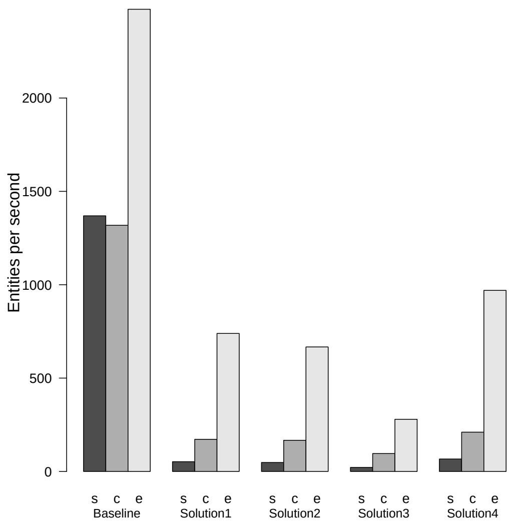
Figure 6.4: Throughput inserting entities

6.4 Update

The `update` operation triggers referential integrity validations whenever entities of `Student`, `Course` and `Enrolment` are updated with new values. Notice that the `Update` operation is performed on the primary keys of `Student` and `Course` entities, and on the foreign keys (`CourseId`) of `Enrolment`. Figure 6.5 presents the results of the `update` operation on each entity for all the solutions. Specifically, Figures 6.5(a) and 6.5(b) present the average response time and throughput of `update` respectively.



(a) Response time for Update operation



(b) Throughput for Update operation

Figure 6.5: Performance of Solutions in Update

It can be seen from the results that the `update` operation on `Enrolment` is faster than `update` on `Student` and `Course` in all the solutions. Updating `Enrolment` is faster since before inserting the new values, it only involves identifying relevant FK constraints in the metadata and accessing the parent column families `Student` and `Course` to ensure that the new foreign key values exist. Moreover, `update` in `Enrolment` involves changing the foreign key attributes and not the primary key column.

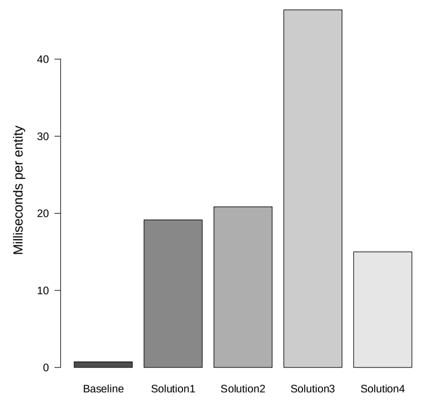
The `update` operation on `Student` is slower since the primary key is changed and it is a cascaded operation that updates `Enrolment` as well. After accessing the relevant FK constraints, the child dependencies are retrieved from `Enrolment` and updated with the new value for the `StudentId`. As such, this operation involves inserting a `Student` entity with a new `StudentId`, updating all the child entities of the old entity to the new entity and deleting the old `Student` entity.

Finally, the `update` operation on a `Course` takes less time than `update` on `Student` because it is not a cascaded operation as the `DeleteRule`[1] for `Course` entities is `NoDelete` and has child dependencies in `Enrolment`. Thus, exceptions are raised each time an `update` is attempted on `Course` entities, which is represented by the response time and throughput. That is, these indicators consider the time taken for referential integrity validations and exceptions. The `update` on `Course` is slower than on `Enrolment` because it involves accessing the `Enrolment` column family to identify existing child dependencies. Since these child dependencies exist when the experiments are run, the exceptions are raised each time.

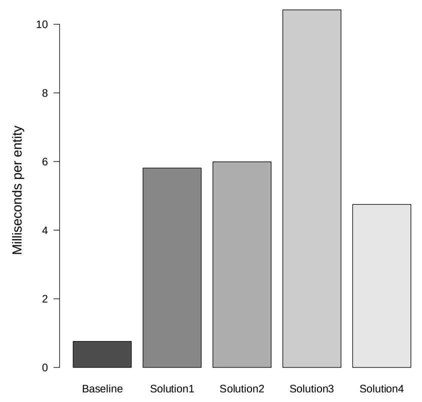
More detailed information about the performance of each solution is presented in Figures 6.6 and 6.7. These figures show that Solution 4 is the fastest amongst all the solutions, while Solution 3 is the slowest. Solutions 1 and 2 perform almost similarly although the additional search for the top row in Solution 2 makes it just slightly slower than Solution 1. Note that in Solution 3 the `Metadata` column family is accessed multiple times in each validation, thus making it the slowest. Multiple accesses are needed in such a case, to first retrieve the relevant FK constraints and then to retrieve information about the child or parent entities. Although Solution 4 stores metadata separately like Solution 3, it caches and re-uses the list of constraints, thus avoiding additional connections to the metadata cluster to access the `Metadata` column family each time operations are invoked on the entities.

[1]Notice that for the sake of simplicity, this rule is also used for update operations.

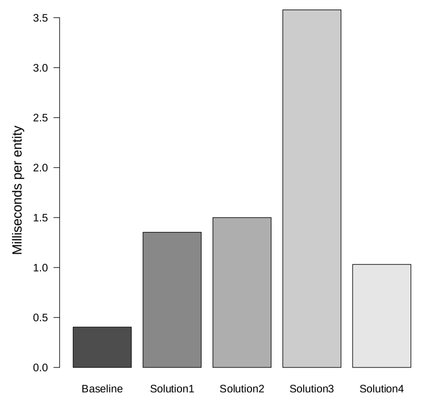
When compared to the baseline, the `update` operation takes considerably more time in all the solutions because of their different metadata storage and retrieval mechanisms. In this operation, when entities have child dependencies (`Student` and `Course`), Solutions 1 and 2 are more than 26 times slower than baseline, Solution 3 almost 60 times slower, while Solution 4 is only 20 times slower than the baseline in such updates. Differently, updates on child entities make Solution 1 and 2 more than 3 times slower than the baseline, Solution 3 nearly 8 times slower, while Solution 4 is only 2 times slower than the baseline in such updates.



(a) Update on Student

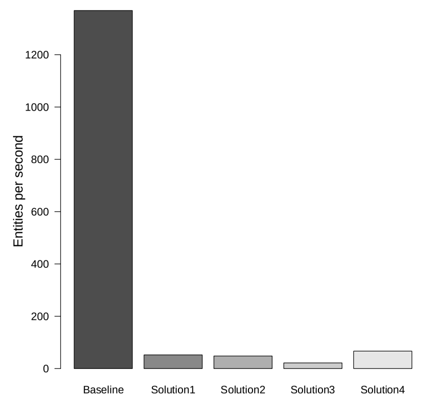


(b) Update on Course

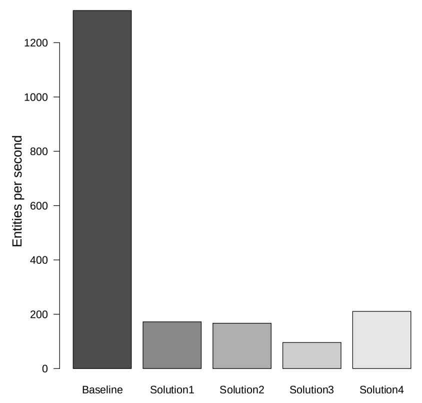


(c) Update on Enrolment

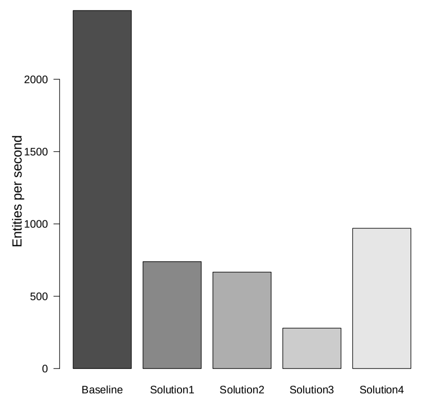
Figure 6.6: Response time updating entities



(a) Update on Student



(b) Update on Course

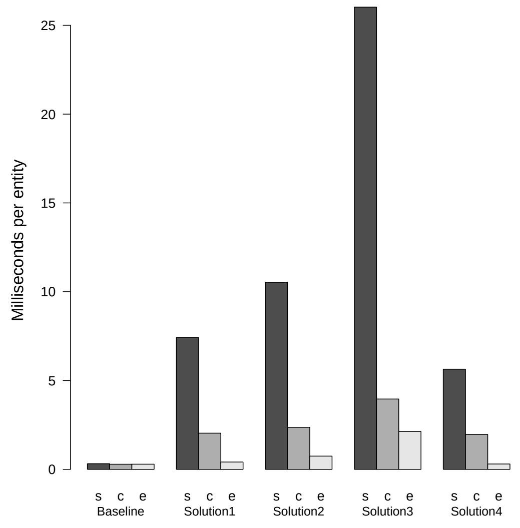


(c) Update on Enrolment

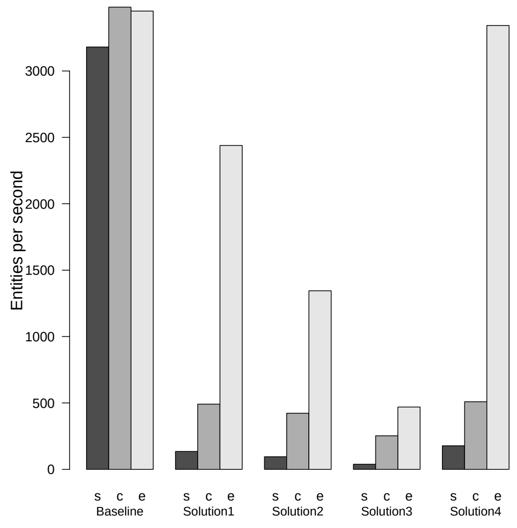
Figure 6.7: Throughput updating entities

6.5 Delete

The `delete` operation triggers referential integrity validations whenever entities are deleted. Figure 6.8 presents the results of the `delete` operation on each entity for all the solutions. Specifically, Figure 6.8(a) shows the average response time to perform a single `delete` on each entity according to each solution and Figure 6.8(b) presents the respective throughput for this operation.



(a) Response time for Update operation



(b) Throughput for Update operation

Figure 6.8: Performance of Solutions in Update

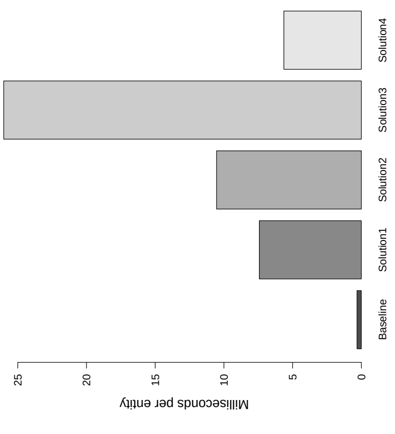
The results show that the `delete` operation on `Enrolment` is the fastest in all the solutions. Deleting `Enrolment` entities is faster as these have no referential integrity constraints to satisfy since `Enrolment` has no child dependencies in it. Nonetheless, this operation is slower than the baseline in all the solutions because it involves accessing metadata to retrieve the relevant constraints of `Enrolment` in order to determine if any child dependencies exist or not.

The `delete` operation on `Student` is the slowest. Deleting `Student` entities is a cascaded operation which involves deleting the child entities in the `Enrolment` column family. This operation is the slowest because the child entities in `Enrolment` which have a reference to `Student` have to be deleted first.

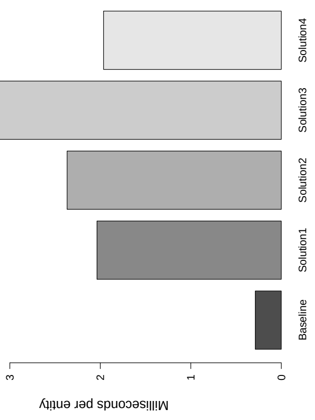
Finally, the `delete` operation on `Course` is faster than deleting `Student` entities. Deleting `Course` entities also involves accessing the relevant constraints and finding the child dependencies in `Enrolment`. However, at this stage, all the entities in `Enrolment` are already deleted before `delete` is invoked on `Course` entities. Hence, `delete` in `Course` actually deletes the entities as there are no existing child dependencies.

More detailed information about the performance of this operation can be seen in Figures 6.9 and 6.10. It can be seen from these results that Solution 4 takes the least time to complete a `delete` operation on each entity, while Solution 3 takes the most time. Since Solution 4 caches the metadata of all the entities, it avoids multiple accesses to the `Metadata` column family, whereas Solution 3 requires accessing `Metadata` each time a constraint has to be accessed for an entity. The performance of Solutions 1 and 2 are comparable to each other even though Solution 2 takes slightly more time due to its additional search operation to locate the top row.

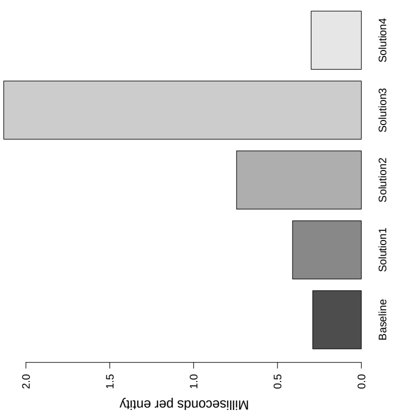
When compared to the baseline, all the solutions take longer to delete entities. As mentioned previously, this is because all the solutions involve accessing relevant constraints and performing referential integrity validations. In this operation, when entities have child dependencies (`Student` and `Course`), Solutions 1 and 2 are more than 23 times slower than the baseline, Solution 3 almost 80 times slower and Solution 4 up to 17 times slower than the baseline. On the other hand, deletes on child entities make Solutions 1 and 2 almost 2 times slower than baseline, Solution 3 almost 7 times slower ,while Solution 4 is almost similar to the baseline, which shows that accessing the metadata does not cause much difference in the performance.



(a) Delete on Student

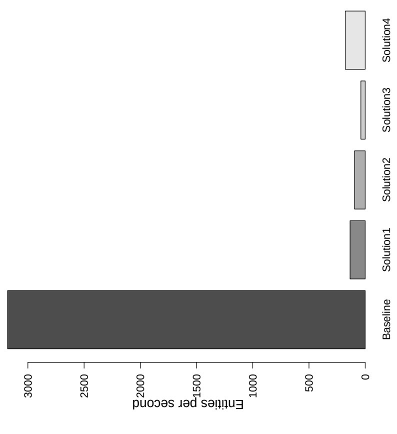


(b) Delete on Course

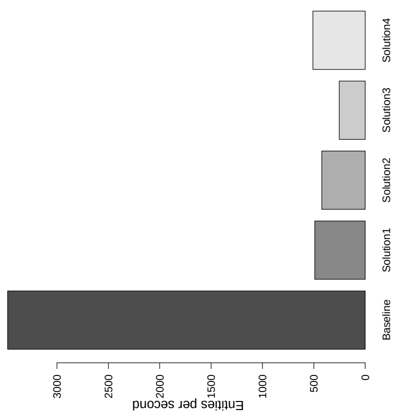


(c) Delete on Enrolment

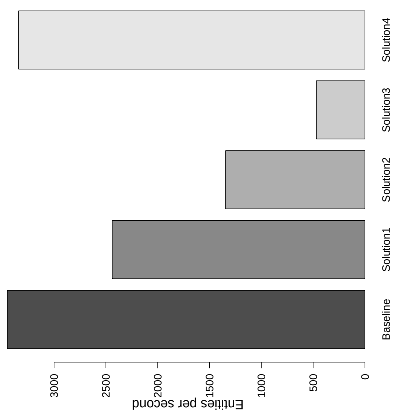
Figure 6.9: Response time deleting entities



(a) Delete on Student



(b) Delete on Course



(c) Delete on Enrolment

Figure 6.10: Throughput deleting entities

6.6 Comparison of the Operations

In order to compare the operations, their performance is grouped by solutions and presented in Figures 6.11 and 6.12. Generally, the `insert` operation takes the least time across the solutions as it does not involve any cascaded operations and referential integrity constraints have to be satisfied only for the child entities. This involves ensuring the existence of foreign keys as primary keys in the parent column families and inserting the child entities into their respective column families.

On the other hand, the `update` operation takes the most time in every solution, mainly due to its cascaded behaviour on parent entities, which involves changing the parent primary key, accessing child column families and changing its foreign key values. Note that `update` on `Enrolment` is similar to `insert` on `Enrolment` across the solutions, because both operations involve checking whether the foreign keys exist in the parent column families and inserting the values only in `Enrolment`. However, `update` on parent entities (`Student` and `Course`) take more time than inserting parent entities because `update` involves additional searches and operations (`insert` and `delete`) in both the parent and child column families.

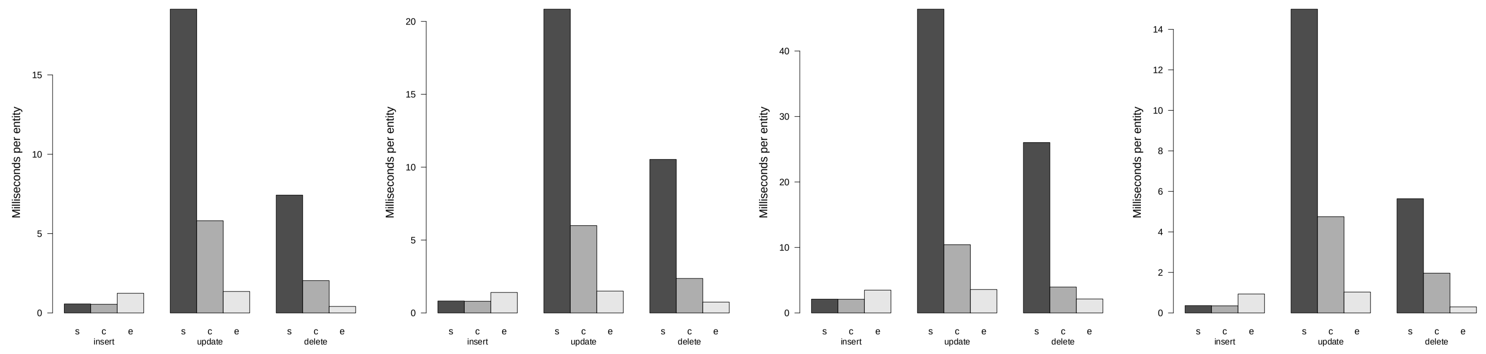
The `delete` operation is slower than the `insert` in the case of parent entities and faster than `insert` in the case of child entities. This is because referential integrity constraints have to be satisfied only in the case of the parent entities for this operation. However, in general, deleting entities is faster than updating them. This is because updating entities involves more operations as both `insert` and `delete` are performed on child and parent column families, while deleting entities involves inserting empty values in the place of the entity attributes to mark them as deleted (tombstone effect). Moreover, in `update`, referential integrity constraints need to be satisfied for both parent and child entities, but in `delete`, these have to be satisfied only for the parent entities.

Further information about the results for the operations and solutions are provided in Appendices A, B, and C.

6.7 Summary

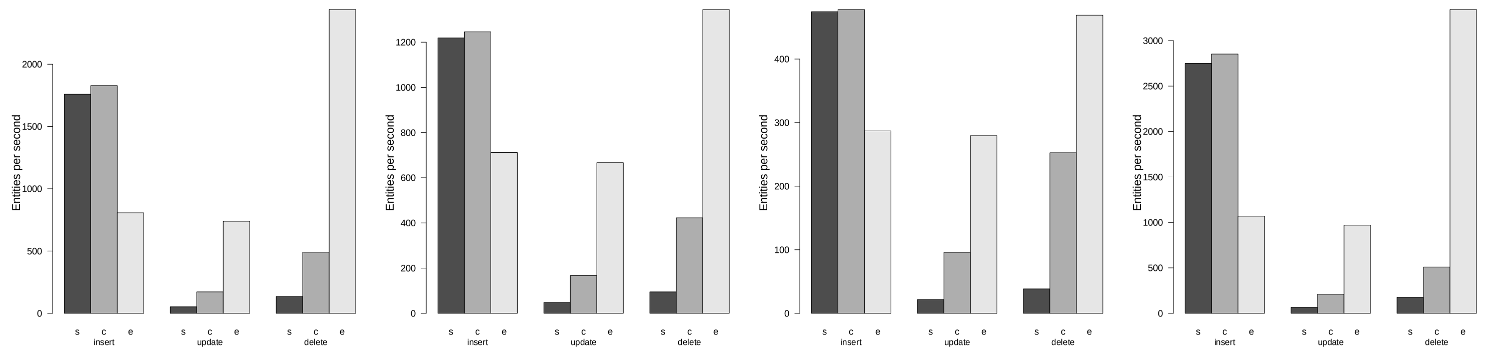
This chapter presented the results and discussions from the experiments designed to evaluate the performance of the different CRUD operations, under different referential integrity constraints, in each of the solutions. The results were assessed in terms of the average response time and throughput of each operation. The results reflected that Solution 4 performs the best amongst the solutions, and performs similar to the baseline when no referential integrity constraints need to be satisfied (e.g. inserting parent entities), because it caches the metadata and re-uses it to avoid multiple access to the `Metadata` column family. Solution 3 performs the worst amongst all solutions and is slower than the baseline even when no referential integrity constraints need to be satisfied because simply accessing the metadata from a separate column family each time affects its performance. Solutions 1 and 2 perform similarly in all the operations on the entities, which is mainly because the metadata is embedded with the actual data. Solution 2 consumes slightly more time than Solution 1 as it searches for the top row to identify constraints on each operation.

The results showed that amongst the operations, `insert` took the least time while `update` took the most time, and `delete` was faster than `insert` only in the case of child entities. These variations were mainly due to the different referential integrity rules that are applied on parent and child entities, especially because of the `DeleteRule` applied on these entities.



(a) Solution 1 (b) Solution 2 (c) Solution 3 (d) Solution 4

Figure 6.11: Response Time of the Solutions



(a) Solution 1 (b) Solution 2 (c) Solution 3 (d) Solution 4

Figure 6.12: Throughput of the Solutions

The entities required different behaviours in each operation due to the various referential integrity rules as well as the data manipulation rules applied on them. `Enrolment` entities required to satisfy referential integrity constraints during `insert` and `update` operations as it is a child entity, while `Student` and `Course` are parent entities and required to satisfy these constraints in both `update` and `delete`. Thus, parent entities are faster to operate upon in an `insert` operation, while child entities are faster only in a `delete` operation.

Chapter 7

Conclusions and Future Work

Database Management Systems (DBMSs) on the cloud commonly adopt the key-value data model which stores data as key-value pairs without strict or rigid schemas (contrary to the traditional relational model). This key-value data model provides cloud Not only SQL (NoSQL) DBMSs with features like scalability, flexibility and robustness, which are essential in the cloud environment. However, in order to provide these, other features such as referential integrity are often sacrificed.

In such cloud DBMSs, maintaining referential integrity incurs an additional computational cost that is avoided in order to provide high data availability and partition-tolerance, following the CAP theorem [8, 52]. However, when referential integrity is not maintained, data dependencies are not necessarily correct as dangling references can be introduced. Hence, maintaining referential integrity is left to be dealt with by the application.

This thesis has incorporated validation mechanisms for maintaining referential integrity in Apache Cassandra, a prominent cloud column-oriented key-value DBMS. These mechanisms are provided within an Application Programming Interface (API) that serves as a middle layer between the applications and the DBMS. The API prevents the execution of operations that violate referential integrity constraints and does so by triggering validations which ensure that referential integrity constraints applied on data are always satisfied. More importantly, the API presents four approaches, as solutions, to store the referential integrity constraints as metadata.

The performance of each solution is analyzed in terms of response time and throughput upon a set of experiments. These experiments were designed to validate and test the API and the four solutions and performed Create, Read, Update and Delete (CRUD) operations upon artificial data created for an example application, specifically designed for this purpose. These experiments were developed to determine how each solution affects the performance of the CRUD operations, specifically those where referential integrity validations are triggered (namely Create, Update and Delete). In order to perform reliable experiments and obtain unbiased results, a homogeneous set of nodes in the university labs were used and manually configured to form a Cassandra cluster. These experiments presented results which showed clear differences in performance across the solutions. The results helped in clearly understanding the performance of each solution as it was compared and analysed against the baseline experiment where no referential integrity validations were implemented. However, the final word on which one is better depends on the application as each solution presents different trade-offs.

The first solution has the second-best performance as it embeds the metadata within each entity, thus providing immediate access to the constraints once the entity is retrieved. However, this solution requires a large amount of disk space as it stores the metadata within every entity. Additionally, if changes are to be performed on the metadata, these must be updated in all entities, thus making its management difficult. While this solution has a rather good performance due to its quick access to the constraints, the trade-off involves large space requirements coupled with complex metadata management *versus* good performance.

The second solution has a slightly worse performance than the first one as it stores the metadata in the top row of each column family. This solution requires less disk space as metadata is not included within each entity.

Also, its management is much simpler as changes in metadata require only updating it in every column family. However, compared with the previous solution, this one requires an additional search operation to retrieve the metadata, therefore the performance is slightly worse. In this case, the trade-off to decide upon involves lower disk space requirements and simpler management of metadata *versus* the delay induced by an additional search operation to retrieve the metadata.

The third solution has the worst performance of all. In this case, the constraints relevant to every entity are stored in a single column family named `Metadata`, thus providing centralized metadata storage. Moreover, changes to the constraints require updating only the `Metadata` column family. However, the performance is significantly reduced because validations have to perform additional accesses to `Metadata` to retrieve the relevant constraints. Thus, the trade-off in this approach is low disk space requirements and simple management of metadata *versus* a poor performance.

Lastly, the fourth solution has the best performance of all. It stores the metadata in a single column family (as the previous solution), but in a separate dedicated cluster. This approach requires to connect to such a cluster every time metadata needs to be accessed. However, once metadata is retrieved for the first time, it is stored in cache in order to re-use it and avoid future connections to the cluster. Caching is the key aspect that makes this solution have a superior performance than the rest, but when metadata is altered, additional mechanisms are required to properly have the cache updated. The trade-off here is between high performance, low disk space requirements, and simple management of metadata *versus* having a potentially stale cache or implementing mechanisms to prevent it.

In summary, this thesis has presented a study about the existing data models used by cloud DBMSs, and explored the challenge of imposing referential integrity constraints, especially in cloud NoSQLs DBMSs. Based on this, an API is designed and implemented to maintain referential integrity in Apache Cassandra. More importantly, four solutions to store the referential integrity constraints were devised and incorporated into the API. A set of experiments was performed and their results were analysed to asses the performance of the solutions. The results consolidated the metadata storage as an influential and important aspect regarding the performance of referential integrity validations in cloud NoSQL DBMSs. Furthermore, it was concluded that the solutions present different trade-offs between performance, disk space requirements, and metadata management, all of which have to be carefully considered by applications in order to choose the most appropriate solution according to their demands.

This research can be further extended by:

- Incorporating mechanisms in the API to support thread-safe operations on the data such that referential integrity is not compromised when multiple concurrent operations take place. In order to achieve this, locking mechanisms and transaction support for Apache Cassandra might be implemented using libraries such as Cages.

- Implementing trigger procedures in order to initiate events before and after performing any CRUD operations. This could be implemented by extending the validation handlers to execute methods for such events in every operation. Also, entities could provide annotations for such methods.

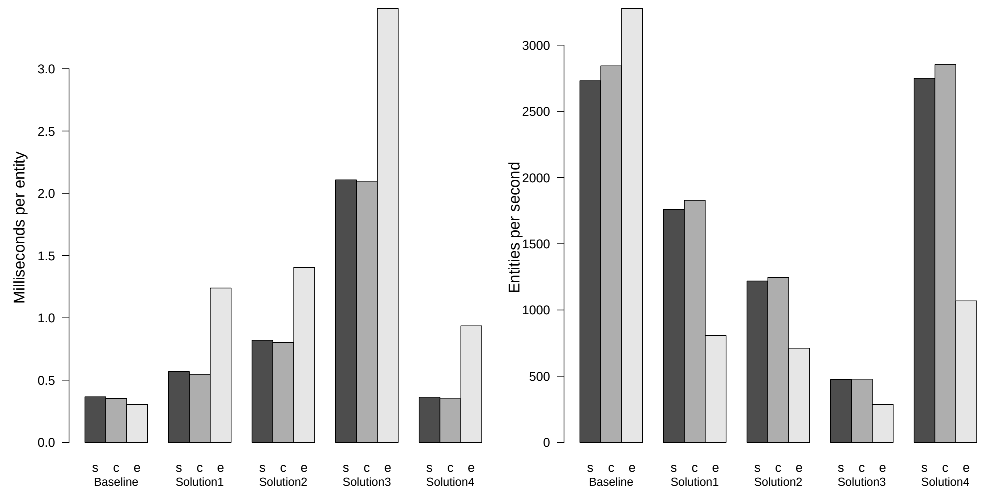
- Implementing the four solutions on another key-value DBMS such as Google's BigTable or Apache's HBase in order to compare the performance with the results presented in this thesis.

- Making the API DBMS-agnostic. That is, allowing the possibility to switch the underlying cloud NoSQL DBMS with minimal reconfiguration in order to provide consistent usage of the API on DBMSs such as BigTable, HBase, and others.

- Incorporating constraints for composite primary keys and implementing their respective handling.
- Evaluating the performance impact of Hector by experimenting with other APIs for Cassandra such as Pelops or Kundera.
- Adapting the API to work on a real cloud environment such as Amazon EC2 where heterogeneity and dynamism are inherent properties as nodes may have different characteristics and more nodes can be added or removed as well. Moreover, useful information about the solutions proposed in this thesis can be drawn from performing the experiments in such environments.
- Consolidating the performance of the API and the solutions by using additional benchmarks such as Yahoo Cloud Service Benchmark (YCSB) which measures latency, scalability, and other features of cloud DBMSs.

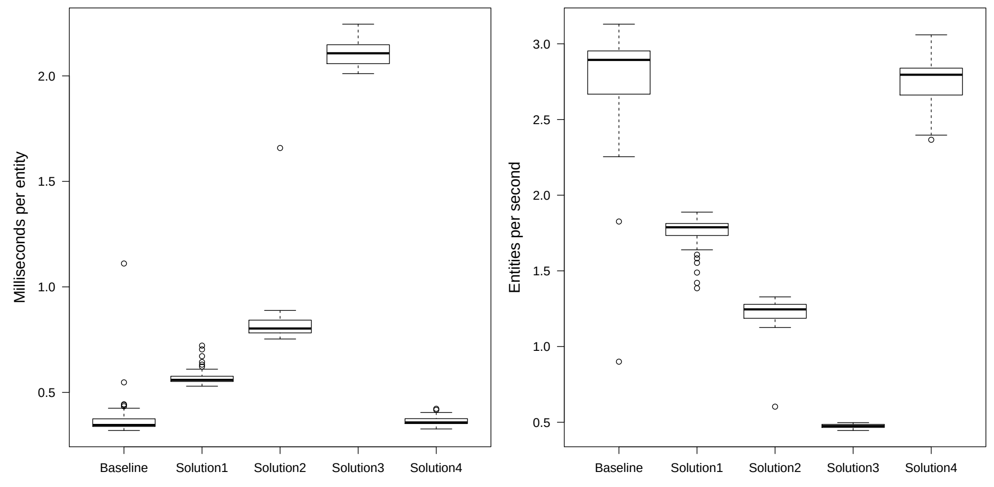
Appendices

Appendix A
Insert



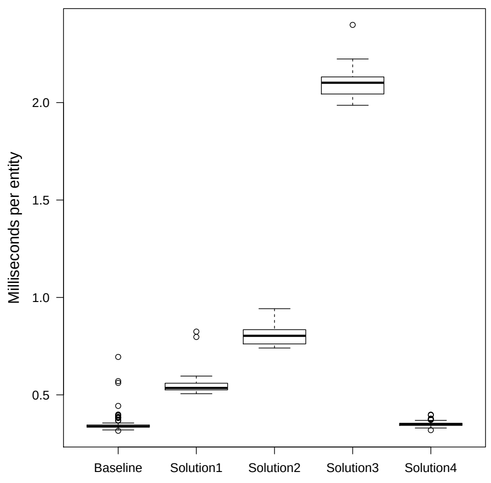
(a) Response time (b) Throughput

Figure A.1: Performance of `insert`

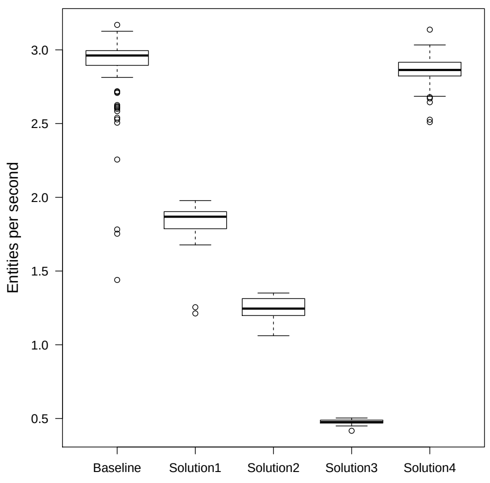


(a) Response time (b) Throughput

Figure A.2: Performance of `insert` students

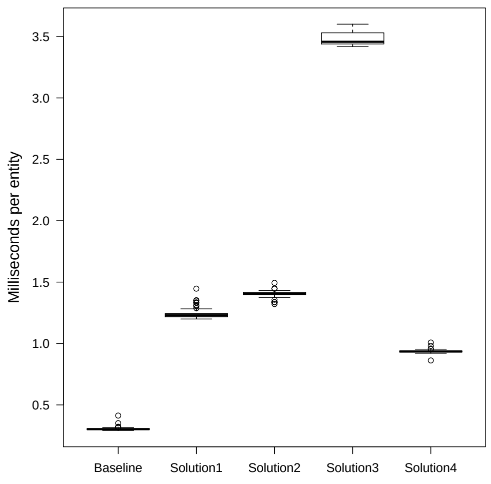


(a) Response time

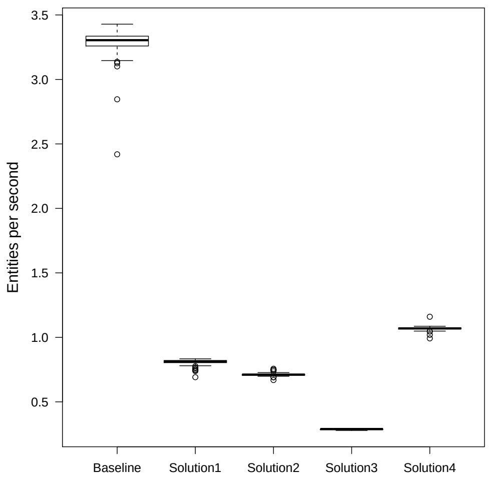


(b) Throughput

Figure A.3: Performance of `insert` courses



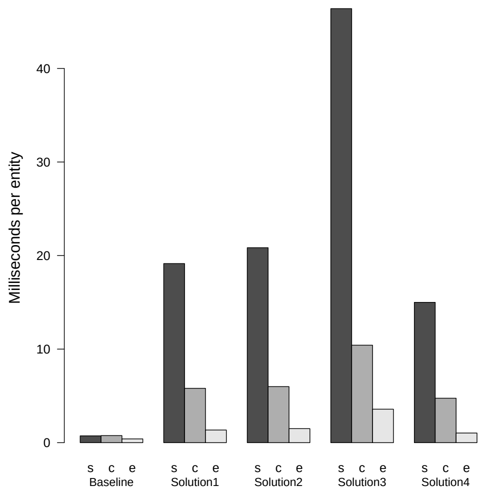
(a) Response time



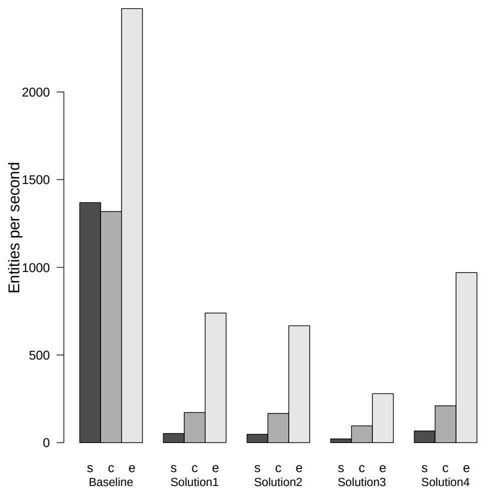
(b) Throughput

Figure A.4: Performance of `insert` enrolments

Appendix B
Update

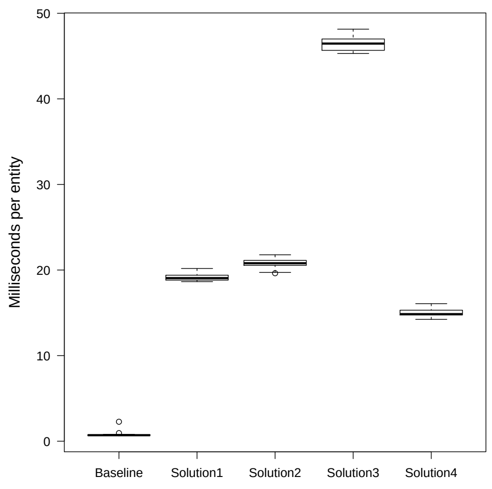


(a) Response time

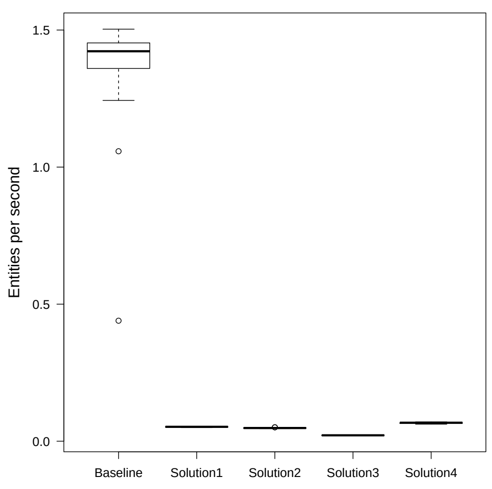


(b) Throughput

Figure B.1: Performance of `update`

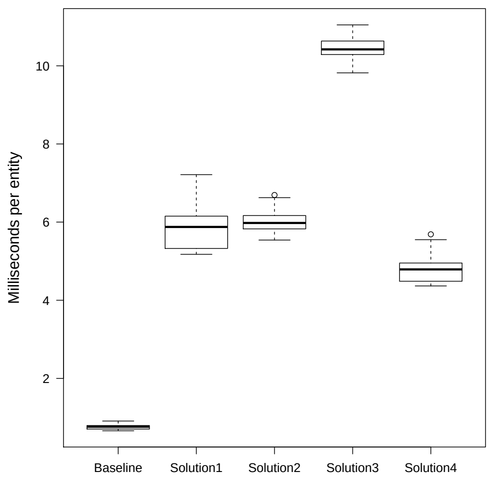


(a) Response time

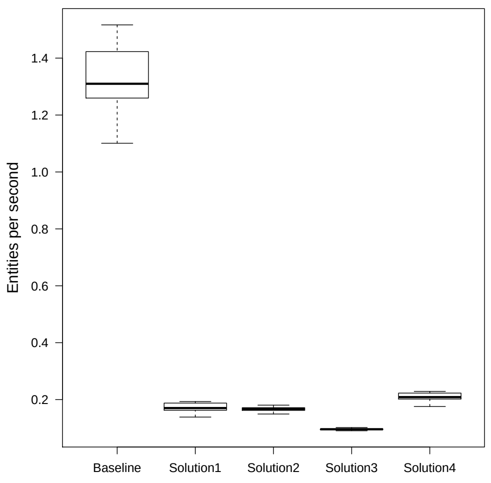


(b) Throughput

Figure B.2: Performance of `update` students

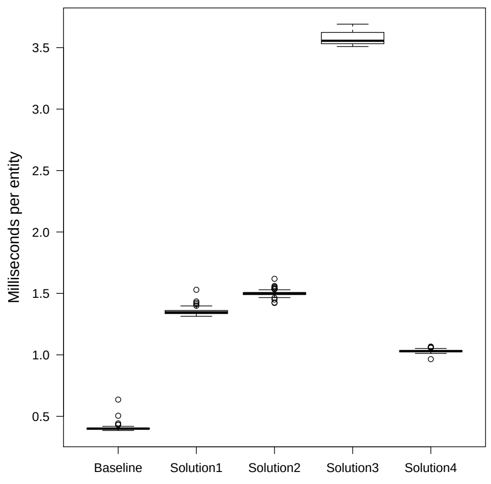


(a) Response time

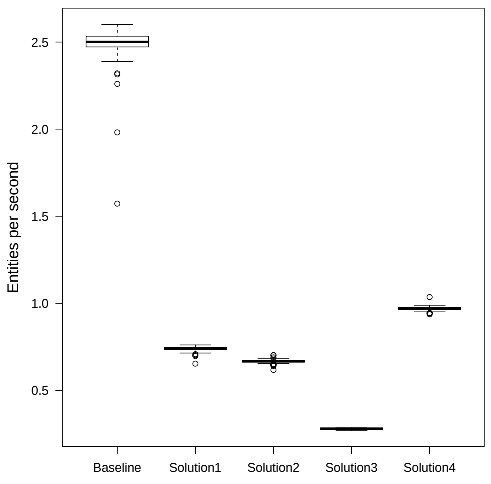


(b) Throughput

Figure B.3: Performance of `update` courses



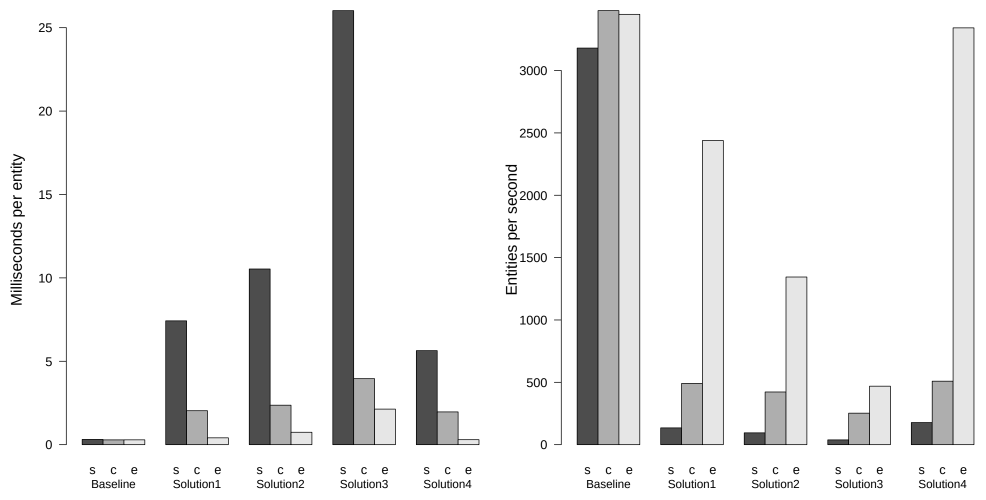
(a) Response time



(b) Throughput

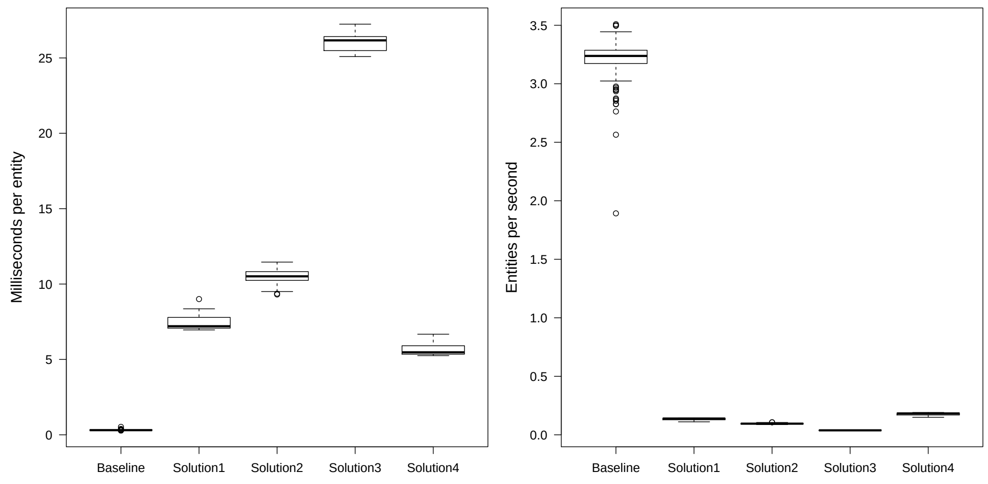
Figure B.4: Performance of `update` enrolments

Appendix C
Delete



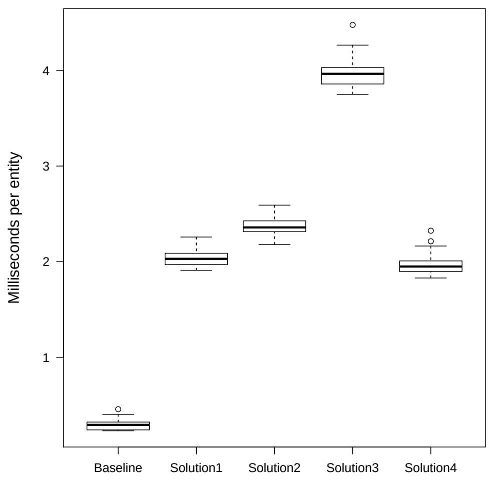
(a) Response time (b) Throughput

Figure C.1: Performance of `delete`

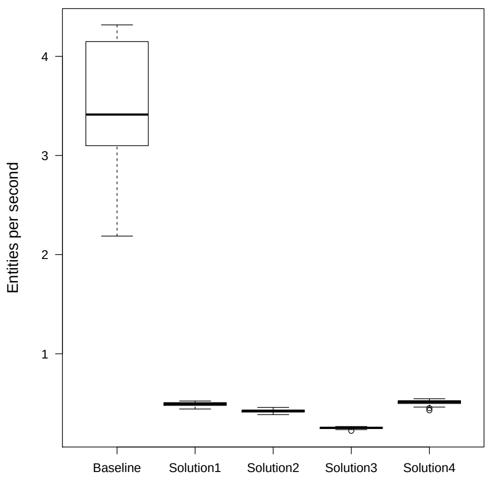


(a) Response time (b) Throughput

Figure C.2: Performance of `delete` students

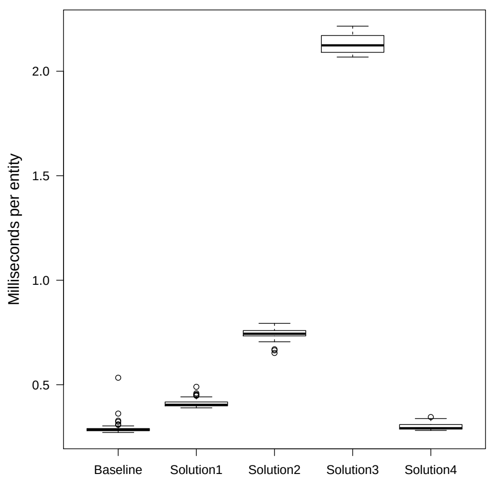


(a) Response time

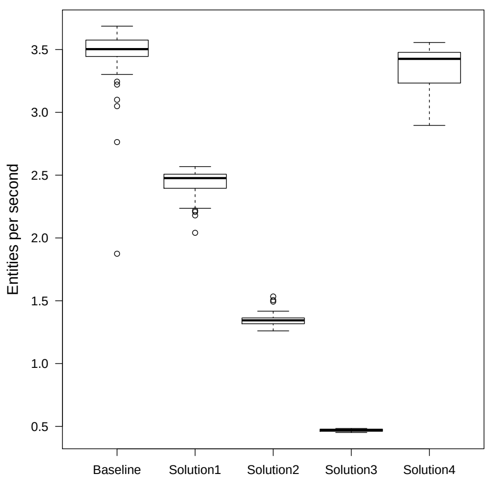


(b) Throughput

Figure C.3: Performance of `delete` courses



(a) Response time



(b) Throughput

Figure C.4: Performance of `delete` enrolments

Bibliography

[1] D. Abadi. Problems with CAP, and Yahoo's little known NoSQL system, 2010. [Online]. Available: `http://dbmsmusings.blogspot.co.nz/2010/04/problems-with-cap-and-yahoos-little.html` [Accessed: July 2011].

[2] D. J. Abadi. Data Management in the Cloud: Limitations and Opportunities. *IEEE Data Engineering Bulletin*, 32(1):3–12, 2009.

[3] S. Bell and P. Brockhausen. Discovery of constraints and data dependencies in relational databases (extended abstract). In *Machine Learning: ECML-95*, volume 912, pages 267–270. Springer Berlin / Heidelberg, 1995.

[4] D. Bermbach and S. Tai. Eventual Consistency: How soon is eventual? An evaluation of Amazon S3's consistency behavior. In *Proceedings of the 6th Workshop on Middleware for Service Oriented Computing*, MW4SOC '11, pages 1–6, New York, NY, USA, 2011. ACM.

[5] H. Bin and P. Yuxing. A Novel Metadata Management Scheme in Cloud Computing. In *Proceedings of the 2nd International Conference on Software Technology and Engineering*, volume 1, pages 433–438, 2010.

[6] M. Blaha. Referential integrity is important for databases, 2005. Modelsoft Consulting Corporation.

[7] H. Boral and D. J. DeWitt. A Methodology for Database System Performance Evaluation. In *Proceedings of the 1984 ACM SIGMOD International Conference on Management of data*, SIGMOD'84, page 176, Boston, Massachusetts, 1984.

[8] E. A. Brewer. Towards robust distributed systems (abstract). In *Proceedings of the 19th annual ACM Symposium on Principles of distributed computing*, PODC '00, page 7, New York, NY, USA, 2000. ACM.

[9] E. A. Brewer. Pushing the CAP: Strategies for Consistency and Availability. *IEEE Computer*, 45(2):23–29, 2012.

[10] J. Browne. Brewer's CAP Theorem, 2009. [Online]. Available: `http://www.julianbrowne.com/article/viewer/brewers-cap-theorem` [Accessed: August 2011].

[11] R. Buyya, J. Broberg, and A. M. Goscinski. *Cloud Computing Principles and Paradigms*. Wiley Publishing, 2011.

[12] D. G. Campbell, G. Kakivaya, and N. Ellis. Extreme scale with full SQL language support in microsoft SQL Azure. In *Proceedings of the 2010 international conference on Management of data*, SIGMOD '10, pages 1021–1024, New York, NY, USA, 2010. ACM.

[13] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: a Distributed Storage System for Structured Data. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation - Volume 7*, OSDI '06, pages 15–15, Berkeley, CA, USA, 2006. USENIX Association.

[14] CloudComputingDefined. Cloud computing defined. [Online]. Available: `http://www.cloudcomputingdefined.com/` [Accessed: March 2012].

[15] B. F. Cooper. The Prickly Side of Building Clouds. *IEEE Internet Computing*, 14:64–67, 2010.

[16] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SOCC '10, pages 143–154, New York, NY, USA, 2010. ACM.

[17] DataStax. Apache Cassandra 0.8 Documentation, 2011. [Online]. Available: `http://www.datastax.com/docs/0.8/dml/about_writes` [Accessed: January 2012].

[18] DataStax. Understanding the Cassandra Data Model, 2011. [Online]. Available: `http://www.datastax.com/docs/0.8/ddl/index` [Accessed: October 2011].

[19] DataStax. Apache Cassandra 0.8 Documentation, 2011. [Online]. Available: `http://www.datastax.com/docs/0.8/dml/about_reads` [Accessed: January 2012].

[20] DataStax. Apache Cassandra Backgrounder, 2012. [Online]. Available: `http://www.datastax.com/wp-content/uploads/2011/02/DataStax-cBackgrounder.pdf` [Accessed: February 2012].

[21] C. J. Date. Referential Integrity. In *Proceedings of the 7th International Conference on Very Large Data Bases*, volume 7, pages 2–12, Cannes, France, 1981. VLDB Endowment.

[22] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. In *Proceedings of 21st ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, pages 205–220, New York, NY, USA, 2007. ACM.

[23] D. DeWitt, S. Madden, and M. Stonebraker. How to build a high-performance data warehouse. [Online]. Available: `http://db.lcs.mit.edu/madden/high_perf.pdf` [Accessed: June 2011].

[24] P. Echague. Hector A high level Java client for Apache Cassandra, 2011. [Online]. Available: `http://rantav.github.com/hector/build/html/index.html` [Accessed: October 2011].

[25] R. Elmasri and S. B. Navathe. *Fundamentals of Database Systems, 2nd Edition*. Benjamin/Cummings, 1994.

[26] D. Florescu and D. Kossmann. Rethinking Cost and Performance of Database Systems. *ACM Special Interest Group on Management of Data*, 38(1):43–48, 2009.

[27] Y. Fu, N. Xiao, and E. Zhou. A Novel Dynamic Metadata Management Scheme for Large Distributed Storage Systems. In *Proceedings of the 10th IEEE International Conference on High Performance Computing and Communications*, pages 987–992, 2008.

[28] C. George, H. Miao, and M. Hale. Maintaining Referential Integrity on the Web. In *Formal Methods and Software Engineering*, volume 2495, pages 20–21. Springer Berlin / Heidelberg, 2002.

[29] S. Gilbert and N. Lynch. Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-tolerant Web Services. *ACM Special Interest Group on Algorithms and Computation Theory News*, 33(2):51–59, 2002.

[30] G. Hackl, W. Pausch, S. Schonherr, G. Specht, and G. Thiel. Synchronous Metadata Management of Large Storage Systems. In *Proceedings of the 14th International Database Engineering ; Applications Symposium*, IDEAS '10, pages 1–6, New York, NY, USA, 2010. ACM.

[31] J. Han, E. Haihong, G. Le, and J. Du. Survey on NoSQL databases. In *Proceedings of the 6th International Conference on Pervasive Computing and Applications*, ICPCA 2011, pages 363 –366, 2011.

[32] Henry. Consistency and Availability in Amazon's Dynamo, 2008. [Online]. Available: `http://the-paper-trail.org/blog/consistency-and-availability-in-amazons-dynamo/` [Accessed: February 2012].

[33] E. Hewitt. *Cassandra: The Definitive Guide*. Definitive Guide Series. O'Reilly Media, 2010.

[34] Hewlett Packard. There is no free lunch with distributed data, 2005. [Online]. Available: `ftp://ftp.compaq.com/pub/products/storageworks/whitepapers/5983-2544EN.pdf` [Accessed: August 2011].

[35] M. Hirabayashi. Tokyo Cabinet: A Modern Implementation of DBM, 2010. [Online]. Available: `http://fallabs.com/tokyocabinet/` [Accessed: May 2011].

[36] M. Hogan. Cloud Computing and Databases, 2008. Scale DB Inc.

[37] N. Kennedy. The Anatomy of Cloud Computing, 2009. [Online]. Available: `http://www.niallkennedy.com/blog/2009/03/cloud-computing-stack.html` [Accessed: Jan 2012].

[38] A. Lakshman and P. Malik. Cassandra: A Decentralized Structured Storage System. *ACM SIGOPS Operating Systems Review*, 44(2):35–40, 2010.

[39] V. Mateljan, D. Cisic, and D. Ogrizovic. Cloud Database-as-a-Service (DaaS)-ROI. In *Proceedings of the 33rd International Convention on Information and Communication Technology, Electronics and Microelectronics*, MIPRO '10, pages 1185 –1188. Rijeka, 2010.

[40] Microsoft. Scaling out with SQL Azure, 2010. [Online]. Available: `http://www.microsoft.com/download/en/details.aspx?id=13300` [Accessed: October 2011].

[41] Microsoft. System Tables SQL Server 2000, 2011. [Online]. Available: `http://msdn.microsoft.com/en-us/library/aa260604(v=sql.80).aspx` [Accessed: May 2011].

[42] C. Pathivada. Avoid Referential Integrity Errors When Deleting Records from Databases. *SQL Server Magazine*, 11(6):15, 2009.

[43] PostgreSQL Global Development Group. PostgreSQL 8.3.17 Documentation, 2011. [Online]. Available: `http://www.postgresql.org/docs/8.3/static/infoschema-table-constraints.html` [Accessed: May 2011].

[44] D. Pritchett. BASE: An ACID Alternative. *ACM Queue - Object-Relational Mapping*, 6(3):48–55, 2008.

[45] A. Rahien. Scaling out with SQL Azure, 2010. [Online]. Available: `http://ayende.com/Blog/archive/2010/04/11/that-no-sql-thing-ndash-document-databases.aspx` [Accessed: June 2011].

[46] R. Ramakrishnan. CAP and Cloud Data Management. *IEEE Computer*, 45(2):43 –49, 2012.

[47] Y. Shi, X. Meng, J. Zhao, X. Hu, B. Liu, and H. Wang. Benchmarking Cloud-based Data Management Systems. In *Proceedings of the 2nd International Workshop on Cloud Data Management*, CloudDB '10, pages 47–54, New York, NY, USA, 2010. ACM.

[48] SNIA. Cloud storage for cloud computing, 2009. [Online]. Available: `http://ogf.org/Resources/documents/CloudStorageForCloudComputing.pdf` [Accessed: April 2011].

[49] J. Spring. Monitoring Cloud Computing by Layer, Part 1. *Security Privacy, IEEE*, 9(2):66 –68, 2011.

[50] J. Spring. Monitoring Cloud Computing by Layer, Part 2. *Security Privacy, IEEE*, 9(3):52 –55, 2011.

[51] M. Stonebraker. SQL databases vs. NoSQL databases. *Communications of the ACM*, 53(4):10–11, 2010.

[52] M. Stonebraker, S. Madden, D. J. Abadi, S. Harizopoulos, N. Hachem, and P. Helland. The end of an architectural era: (it's time for a complete rewrite). In *Proceedings of the 33rd International Conference on Very large data bases*, VLDB '07, pages 1150–1160. VLDB Endowment, 2007.

[53] D. B. Terry, M. M. Theimer, K. Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser. Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System. In *Proceedings of the 15th ACM Symposium on Operating systems principles*, SOSP '95, pages 172–182, New York, NY, USA, 1995. ACM.

[54] W. Vogels. Eventually consistent. *Communications of the ACM*, 52(1): 40–44, 2009.

[55] H. Wada, A. Fekete, L. Zhao, K. Lee, and A. Liu. Data consistency properties and the trade-offs in commercial cloud storage: the consumers' perspective. In *Proceedings of the 5th Biennial Conference on Innovative Data Systems Research*, CIDR '11, pages 134–143, 2011.

[56] I. Wilkes. Cloud storage in a post-SQL world, 2010. [Online]. Available: `http://arstechnica.com/business/data-centers/2010/02/-since-the-rise-of.ars` [Accessed: March 2011].

[57] J. Wu, L. Ping, X. Ge, Y. Wang, and J. Fu. Cloud Storage as the Infrastructure of Cloud Computing. In *Proceedings of International Conference on Intelligent Computing and Cognitive Informatics*, ICICCI '10, pages 380 –383, 2010.

[58] L. Xia, H. Duan, L. Li, and X. Nie. A Design of Efficient Metadata Cluster in Large Distributed Storage Systems. In *Proceedings of International Conference on Apperceiving Computing and Intelligence Analysis*, ICACIA '09., pages 294–296, 2009.