UNIVERSITY OF TARTU
Institute of Computer Science
Software Engineering Curriculum

Soltan Garayev

SCHEME DESIGN IN NON-RELATIONAL MODEL DATABASE TO MIGRATE DATA FROM RELATIONAL MODEL DATABASE

Master Thesis (30 ECTS)

Supervisor(s): Pelle Jakovits, PhD

Tartu 2020

Scheme Design in Non-Relational Model Database to Migrate Data from Relational Model Database

Abstract:
The usual preference of storing data is Relational Databases for enterprise applications. It provides relational model, and helps to keep data in organized structure. For last few decades relational database models have been the first choice of the companies. Things started to change with the term of Big Data which is about storing all kind of data related to end-user behaviour. User behaviour is random which can be understood non-structural as well. In this scenario usual relational database model does not fulfill the requirements. There was a need, and a new concept named NoSQL was the best option to fill this gap. NoSQL means Not Only SQL, and it provides a suitable environemtn to companies to store data in any form without having to define a structure. As time passes non-relational databases became priority in other concepts such as storing not only randomly structured data but structured data as well. If we think about big tech companies like Facebook or Google usual relational database model is unsatisfactory for storing huge amount of data. Companies started to move data from relational database to non-relational databases to store their data. There are articles/papers on data migration from relational to non-relation databases. The main issue here is designing the scheme in non-relational data model based on existing complex relational data model. In this research we aim to suggest ways to solve this issue.

Keywords:
Data Migration, RDBMS, Relational Database Management System, Relational Data Model, Non-Relational Data Model, NoSQL, Postgres, PostgreSQL, Cassandra, Scheme Design

CERCS: P170 Computer science, numerical analysis, systems, control

Skeemikujundus mitterelatsionaalses mudeliandmebaasis andmete migreerimiseks relatsioonimudelite andmebaasist

Abstraktne:
Tavaline eelistus andmete salvestamiseks on ettevõtterakenduste relatsiooniandmebaasid. See pakub relatsioonimudelit ja aitab hoida andmeid organiseeritud struktuuris. Viimaste aastakümnete jooksul on ettevõtete esimene valik olnud relatsioonilised andmebaasimudelid. Asjad hakkasid muutuma seoses suurandmete terminiga, mis seisneb igasugu lõppkasutaja käitumisega seotud andmete talletamises. Kasutaja käitumine on

juhuslik, mida võib mõista ka mittestruktuursena. Selle stsenaariumi korral ei vasta tavaline relatsioonilise andmebaasi mudel nõuetele. Vajadus oli ja uus kontseptsioon nimega NoSQL oli parim võimalus selle lünga täitmiseks. NoSQL tähendab mitte ainult SQL-i ja see pakub ettevõtetele sobivat keskkonda andmete salvestamiseks mis tahes kujul, ilma et oleks vaja struktuuri määratleda. Aja möödudes muutusid mitteseotud andmebaasid prioriteetseks teistes mõistetes, näiteks mitte ainult juhuslikult struktureeritud andmete, vaid ka struktureeritud andmete talletamine. Kui mõtleme sellistele suurtele tehnoloogiaettevõtetele nagu Facebook või Google, on tavaline relatsioonilise andmebaasi mudel ebarahuldav tohutu hulga andmete talletamiseks. Ettevõtted asusid andmete salvestamiseks teisaldama relatsiooniandmebaasidest mitterelatsioonandmebaasidesse. On artikleid / artikleid andmete migratsiooni kohta relatsioonandmebaasidest mitteseotud andmebaasidesse. Põhiküsimus on skeemi konstrueerimine mitterelatsioonilises andmemudelis, mis põhineb olemasoleval keerulisel relatsiooniandmemudelil. Selle uurimistöö eesmärk on pakkuda välja viisid selle probleemi lahendamiseks.

Keywords:
Andmete migratsioon, RDBMS, relatsiooniandmebaasi haldussüsteem, relatsiooniline andmemudel, mitterelatsiooniline andmemudel, NoSQL, Postgres, PostgreSQL, Cassandra, skeemi kujundamine

CERCS: P170 Arvutiteadus, arvuline analüüs, süsteemid, juhtimine

List of abbreviations and terms

RDBMS	Relational Database Management System
N-RDBMS	Non-Relational Database Management System
SQL	Structured Query Language
CQL	Cassandra Query Language
DB	Database
RF	Replication Factor
MFS	Mobile Finance System

Table of Contents

List of Figures

1. Introduction

More than a few decades ago we started to hear a term of **database**. It entered our lives with the computer technology. There were different concepts on how to store data and use it effectively, but at the end relational model was winner. There were some other models such as *network model, hierarchical model, object databases, XML databases* but none of them could compete with relational model.

Relational model started to be accepted by people in mid-80s. Name of the systems used relational model is Relational Database Management Systems (RDBMS), and the communication tool is Structured Query Language (SQL). The main reason of RDBMS being chosen by companies is its structure and the availability it provides. The best part of relational model is not making developer to think about data processing implementation. All process related to store/update/fetch data is the responsibility of the database. The other thing is the way of keeping data. As the name says the structure based on the relations between entities - tables in SQL. In relational model, there are databases, each database has schemes, each scheme has tables, and all tables have columns and rows.

Today almost all businesses use relational model to store their data as it is very useful and perfect fit to the need, as it provides relational model which requires to design structure to keep the data. Unfortunately, relational model does not let us to store this randomly generated data. Therefore we started to hear new term - NoSQL. NoSQL hashtag was misunderstood. It does not mean "No SQL", but "Not Only SQL". This non-relational model was proposing to store randomly generated data without being have to have a predefined structure. That was not the only advantage of NoSQL over RDBMS. The others are:

- better scalability on huge datasets
- to have a more qualified query operations
- to be open source (most of them) [1]

1.1 Problem Statement

With these advantages companies started to store their log/report data in NoSQL databases, then they decided not to limit themselves and to store the business data in NoSQL databases (DBs). That is easy for new companies/startups, not for the ones which have been in the market for many years. They already have their data in relational model, and their database schemas can be very complex with ten, hundreds of relational tables, and it is not an option to re-start everything with NoSQL DB. So, the best choice here is data migration from relational model to non-relational one. There are some tools out there which can be used and perform well. But the question is how to design efficient non-relational scheme based on the data from relational one. That's the question and in this research the possible answers to this question will be discussed. Researcher tries to find an answer to this question, by analyzing the existing practices. Firstly, it is necessary to understand non-relational model and databases that developed on non-relational concept. Then, we will check existing solutions for migration issues. As all companies have different kind of systems, they all follow different steps for migration. We will try to find, and analyze them. Based on these data researcher aims to have his own theory. Researcher will discuss his theory and he will put his theory in practice. Then the result of the process will be shared and discussed in thesis paper. At the end based on all these, researcher will discuss what can/should be done in future.

1.2 Thesis Structure

In the following sections we will analyze our choice of database structure which uses relational model and the other choice of ours which uses non-relational model, respectively. The following section will discuss existing migration tools, performance evaluation, advantages and disadvantages of these tools. Based on the data we collect by analyzing these tools we will be able to come up with a solution to the issue which this research is about. Then, solution of researcher will be applied to a real world case, and performance values will be discussed. The last section will be about the future work and summary.

2. Data Model

In this section data migration methodology will be discusses. Researcher will explain the concepts of RDBMS and NoSQL, then reasons of chosen databases is discussed.

2.1 Data Storing Models

As we said earlier one of the most important and tricky part of building either small or enterprise applications is to decide how to store data. Today the majority prefer relational model, while others use non-relational one. Both have some advantages and disadvantages. In this section the models and the systems which use these models are discussed.

2.1.1 Relational Model

In 1970, E. F. Codd introduced a new model named relational model. [2] The systems that use relational model are called Relational Database Management Systems (RDBMS). In relational model, systems aim to structure the data, therefore these systems provide structured environment. This environment has schemes, tables, columns and rows. Based on these properties we are able to say that relational model aims to create a relation among data. Having structured data helps us to understand business model more clear, and adapt the data to any unexpected scenario. [3]

Suppose we build a delivery system application, and we need to store user data. Basically, we need user's first name and last name, contact information such as phone number and email address and delivery address. For now, we will skip the scheme part, and will focus on table part. In order to store these user data we need to create table named **USERS**. This name is up to us, and we are able to name it as we want. As it has been told earlier, in relational models tables have columns which created during table creation with specific name and characteristics. In our case, we create columns named **firstname**, **lastname**, **email**, **phone**, **street**, **city**, **country**. Now, we have a table with seven columns but zero row - data.

Fig 1 illustrates simple users table which has 2 rows. There two users with contact and delivery information. This table has been created in Oracle. In order to come this step

	first_name character varying (255)	last_name character varying (255)	email character varying (255)	phone character varying (255)	street character varying (255)	city character varying (255)	country character varying (255)
1	John	Doe	johndoe@email.com	+123456789	Marat Amirkhanov	Baku	Azerbaijan
2	Jane	Doe	janedoe@email.com	+987654321	Ali Valiyev	Ganja	Azerbaijan

Figure 1. *Basic USERS Table*

researcher followed some steps. Researcher uses a graphical interface to communicate with database. Here is the steps:

1. Create database
2. Create table
3. Populate table

```
create table EMPLOYEES (
   empno               number,
   name                varchar2(50) not null,
   job                 varchar2(50),
   manager             number,
   hiredate            date,
   salary              number(7,2),
   commission          number(7,2),
   deptno             number,
   constraint pk_employees primary key (empno),
   constraint fk_employees_deptno foreign key (deptno)
       references DEPARTMENTS (deptno)
);
```

Figure 2. *create table query* [4]

Fig 2 illustrates query to create table in Oracle. As it is seen it is mandatory to follow some syntax rules, and use some keywords such as **create, table, varchar** and to give name to columns. As figure shows all columns are supposed to have text values and character limit is 255.

In order to populate tables we need to follow some insertion rules. There is query to insert data into a table. We need to use some keywords such as **insert, into, values** and to specify table name which is **EMPLOYEES** in our case. Once we specified these values the last step is inserting values which are data - rows of our table. It is illustrated in *Fig 3*.

```
insert into EMPLOYEES
   (name, job, salary, deptno)
   values
   ('Sam Smith','Programmer',
    5000,
  (select deptno
  from departments
  where name = 'Development'));

insert into EMPLOYEES
   (name, job, salary, deptno)
   values
   ('Mara Martin','Analyst',
   6000,
   (select deptno
   from departments
   where name = 'Finance'));

insert into EMPLOYEES
   (name, job, salary, deptno)
   values
   ('Yun Yates','Analyst',
   5500,
   (select deptno
   from departments
   where name = 'Development'));
```

Figure 3. *Populate table query* [4]

Advantages

Data is structured. If a developer needs to store users data, he/she has to create users table with some properties. The structure have to be specified. So, it is easy to understand what data is stored in which data without being have to check all rows. The other advantage is developer does not have to be ready to handle unexpected scenarios. He/she know what data will be stored in table, and what data is supposed to fetched from table. It also make to create relations between multiple tables which is called data normalization.

Disadvantages

Data is structured. It means, developer risks to lose some data. Let's assume we need to store user behaviour in our database, and in structured data model - relational model it is not possible to store all kind of data as the model is not available for it.

2.1.2 Non-Relational Model

Non-relational model is mostly used NoSQL databases. NoSQL databases concept is trend topic of last decade because of its advantages on use over RDBMS. There are three types of non-relational database models.

- Key-Value Stores
- Column Oriented Stores
- Document Based Stores

SimpleDB of Amazon is very good example for **Key-Value Store** systems. In these systems in order to store data key-value concept is used. To fetch data from db developers need to use key. In this kind of systems it is possible to store both structured and unstructured data. [5]

This type of stores are the most known non-relational stores. The data are stored as pair, and simplicity of the structure helps to manage the data in an efficient way. The key can be anything: it can be a simple text, (ie. file name, url or hash) or structured. Oracle NoSQL composite key concept is a great example to structured key. And the other part of the pair, value, is used to represent the data we aim to store, which can be with arbitrary structure, size and type. This data can be a string or a document, or storing an image is also possible. [6]

The issue here is that, as these type databases work on key-value principle, the best use case for them is performing search operations with key. Therefore, fetching data with an identification (key) parameter is highly suggested. For example, fetch an order by its id, or get user information by its username etc. Not all applications works only in this way. So, it may require to search for a data based on any parameter. Let's assume we need to get users who are younger than 30 years of age. This kind of queries are not supported by the system (in simpler versions). Therefore it is needed to perform this kind of operations in client side. On the other hands, advanced systems such as Redis provide this kind of functionalities. [6]

We can group key-value database in three types:

1. In memory: Stores data in memory, in order to provide fast access to data. This type of stores are used to store transient data.
2. Persistent: Stores data in disks: HDD/SSD. This type of stores are suggested to be used when non-transient data should be stored.

3. Hybrid: Best example is Redis. It is combination of both in memory and persistent methodology. Firstly, data is stored in memory, and it is written into disk. [6]

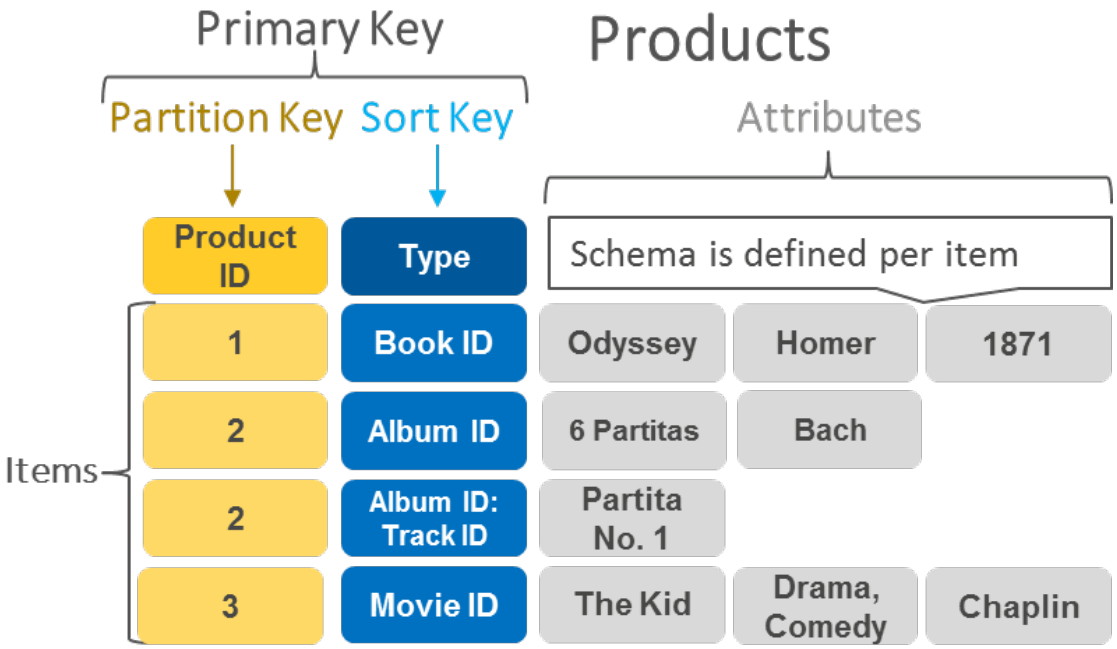

Figure 4. *Key Value Storage Architecture* [7]

4 illustrates the architecture of a simple key value database. In the figure, we can see that key part consists of two keys: partition key and sort key, while the value part is used to store products data.

We said earlier that NoSQL database is good to store unstructured data, but structured data can be stored in NoSQL databases, too. Still, NoSQL databases have advantages over RDBMS. To store structured data we can use **Column Oriented Stores**. These systems are not shcemeless, and we do not have to read other columns data but only the ones we want to. HBase and Cassandra DBs are example of this kind of stores. [5]

Bigtable of Google is the inspiration for this type of stores. In this type of databases, a table is called a column family, and a column family consists of a number of columns. At the same time, a column family is part of the schema. The schema is flexible and lets columns to be removed and added during runtime. [5]

A column has two parameters: name and value. These values can be anything, a string or a (un)structured data. Values in databases are represented as rows. This type of structure is familiar to us from key-value stores. So, we are allowed to say that column oriented databases are extended version of key-value stores. [5]

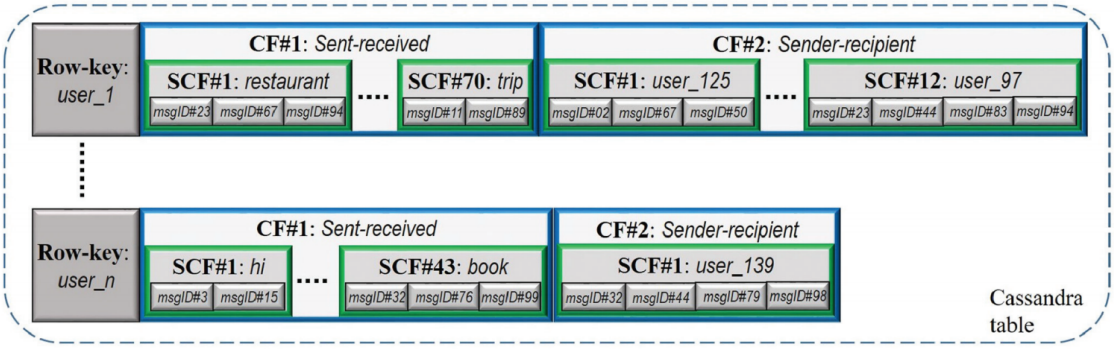

Figure 5. *Column Oriented Storage Example* [6]

As *Fig 5* shows that because of the schema flexibility it is possible column families to have different number of columns.

In **Document Bases Stores** we store data as collection of documents. These documents can be JSON or any other format, but JSON is the one which used mostly. MongoDB, Apache CouchDB can be considered as Document Bases Stores. [5]

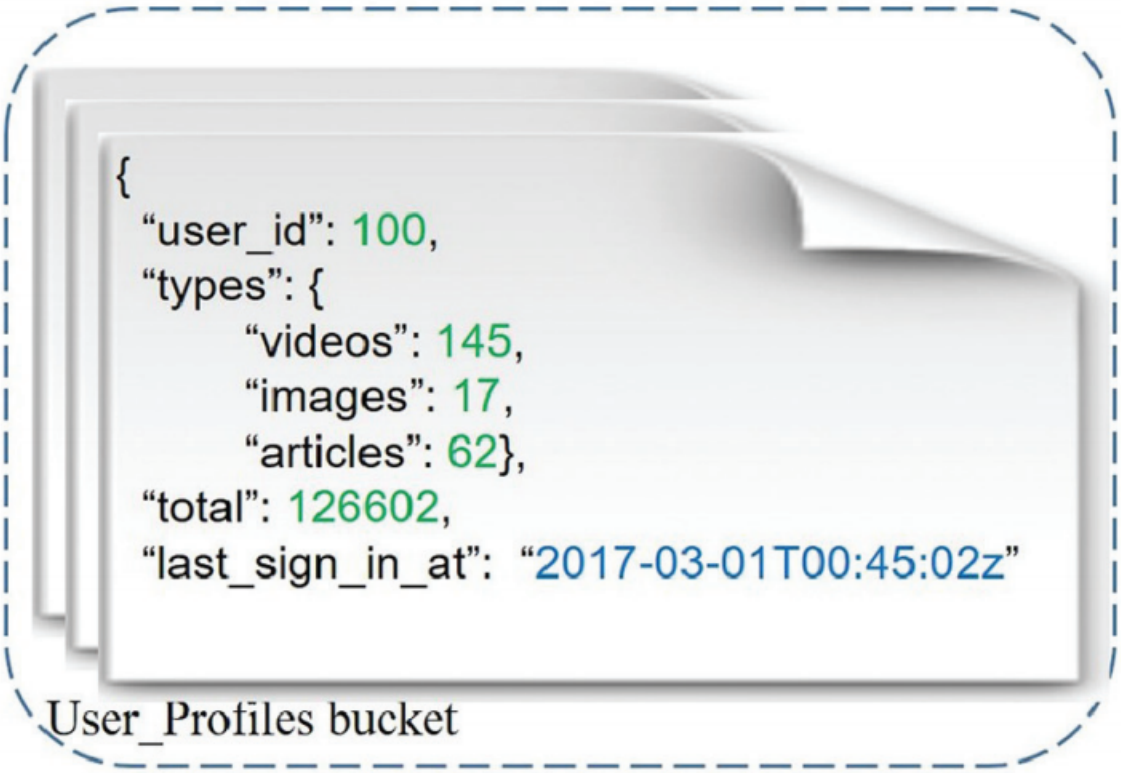

Figure 6. *Document Store Example* [6]

These type of stores can be called as extended key-value databases. It is suggested to use this type of databases when Content Management System related systems are developed. As it is illustrated in *Fig 6*, users' data from a blogging platform can be stored in document model.

	Fit scenario(s)	Strength(s)	Limitation(s)
Key-value	Objects are only accessed via a single *Key*, object caching, and where objects are not related.	Scalable, a very fast random access via *Key*, and ease of data partitioning	The responsibility of applications for the modeling of *values*, the indexing and querying of objects just by their *Keys*, and the user needs to have the key of an object in order to query it.
Wide-column	Batch-oriented parallel processing of large aggregated datasets	With regard to query workload, a hierarchy of aggregates, such as column-families, are designed that, in turn, increase the performance of queries. A suitable model for storing huge amounts of data, as it can be efficiently partitioned horizontally (by rows) and vertically (by column-families).	Limited ad-hoc querying as any change in the application-specific access patterns will impact the design to a large degree; the predefined set of column-families makes it difficult to use wide-column stores for applications with evolving schemas.
Document	Data can be easily interpreted as documents and are constantly evolving.	A rich data model to store data with arbitrary complexity, such as nested structures, arrays, and scalar values; each component of a document can be accessed via secondary indices.	There is no standard API or query languages.

Figure 7. *NoSQL Stores comparison* [6]

Fig 7 illustrates the comparison of three types of methodologies those rely on non-relational model. In the illustration it is possible to see the differences of stores in three categories: 1) fit scenario, 2) strength, 3) limitations.

CAP Theorem

It is impossible to have all these three criteria at the same time. Only two of them can be chosen at the same time. There is similar concept in NoSQL systems. It is called CAP Theorem.

- Consistency: This means the correct data should be returned for each request. It is usual to have several instances on several servers for any database, and they should provide consistency in data. [8]
- Availability: This stands for responding to all requests even if the server goes down. [8]
- Partition Tolerance:As we said earlier it is normal practice to have several servers to store data. Partition tolerant systems provides us with data even if there is a partitioning problem - communication issue between nodes. [8]

As *Fig 8* shows that, 1) consistency promises to get the most correct value, 2) availability promises to get a response for any request, 3) partition-tolerance promises to get response even if some nodes crashes.

Each database systems should choose two of these properties in order to serve users. RDBMS choose consistency and availability. NoSQL databases divided here. Some of

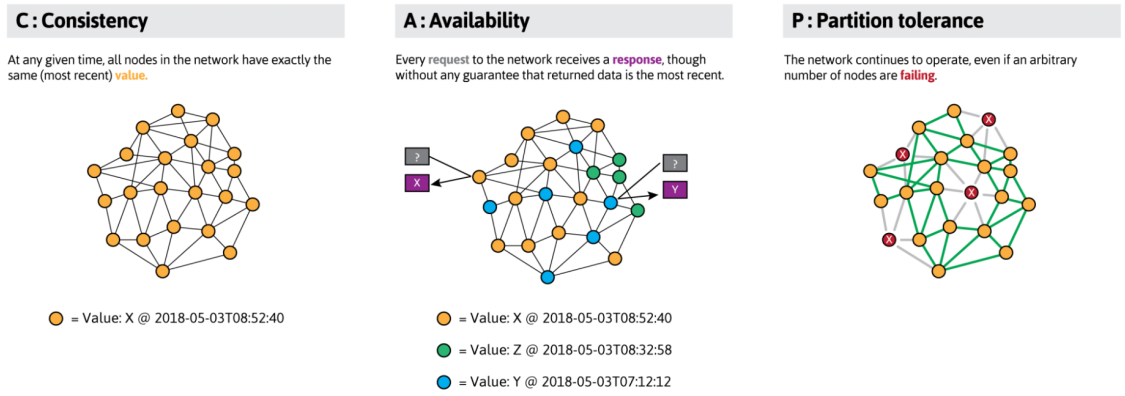

Figure 8. *CAP Theorem Parts [9]*

them choose consistency and partition tolerance, while the others prefer partition tolerance and availability.

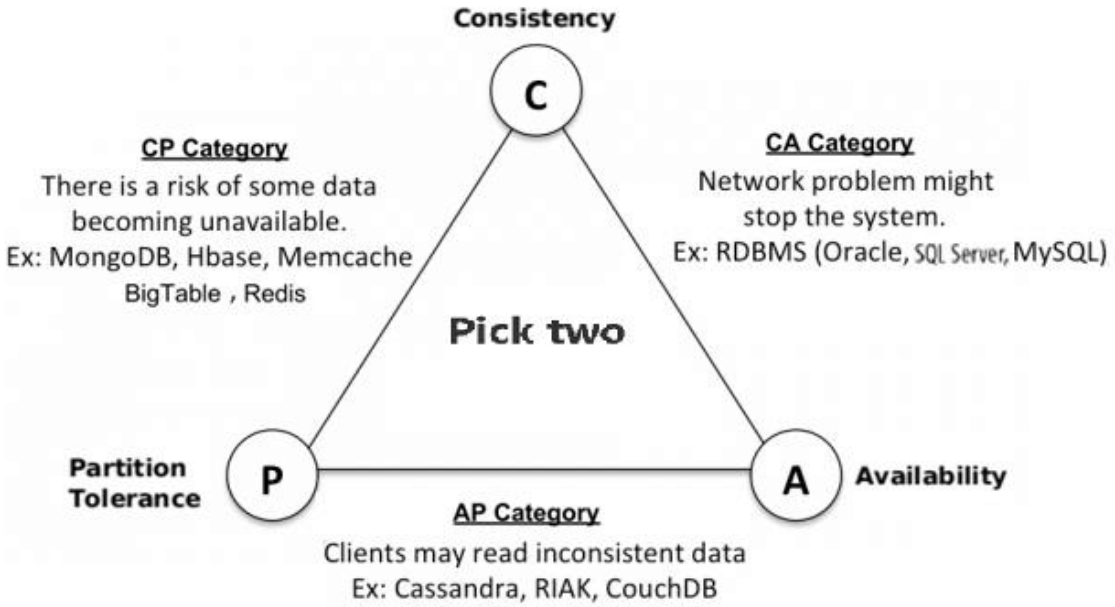

Figure 9. *CAP Theorem [10]*

RDBMS systems mostly work on a single server, and in order to get more efficiency system parameters need to be improved. It is not the case for NoSQL DBs. NoSQL DBs can be designed to work on multiple cloud servers. [5]

2.2 Databases

In order to store data, we need to use systems which apply relational model or non-relational model. These systems are databases. For this thesis, researcher had to choose what databases to use. From RDBMS, there are options like Oracle, MySQL, MSSQL, and

PostgreSQL. Oracle is our choice of database for this thesis, as it is one of the first RDBMS database in the business. Additionally, Oracle is the first database that has been developed for enterprise grid computing, according to Oracle documentation, is cost efficient and the most flexible database in the business. [11] Additionally, Oracle is the database that researcher uses in daily basis at the company he works for. In order to put the theories in practice with real data, it is necessary to use Oracle.

The other option is Cassandra. We have several options when it comes to choose NoSQL database. The trends are MongoDB, HBase etc. But, as Cassandra supports column oriented storage it is the most similar NoSQL DB to RDBMS among NoSQL DBs.

Additionally, there are some other few reasons to choose Cassandra.

- Cassandra handles large datasets easily [12]
- Cassandra is highly fault tolerant [12]
- Cassandra has a huge community [12]
- Cassandra has been developed based on Google's Bigtable and Amazons Dynamo DB. So it is a valuable to have comparison on products of huge companies.

In following subsections selected databases are discussed.

2.2.1 Why Oracle

Oracle was founded by Larry Ellison in 1977. It is the first known database that implemented relational-model in practice. [13]

SQL in Oracle

It is possible to create database in Oracle by using SQL. *Fig 10* shows the syntax for creating database in Oracle system. The syntax requires keywords such as **CREATE DATABASE** which is followed by database name. This part is mandatory part of the creating database query. The followings are optional, if they are not specified the default values are assigned by the system itself. As it seems in the figure, it is possible to set users of the database, and some other properties such as log files locations and sizes, character set, data file location and its size. [14]

Additionally, in Oracle there are two different structures:

1. Logical Structure

```
CREATE DATABASE mynewdb
   USER SYS IDENTIFIED BY sys_password
   USER SYSTEM IDENTIFIED BY system_password
   LOGFILE GROUP 1 ('/u01/app/oracle/oradata/mynewdb/redo01.log') SIZE 100M,
           GROUP 2 ('/u01/app/oracle/oradata/mynewdb/redo02.log') SIZE 100M,
           GROUP 3 ('/u01/app/oracle/oradata/mynewdb/redo03.log') SIZE 100M
   MAXLOGFILES 5
   MAXLOGMEMBERS 5
   MAXLOGHISTORY 1
   MAXDATAFILES 100
   CHARACTER SET US7ASCII
   NATIONAL CHARACTER SET AL16UTF16
   EXTENT MANAGEMENT LOCAL
   DATAFILE '/u01/app/oracle/oradata/mynewdb/system01.dbf' SIZE 325M REUSE
   SYSAUX DATAFILE '/u01/app/oracle/oradata/mynewdb/sysaux01.dbf' SIZE 325M REUSE
   DEFAULT TABLESPACE users
      DATAFILE '/u01/app/oracle/oradata/mynewdb/users01.dbf'
      SIZE 500M REUSE AUTOEXTEND ON MAXSIZE UNLIMITED
   DEFAULT TEMPORARY TABLESPACE tempts1
      TEMPFILE '/u01/app/oracle/oradata/mynewdb/temp01.dbf'
      SIZE 20M REUSE
   UNDO TABLESPACE undotbs
      DATAFILE '/u01/app/oracle/oradata/mynewdb/undotbs01.dbf'
      SIZE 200M REUSE AUTOEXTEND ON MAXSIZE UNLIMITED;
```

Figure 10. *Oracle create database query.* [14]

2. Physical Structure

Logical structure is for us to design our tablespaces, schemas and tables. If we have multiple APIs and it is mandatory to configure APIs only connect to the database they should, it is possible to handle this by logical structure. We just create, two different tablespaces or schemas (it is up to us), and assign each to each API. [14]

Physical structure is hardware structure. How we keep our datacenters, servers are topics of physical structure discussion. [14]

```
    CREATE TABLE schema_name.table_name (
        column_1 data_type column_constraint,
        column_2 data_type column_constraint,
        ...
        table_constraint
);
```

The query above illustrates the syntax to create table in Oracle system. It is mandatory to

specify **table name, column name and column data types**, and optional to specify constraints (**UNIQUE, NOT NULL, PRIMARY KEY, REFERENCES**) for each column. It is up to us to specify scheme name, but it is suggested to use it inside creation query [15]

The first example below is a query structure to insert a record into a table in Oracle. The following is the syntax to add a new row. As it illustrates, it is possible to populate a table by selecting rows from another table. The other way is specifying column names and values for each column respectively. If we are about to add a row and we have value for each column, then we are obligated to specify column names. [16]

```
insert_statement ::=
INSERT_INTO {table_reference | [THE] (subquery1)}
    [(column_name [, column_name]...)]
    {VALUES (sql_expression [, sql_expression]...)
    | subquery2} [RETURNING]
    [row_expression [, row_expression]... INTO
    {variable_name | :host_variable_name}
    [, {variable_name | :host_variable_name}]...];


INSERT INTO table1
    SELECT t2.column1, t2.column2 FROM table2 t2;

INSERT INTO table1 (column1, column2, column3, column4)
    VALUES (value1, value2, value3, value4);

INSERT INTO table1
    VALUES (value1, value2, value3, value4);
```

2.2.2 Why Cassandra

Cassandra is our choice of NoSQL database for this thesis. It is fully distributed, fault tolerance and persistent database. Cassandra database contains properties of both Google's Bigtable and Dynamo in order to provide distributed database.

Main Features of Cassandra

- Decentralized: As there are several cluster with same role, there is no single point of failure. There is no master cluster therefore data is distributed among clusters.

- Replication and Multi Data Center Replication Support: There strategies for replication. As Cassandra is decentralized system strategies for deployment is important.
- Scalability: System should be scalable. Read and write operations can be handles easily with no downtime.
- Fault Tolerance: System is distributed, and issue of a single node does not prevent to get/add data.
- MapReduce Support: Cassandra has Hadoop integration, therefore it provides MapReduce support.
- Query Language: Query language is called Cassandra Query Language - CQL. [17]

Data Model of Cassandra

Key concepts of Cassandra database:

- **ColumnFamilies**: sets of Columns
- **Keyspace**: a namespace for ColumnFamilies
- **SuperColumns**: columns which contain Columns
- **Columns**: specified with name, value and timestamp

It is usual practice that each Cassandra cluster to have one KeySpace and KeySpaces have ColumnFamilies. ColumnFamilies can be seen as tables of RDBMS. Visually it can be similar to RDBMS, but in the background working principles are different. As an example new data is added as row in Cassandra, but the difference is columns for each row do not have to be same. There is unique key for each row, and a row can contain multiple ColumnsFamilies. [5]

Cassandra Query Language - CQL

In order to communicate with Cassandra database we need to use Cassandra Query Language - CQL. *Fig 11* shows the query for creating keyspace. The syntax is defined. Required keywords are used, and keyspace name is specified. It is possible to specify some options, if options are not defined default ones used. [18]

Second part of the *Fig 11* shows the example of creating keyspace. A keyspace named **excelsior** is created with options of replication values - class is **SimpleStrategy** and replication factor is 3.

As we said earlier ColumnFamilies can be considired as tables of RDBMS. *Fig 12* illustrates the syntax of creating tables in Cassandra database. The statement has **create table statement**, **column definition**, **primary key**, **partition key**, **clustering colums**, **table**

A keyspace is created using a `CREATE KEYSPACE` statement:

```
create_keyspace_statement ::=  CREATE KEYSPACE [ IF NOT EXISTS ] keyspace_name WITH options
```

For instance:

```
CREATE KEYSPACE excelsior
    WITH replication = {'class': 'SimpleStrategy', 'replication_factor' : 3};

CREATE KEYSPACE excalibur
    WITH replication = {'class': 'NetworkTopologyStrategy', 'DC1' : 1, 'DC2' : 3}
    AND durable_writes = false;
```

Figure 11. *create keyspace query.* [18]

options, **clustering order**. These properties can be specified in the process of table creation. Second part of the figure illustrates some example for creating tables. Keywords are **CREATE, TABLE** and column data types. The other values are tables names and columns names. [18]

Fig 13 illustrates three steps of querying a table. Creating table, populating table and getting data. Firstly, table **t** is created with columns **pk, t, v, s** and **pk** and **t** are columns used for creating **primary key**. Then we use CQL for populating the table. There are two lines of insertion, and the last step is getting data from table.

Creating a new table uses the `CREATE TABLE` statement:

```
create_table_statement ::=  CREATE TABLE [ IF NOT EXISTS ] table_name
                            '('
                                column_definition
                                ( ',' column_definition )*
                                [ ',' PRIMARY KEY '(' primary_key ')' ]
                            ')' [ WITH table_options ]
column_definition      ::=  column_name cql_type [ STATIC ] [ PRIMARY KEY]
primary_key            ::=  partition_key [ ',' clustering_columns ]
partition_key          ::=  column_name
                            | '(' column_name ( ',' column_name )* ')'
clustering_columns     ::=  column_name ( ',' column_name )*
table_options          ::=  COMPACT STORAGE [ AND table_options ]
                            | CLUSTERING ORDER BY '(' clustering_order ')' [ AND table_options ]
                            | options
clustering_order       ::=  column_name (ASC | DESC) ( ',' column_name (ASC | DESC) )*
```

For instance:

```
CREATE TABLE monkeySpecies (
    species text PRIMARY KEY,
    common_name text,
    population varint,
    average_size int
) WITH comment='Important biological records';

CREATE TABLE timeline (
    userid uuid,
    posted_month int,
    posted_time uuid,
    body text,
    posted_by text,
    PRIMARY KEY (userid, posted_month, posted_time)
) WITH compaction = { 'class' : 'LeveledCompactionStrategy' };

CREATE TABLE loads (
    machine inet,
    cpu int,
    mtime timeuuid,
    load float,
    PRIMARY KEY ((machine, cpu), mtime)
) WITH CLUSTERING ORDER BY (mtime DESC);
```

Figure 12. *create cassandra table query*. [18]

```
CREATE TABLE t (
    pk int,
    t int,
    v text,
    s text static,
    PRIMARY KEY (pk, t)
);

INSERT INTO t (pk, t, v, s) VALUES (0, 0, 'val0', 'static0');
INSERT INTO t (pk, t, v, s) VALUES (0, 1, 'val1', 'static1');

SELECT * FROM t;
 pk | t | v      | s
----+---+--------+-----------
 0  | 0 | 'val0' | 'static1'
 0  | 1 | 'val1' | 'static1'
```

Figure 13. *create, insert and select query in cassandra* [18]

3. Apache Cassandra

Usually, startups start with Monolith architecture and simple database design. In the beginning phase these are enough to handle the project as it is small. Small here stands for the application code is not that huge, database structure is simple, and request rate into the system is not very high. The issue starts when the system starts to get bigger and bigger. Sometimes, engineers decide to change architecture from monolith to microservices as it provides more solutions to the problems, and sometimes they decide to make some changes in database. There are few solutions here. It might be necessary to add new servers to handle big data, or to change the design of schemata to handle the issues. The other option is to change the database itself entirely, and to migrate data into the one which can handle data better, and provide better performance.

In order to migrate data into new database with minimum error ratio, it is necessary to understand the new database structure well. It is not easy task to do. First thing is to decide new database structure - how to organize data, because each system has different priorities and therefore they provide services in different ways. As we aim to design NoSQL DB structure to migrate our data from RDBMS DB, we will focus on Cassandra database structure firstly. In this chapter we will talk about specific details of Cassandra database which we think should be known before designing the structure and migrating the data.

3.1 Cassandra in Details

We know that RDBMS DB is based on table-oriented concept. It is necessary to design tables in a way to get results in minimum time. Unlike RDBMS, Cassandra focuses on query-oriented methodology. It is suggested to think about the queries that system is going to run to get data before starting to create schemas and tables.

3.1.1 Architecture

Cassandra architecture has been designed by the concepts of Facebook's Dynamo and Google's BigTable. As we already mentioned that Cassandra is query-based database, it does not support relations between entities. In RDBMS DBs we create relations between ta-

bles to normalize data and not to repeat data. Cassandra does not see data de-normalization as an issue. As it has been designed to handle huge amount of data this should not be a problem anyway. [19]

Partitioning

In order to provide efficient performance, Cassandra partitions all data, and stores each partition in multiple nodes. It is called replication. All transactions (inserts, deletes, updates) are recorded by timestamp, and the latest transaction is taken into consider during query operation.

The important part here is to understand how Cassandra performs partition operation. It uses consistent hashing algorithm. By this algorithm Cassandra generate token(s) and decide which node(s) will be responsible to store these tokens. [19]

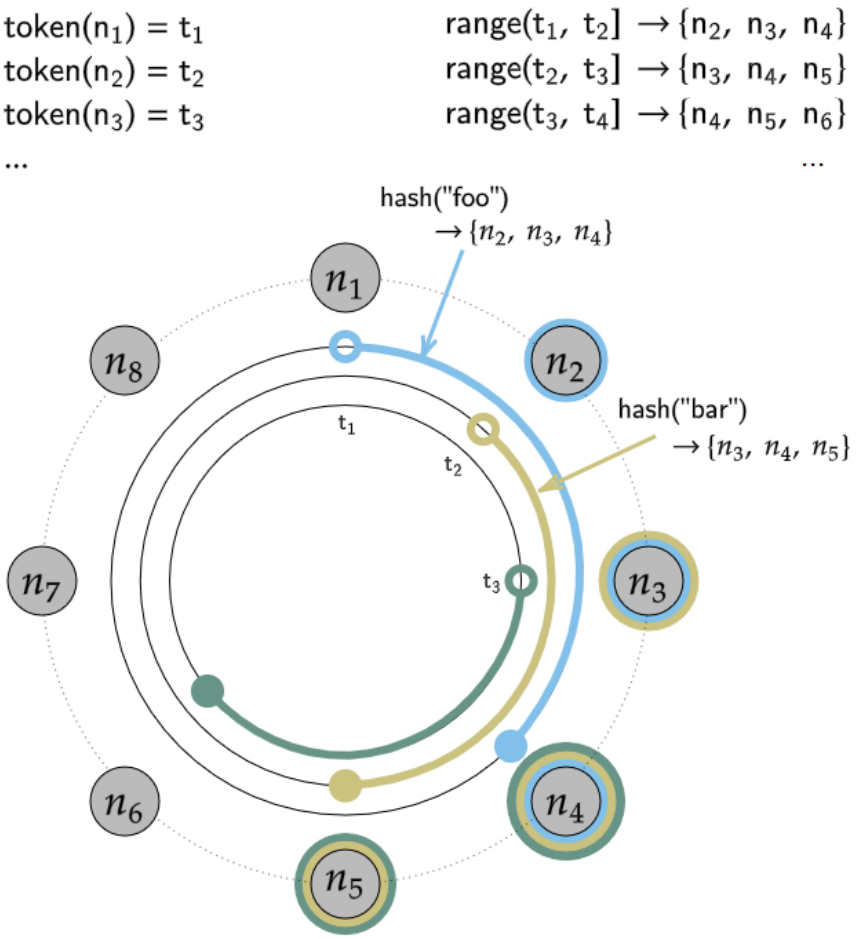

Figure 14. *Cassandra Ring* [19]

Fig 14 illustrates ring of nodes to store partition tokens. As you see, token 1 and 2 are store in node 2, node 3 and node 4. This process goes in one direction - clockwise. This structure works fine, unless we need to increment physical node as it will cause imbalance.

To solve this problem "virtual nodes" concept has been suggested. The solution is to let a single physical node hold multiple partitions. In fact, we do not add any physical nodes, we just add virtual ones and it makes clusters look big enough to store partitions. *Fig 15* shows that there are 8 nodes even though we have 4 physical nodes. [19]

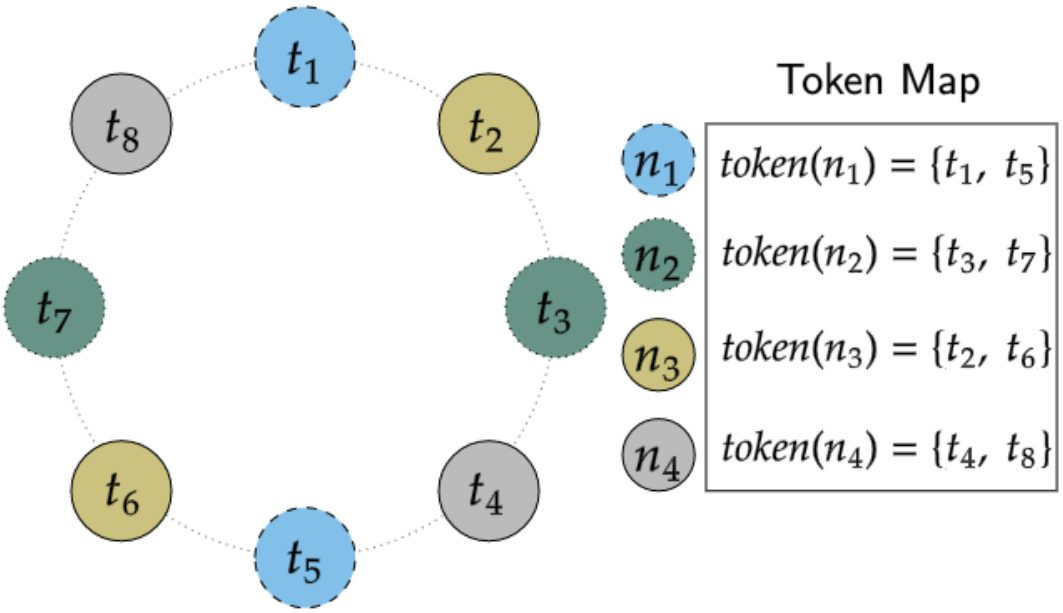

Figure 15. *Cassandra Virtual Nodes* [19]

3.1.2 Data Versioning and Consistency

In previous section we mentioned that multiple copies of data stored in different clusters in order to prevent data loss in case of network failure. In following section we will discuss the strategies of storing data replications. Cassandra has an understanding which called Replication Factor (RF). This term is used to specify how many partition replication will exist. [19]

Replication Strategy

Cassandra provides two different replication strategy.

- Simple Strategy: Lets us define a replication factor. This is the number of nodes that will contain a replica of each row [19]. Using this strategy is not suggested in production [18].
- Network Topology Strategy: This is the strategy which is suggested to use in production environment, even we have one cluster. Because this strategy allows us to add new physical or virtual datacenters later [19]. Besides that Network Topology Strategy lets us define replication factor for each data center independently. [18]

As Cassandra stores data in multiple datacenters, it is possible that some servers to have newer data than others. Cassandra knows that by timestamp value of each row. This timestamp value can be specified either by client or node itself. The server with latest timestamp value count as the owner of the newer data. To prevent this inconsistency Cassandra applies different techniques. [19]

1. Replica read repair <read-repair>: synchronization happens during read operation
2. Hinted handoff <hints>: synchronization happens during write operation
3. Anti-entropy repair <repair>: synchronization happens by comparing data of replicas [19]

The question here is when do we know our read operation is successful? Data is stored in multiple servers, so Cassandra provides a term names consistency level. This term let the system consider read operation successful when specified number of clusters respond. There are different levels. [19]

- ONE: a single replica response is enough
- TWO: two replica responses are enough
- THREE: three replica responses are enough
- QUORUM: majority of replicas response is enough. majority = (n / 2 + 1)
- ALL: all replicas must respond
- LOCAL QUORUM: majority of replicas in local data center is enough
- EACH QUORUM: a majority of replicas in each data center is enough
- LOCAL ONE: a single replica response is enough. this guarantees read requests are not sent to remote data center replicas. in multi data center cluster.
- ANY: single replica response is enough [19]

3.2 Data Definition

Cassandra has similar understanding to RDBMS to store data. In Cassandra data is stored in a *table* which located in a *keyspace*. *Fig 16* shows the concept which used to store data. We are able to have Keypsaces. Keyspaces are logical storage. It is used to store tables, and it can have its own specific properties such as strategy or replication factor. Inside keyspaces we have to have tables. Tables use rows and columns to form data and to store it. As the *Fig 16* illustrates we are able to create any amount of tables inside a keyspace, and a table can have any number of columns and rows. [18]

In the previous chapter we have discussed CQL queries to create keyspace and table, and

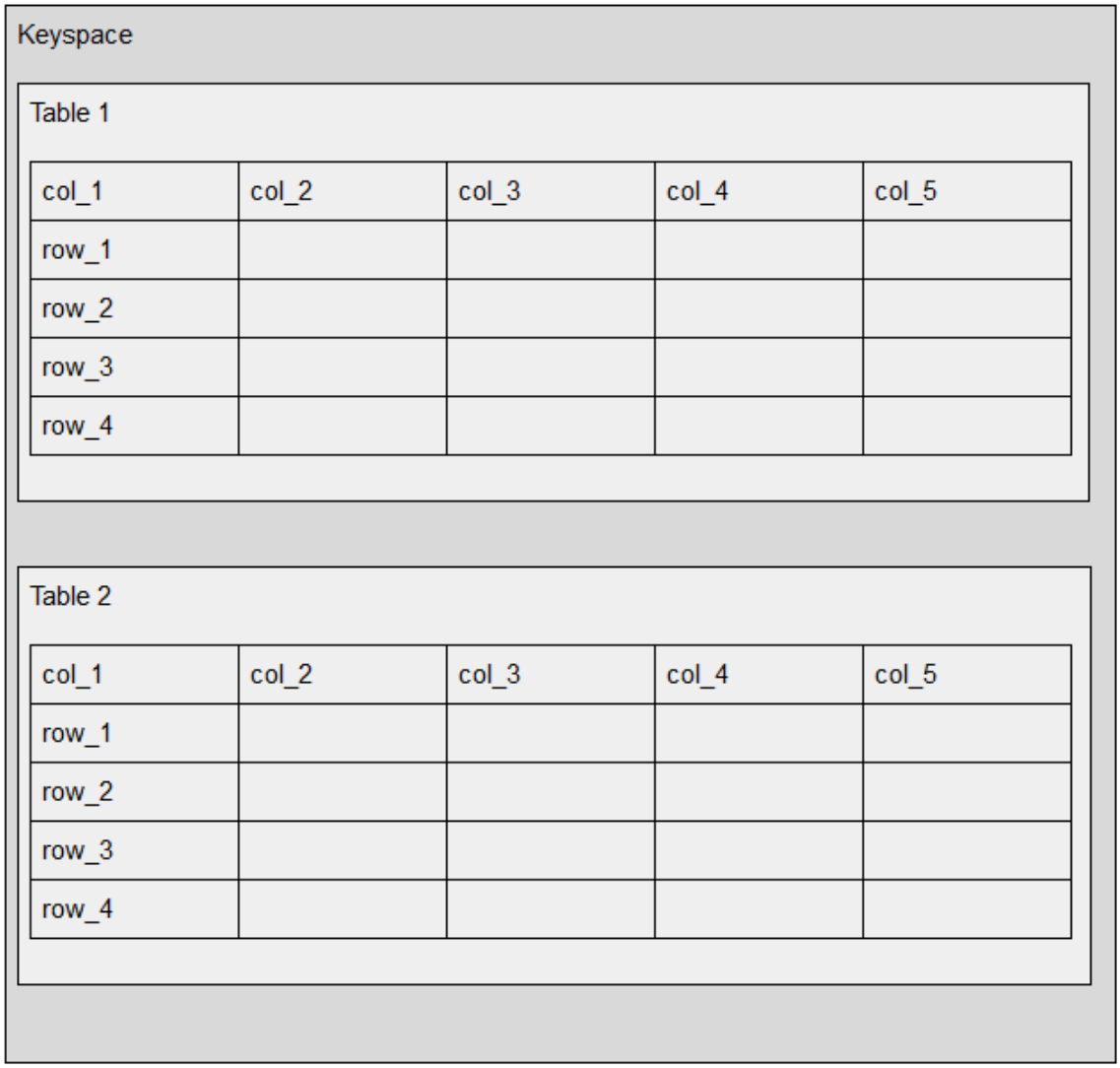

Figure 16. *Cassandra Keyspace and Table* [18]

to insert and to fetch data into/from table with figures. Therefore in this chapter we see no reason to discuss them again.

It is possible to alter keyspace or table, or completely delete them. Cassandra provides us queries for that. Here is the query example to update keyspace properties. [18]

```
ALTER KEYSPACE <keyspace_name> WITH <options>

ALTER KEYSPACE Authentication WITH replication =
    {
        'class': 'NetworkTopologyStrategy',
        'replication_factor' : 2
    };
```

And this example is used to drop the keyspace. [18]

```
DROP KEYSPACE [ IF EXISTS ] <keyspace_name>
DROP KEYSPACE Authentication;
```

The queries to alter and drop a table are these. [18]

```
ALTER TABLE <table_name> <alter_table_instruction>
ALTER TABLE users ADD username varchar;

DROP TABLE [ IF EXISTS ] <table_name>
TRUNCATE [ TABLE ] <table_name>
```

Materialized View

Cassandra, like RDBMS, provides views and these views called *Materialized View*. Like table it used to store data, but these data fetched from existing tables. The only difference is that. We can assume Materialized Views as virtual tables. Like table, we hame similar queries to create, alter and drop these views.

```
CREATE MATERIALIZED VIEW [ IF NOT EXISTS ] view_name AS
              select_statement
              PRIMARY KEY '(' primary_key ')'
              WITH table_options
```

As the statement shows, we use select from table query to populate the view and a view is populated during creation process.

```
CREATE MATERIALIZED VIEW users_view AS
    SELECT * FROM users
    WHERE user_id IS NOT NULL AND username IS NOT NULL
    PRIMARY KEY (user_id, username);
```

This is a simple query example to create a view. We just duplicate not null data from table into the view.

```
ALTER MATERIALIZED VIEW <view_name> WITH <table_options>
DROP MATERIALIZED VIEW [ IF EXISTS ] <view_name>;
```

These queries are used to alter and drop view, respectively. [18]

3.3 Data Modelling in Cassandra

During design process of a database structure of the project, engineers try to avoid data denormalization if they decide to use a RDBMS database. So, they build a structure where there tables for almost each entity and joins to represent the relations between these tables. This is called table oriented modelling. Cassandra does not support joins, so there is no way having relations between tables. There are two ways of solving this problem. First one is to handle this "relations" in client side - in code. This is not suggested. One of the reason is this way does not suit the Cassandra logic. The other reason is handling this in code requires more attention which might not be necessary in the second option. The other option which is suggested to apply is defining the queries that application is going to use during work-time. This will help us to design our structure even before starting to write code. Each query is unique case, and we do not care about normalization. Same data can be stored in multiple tables. Because Cassandra is not table oriented, it is **query-oriented**. Because by Cassandra we believe that to have efficiently designed queries will work for us well enough, there is no need to worry about data normalization at all. [20]

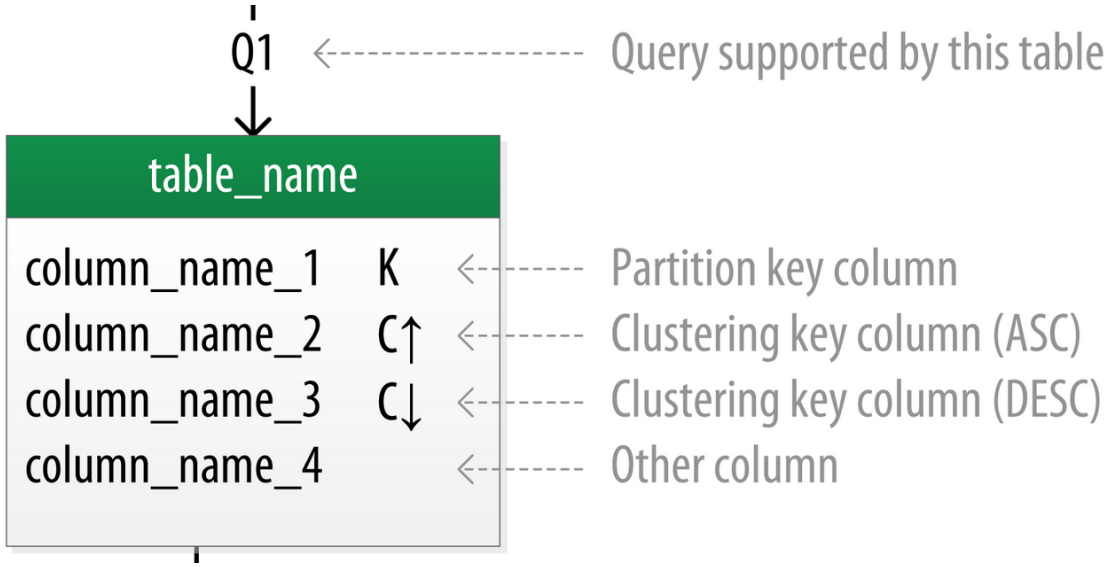

Figure 17. *Cassandra Keyspace and Table* [20]

Fig 17 illustrates query based table. This table is created to run specified query. Table has name and columns which defined during creation process. Each query case is unique, and has case based priorities. Based on these priorities, during query design it necessary to

decide which columns will be used to fetch data. Partition key column is used to decide which column will be used to partition data of table. Clustering key columns (can be single column or multiple columns) are used for the ordering process. [20]

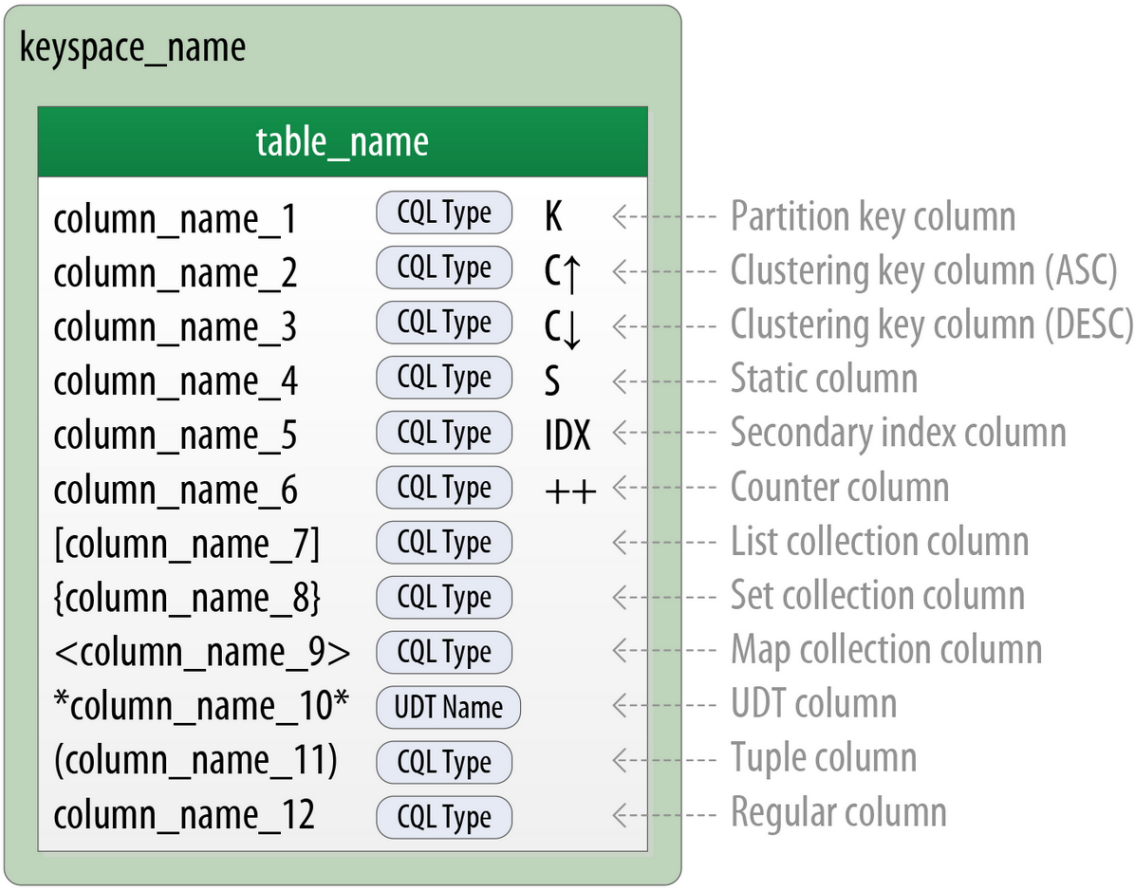

Figure 18. *Cassandra Table Column Types* [20]

Fig 18 shows different column types that we can use during table creation operation. It is possible to store list, set, map or tuple in Cassandra table. There is *static column* which can be discussed. Static column is used to set a value to all rows of a partition. [20]

```
CREATE TABLE t (k text, s text STATIC, i int,
                    PRIMARY KEY (k, i));
INSERT INTO t (k, s, i) VALUES ('k', 's1', 0);
INSERT INTO t (k, s, i) VALUES ('k', 's2', 1);
SELECT * FROM t;

k   |   s    | i
--------------
k   | "s2" | 0
k   | "s2" | 1
```

This query proves that second insertion row updates static column value for all rows. And column *k* is used for partitioning, and column *i* is used for clustering.

4. Data Migration

New technologies are being developed every day to fix bugs of existing technologies or provide services that existing systems do not support. So, it causes companies to decide what to do. It is important to know advantages and disadvantages of moving everything to new technology, therefore the are lots of researches done which compares old and new systems by analyzing them. In this chapter we will discuss the researches that focused on data migration from RDBMS to NoSQL database. We are not going to analyze only the researches that discuss data migration from PostgreSQL to Cassandra. It is better to analyze the data migration results in general.

4.1 From RDBMS to NoSQL

Bouamama [21] has done a research to migrate data from MySQL to Cassandra and HBase, and evaluated performances. The researches used Java programming language to automate data migrration.

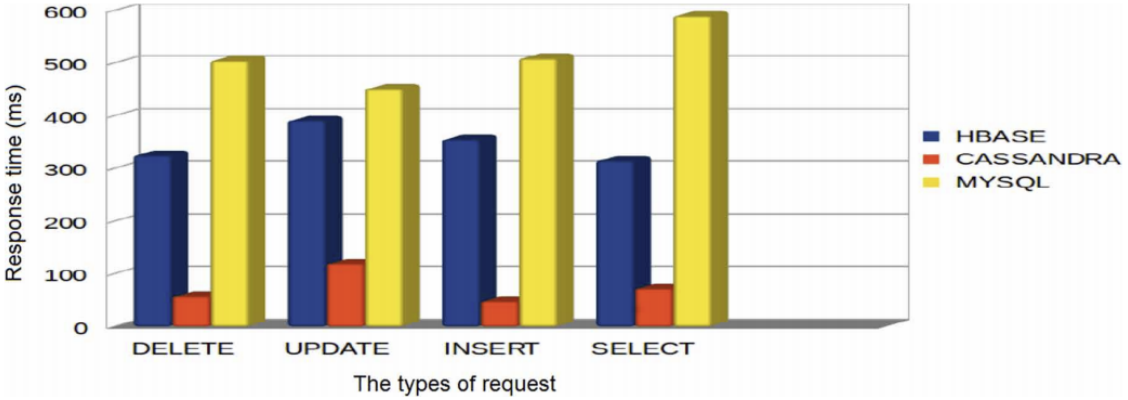

Figure 19. *Performance comparison of MySQL, Cassandra and HBase* [21]

Fig 19 is a graph that illustrates comparison of **delete, update, insert** and **select** queries in RDBMS and NoSQL. Graph shows that MySQL has worst performance. Its response time for **select** query is almost 600 milliseconds, while HBase requires half of this. Cassandra performs best here. For Cassandra this query lasts less than 100 milliseconds. Cassandra shows best performances overall. It needs a little bit more than 100 milliseconds to **update**, and less than approximately less than 50 milliseconds to **insert** and **delete** a record. HBase performs worse than Cassandra, but it still is better than MySQL. To **delete**

HBase needs 300 milliseconds, while MySQL needs 500 milliseconds. The time difference is less than 50 milliseconds in only **update** query in HBase and MySQL. Researcher informed that they took data from Facebook database which contains 1.5 million records. [21]

We assume they did not make any designing process, as the research paper does not contain any information about schema design in NoSQL database. Even in this case, NoSQL database, especially Cassandra shows better performance than RDBMS.

The other research we analyze is focusing on data migration from MySQL to MongoDB. The main reason of us analyzing this research is that researchers had to design the structure in MongoDB before migration. Because MongoDB is not column-oriented NoSQL database, and it is impossible to have same structure as MySQL.

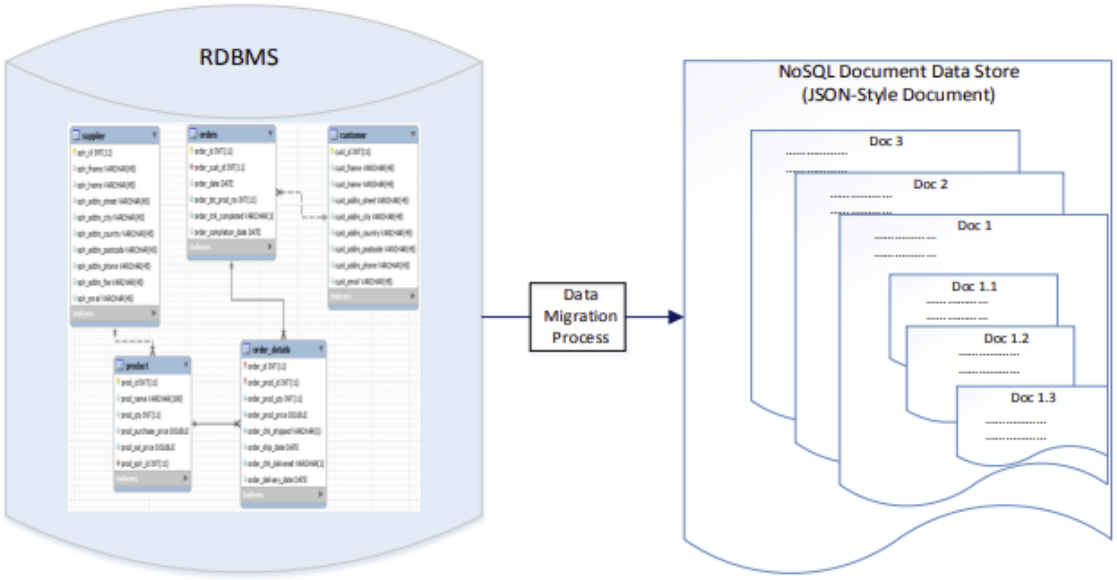

Figure 20. *MySQL and MongoDB Structure* [22]

Fig 20 illustrates the main difference between RDBMS (MySQL) and Document based NoSQL (MongoDB). RDBMS has tables and relations between them, but this is not possible in MongoDB. In MongoDB we have documents, but we are not able to have relations between them. Therefore, in order to migrate data we need to design the schema design in MongoDB. One of the solution is having documents in documents.

Feroz Alam [22] defines 5 steps of data migration.

1. Analyze the structure in RDBMS, and decide how to convert joins
2. Design structure in MongoDB
3. Design entity structure in code (bases on step 2)
4. Create entities in code (based on step 3)

5. Write code to migrate data [22]

The evaluation part of the paper focuses on MySQL and MongDB comparison as it should be, and therefore we do not discuss that here. It is more important for us to focus on the non-relational data model design part of the article and it is more suitable for our thesis.

4.2 Our Proposal

We discussed Cassandra database, and some other researches that focused on data migration. Based on these information our proposal has been developed. Firsty, it is clear that NoSQL databases performs better than RDBMS based databases in most cases. The difference is shown when data is huge. To handle small amount of data, and simple structure RDBMS is fine as well as NoSQL. The decision is up to the engineer.

The author agrees with Mr. Alam on these steps. The steps can be optimized and can be 3 steps instead of 5. Step 3 and 4 seems unnecessary as they do not require effort to decide how to complete the process. In ongoing project, to convert database structure is not an easy decision to make. We offer three steps, and our offer is similar to Mr. Alam's. Researcher will give some real world examples while proposing the steps. These examples are from the project he is currently working on, at Azercell. The project researcher is working on is called Mobile Finance System (MFS), and this topic will be discussed in details on following chapters.

1. Analyze existing structure - decide the advantages and most importantly disadvantages of having current schema design. It is not easy to have this information in not-started project. During the development process, unplanned/unexpected scenarios might happen and these have some consequences. Therefore, having a list of advantages and disadvantages of current schema design is important and it will help us to design schema in NoSQL DB.

   In database a scheme consists of tables, and tables consist of rows and columns. Additionally, in relational database, relations between tables are very important. It is mostly used to normalize data which means prevent data repetition. So, when the task is to analyze advantages and disadvantages of the structure all these components should be considered. The details and importance of all columns in tables, and tables in a scheme should be checked and noted. Analyzer can check column types, and note if correct type has been chosen to represent data of that column. Then, he/she can check all columns of table to see if table has unnecessary columns (the ones which are not used in any query), or to normalize or to denormalize that table would

have much more effective result in running time. The other thing would be to check relations of tables to see whether which of them are necessary and which are not, and which of them should be improved. To check run time of SQL queries is the other thing to check. If running an SQL query takes so long time, it is important to understand what causes this, and this issue can be solved. Is it enough adding index to some columns enough, or scheme should be updated, or SQL query should be improved? And one of last things to check would be names of schemes, tables and columns. Names should let everyone understand what this thing stores.

2. Based on information from step 1 and future plans, design the structure in NoSQL (Cassandra). As we discussed earlier Cassandra is query-oriented database, and to have RDBMS mindset is not helping here. In RDBMS we usually try to normalize our data, but in Cassandra we do not care about it, and having a different mindset might be helping. So, in Cassandra we do not design our schema, we decide what queries we have, and we are going to have in future. Based on that information, we design our query design.

   So, now as we analyzed our existing database structure we know advantages and disadvantages of it. First thing to try to keep having advantages, and improve thing which cause those disadvantages. Then, the first thing is to have list of all select queries. Please, consider that these queries should be analyzed very well, and should be updated if analyzer thinks adding/removing some queries can cause better performance. Then, it is important to note that which columns are necessary to run these queries. This is for understand how many different scenarios exist. If there are seven select queries, it does not mean we have seven different scenarios, and we should create seven tables in Cassandra. If two select query can be run with same columns in **where** clause, so one table in Cassandra can be used for both of them. Once we define how many different scenarios exist, we can create tables for each scenario. The other thing to consider is what table stores what type of data. As we know Cassandra has no problem with having denormalized tables, so having two tables in Oracle for one purpose does not mean that we must do exactly same thing in Cassandra.
3. Having Design entities in code. We assumed this step as unnecessary in Mr. Alam's 5 steps of data migration. Because it is for MongoDB, and it does not require any dedication to write code for creating entities which have exactly same structure as documents in MongoDB. In Cassandra it is different as Cassandra is query-oriented. It is important to decide entity structure. Should we to create an entity for each query, or should we organize some queries and create an entity for each group of queries. Here, group of queries is queries in keyspaces. This decision is up to the engineer. It has nothing to do in database, and it has no effect

on database performance. It is on client side, and it effects code running performance.

It can be different to write code for database operation in each programming language/framework. But, the structure would be same for all of them. We can just copy scheme structure in Cassandra, and apply it to code base. It is very important to have correct mapping, and to chose correct data types. As Cassandra does not provide JOINS, so it is very important to be careful during coding. The checks relational database is responsible for should be done by developer on code base.

The author is working in a company named Azercell as a software engineer, and in order to put the theories in practice he decided to use the data from the project he is currently working on. The next chapter will discuss this process.

5. Scheme Structure Design

Azercell is a mobile operator company in Azerbaijan and was founded in 1996. Even though it has been in Azerbaijan market for more than 20 years, Azercell is a Turkish company. Azercell is the biggest mobile operator company in the country with more than 5 million customers. [23]

Azercell has 49 percent of the market share, and the company provides services such as advance payment system, mobile internet system, front offices, service of call center, 4G technology, "ASAN IMZA" known as mobile e-signature service etc. [23]

5.1 Mobile Payment System

Researcher joined the company as a Software Engineer, and work with back-end systems mostly. The project researcher joined is a mobile payment system. Azercell wanted to provide customers a system which can be used to pay bills and fines/tickets. We called this project Mobile Finance System - MFS. In the project we use Spring Boot known as a Java framework, and to store data we use Oracle database.

In this chapter, we will put the theories in practice. Firstly, we will analyze the existing database structure, then the scheme design in Cassandra. After all these, the data migration implementation and performance values will be discussed and compared.

5.2 Database Structure

In this section researcher explains the database structure which is used to store MFS data, in Azercell. As *Fig 22* shows we created four (4) different tables.

1. **REQUESTS**: this table is used to store request data. This data includes payment amount, commission amount (if payment has a commission), company info (to know of which company invoice is paid). Additionally to all these, we also divided the requests into some types as not all requests are to make payment.
2. **REQUEST HISTORY**: As we implemented microservice architecture, it was necessary to have integration between API in order to complete some steps such as

payment. So, payment transaction has several steps, and we keep data of these steps in history table. History table also has a column which points REQUESTS table, and that is how we get history of a request easily.

3. **REQUEST PARAMETERS**: As all requests have some parameters (these parameters all are different for each company, and each endpoint), we created parameters table to store these parameters. Like history table, we created a column which points the requests table, too.
4. **COMPANIES**: This table is only used to get company details if needed. As each request (especially payment requests) might be related to a company, we created a column in **REQUESTS** table which points this table.

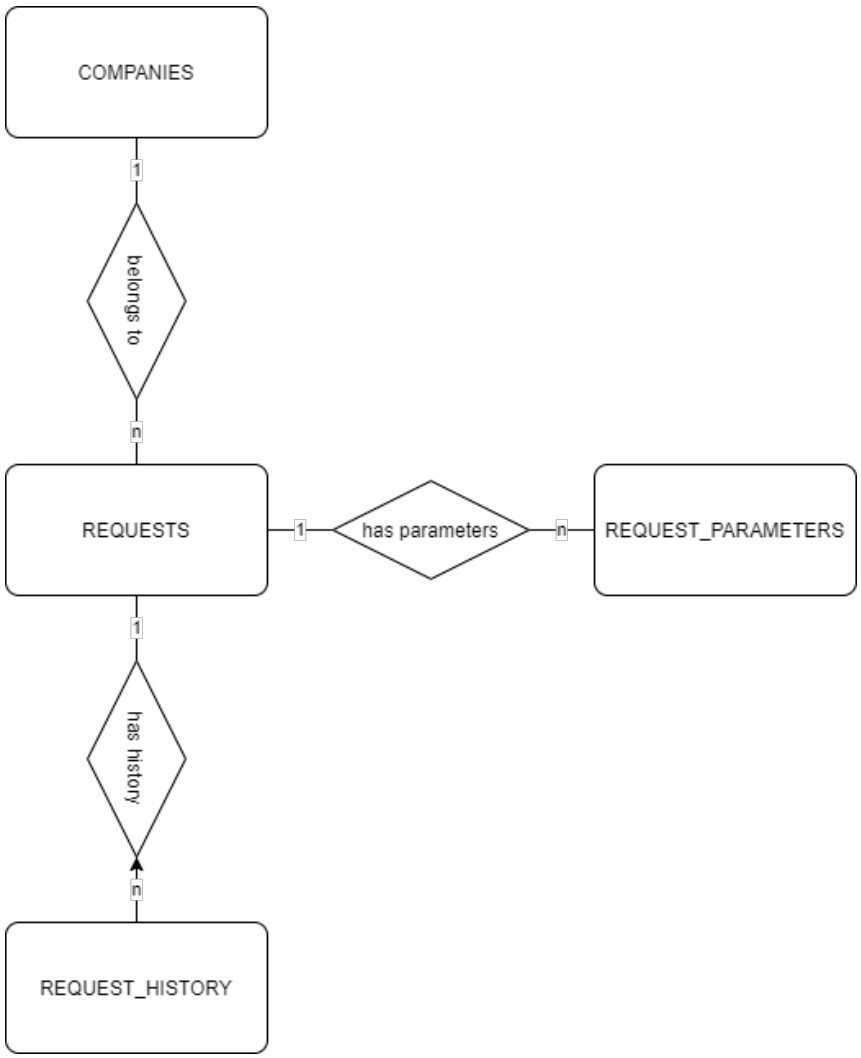

Figure 21. *Current relations between MFS tables in Oracle*

Besided all these, we also created a view and named it **REQUEST HISTORY LAST VIEW**. This view was created based on **REQUEST HISTORY** table to get latest record of each request.

Relations among tables

As you can see on *Fig 22*, there are **ONE TO MANY** relation between all tables. As each request can have multiple requests and history info, it was necessary to create **ONE TO MANY** relationship between **REQUESTS** and other two. This is a little bit different for **COMPANIES** table. One or more than requests can be related to only one company. So, we created **ONE TO MANY** relationship between **COMPANIES** table **REQUESTS** table.

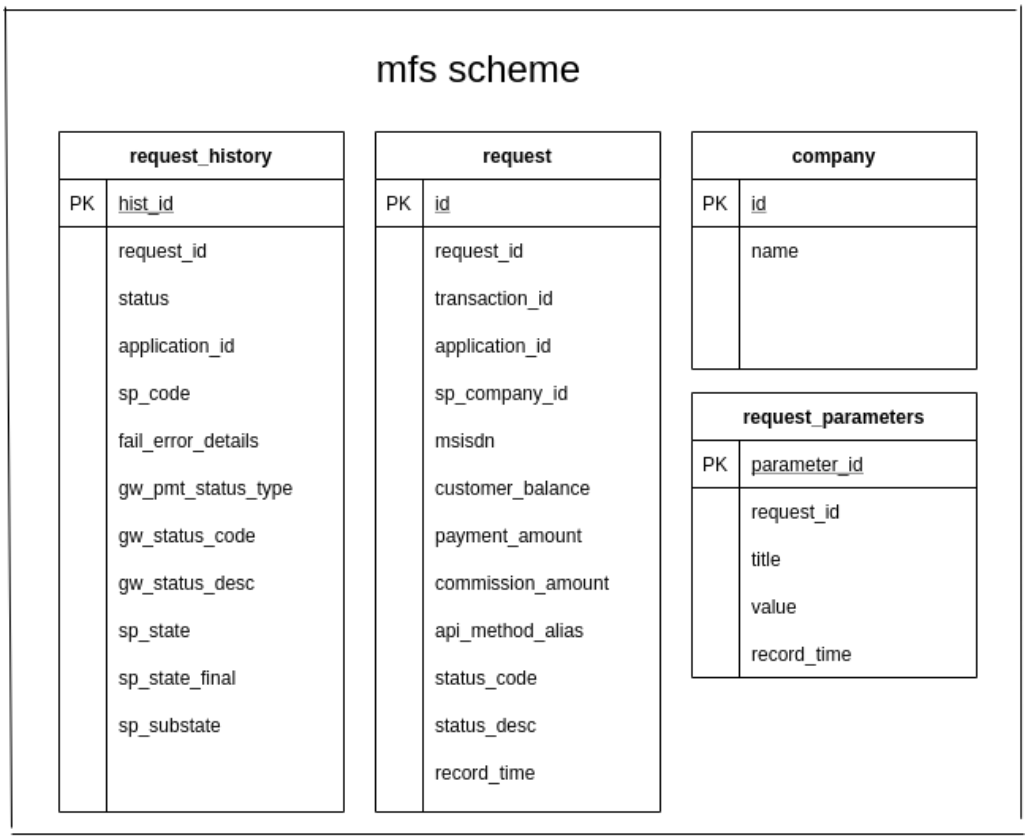

Figure 22. *Current MFS tables in Oracle*

5.3 Theories in Practice

In the previous chapter, we have made our proposal. We defined three steps of Scheme Design in NoSQL Database for data migration from RDBMS. The steps are:

1. analyze current structure
2. design scheme in Cassandra based on queries in Oracle
3. design mapping entities (for new db, Cassandra, design) in code

5.3.1 Analyze current structure

In previous section we discussed the database structure we use at the moment. We discussed tables, purpose for the existence of tables, and the relations among them. Now, researcher analyzes the queries that are used in daily lifecycle of the MFS project. As we plan to migrate data into Cassandra, query analyzing is very important. Becauase Cassandra is query-oriented database.

When MFS project started to be developed there was no intention of storing request parameters that comes from client. Because team decided that's not needed to have. But after a while, team received a task from business guys that these request parameters are necessary, so we started to think how to solve this issue. There were two option, adding a new column in order to store parameters in the existing request table, and creating new one. But the first option required more job than the second. The first one required to involve the database admins to rearrange the database performance tuning, check and update table configuration and etc. The second one is simpler, just add a new table, and join it to the main table with foreign key. Because of time issue team decided to go with the second option. The other issue we faced after the release of project is saving time in database. As we had several instances of database, and we store date without time, application stores some rows by wrong order. We have a log table which aimed to store all the steps of a request. All these rows contain a column for request status, and we know what happened to a request by this status. But, as we have this time issue we needed to update our SQL query. That's definitely a thing we would consider as a problem, and would try to fix it if we had an option to re-create the database. These two things are the main issues we have in the system at the moment. At the moment there are three tables for storing data of a request, one is main table, the other is log table and the other is parameters table. The advantage is it makes developer to write simple code, but team is not happy with this kind of structure. Having one table instead of three can cause some extra code and maybe a little different code architecture, but from database side we would have much better

database structure and much simpler select queries.

Here is the queries we use at the moment:

- get daily/monthly payment amount of a customer (record time and customer number used)
- get not succeeded payment requests in last *n* days (request status used)
- get payments of a customer (customer number used, order by time)
- get a payment request of a customer (customer number and request transaction id used, order by time)
- get a payment request record (request id used)
- get a company record (company id used)

5.3.2 Design scheme in target database

In Cassandra, it is important to create a keyspace. We created and named in *mfs* as it is name of the project. The options of the keyspace are simple. We used *SimpleStrategy* as class parameter value. Replication factor is one (1) as it is not production yet and we have limited hardware resources. Here is the query used to create the keyspace.

```
create keyspace mfs
    with replication = {
        'class': 'SimpleStrategy',
        'replication_factor': 1
    };
```

All tables are created in *mfs* keyspace. After analyzing Oracle queries, researched groups them and has new model. In Oracle we have a query to get a payment request by id, so in Cassandra we create a table named **requestById**. This table is combine of three (3) tables from Oracle: **REQUEST, REQUEST HISTORY, REQUEST PARAMETERS**. One table is for storing all data related to requests. In this table there are two primary keys: *requestId, method* columns as we search for a *payment - request type* request by its id. All data is sorted in descending order of record time.

As we know, MFS project has three tables for a request. In Cassandra, there would be only one table, and as Cassandra supports storing JSON type data, request parameters can be stored as JSON. As Cassandra is query-oriented database, we need to check what

scenarios the MFS application has. In MFS, it is requested to get customer data by his/her msisdn, to get payment data by transaction id, to get payment requests by request type, and to get request by its id. As Cassandra does not support JOINS, and it is not suggested to have materialized views on production environment, researcher decided to have one table for each scenario and store only required data on these tables. If we need to get request by its type, system will query requests by method table, the column which represents request type is primary key of the table. The other column is request id which will be used later to get request data from request by id table. We can apply same logic for all other scenarios, too.

Another query is to get a payment request of a customer. We create a new table for this query and name it **requestByMsisdn**. That is because we need customer number and transaction id of the request. So, first we query **requestByMsisdn** table. By querying this table we get request id, and we use this id to query **requestById** to get the request data. To get all payments of a customer we just need to query **requestByMsisdn** table by given customer number. All data will be sorted by record time in descending order.

Next query is for fetching not-succeeded payment requests in last *n* days. In order to handle this task successfully, we create a new table and name it **requestByMethod**. As we have different types of requests, we have divided them into groups and each group of request has its own method name. Payment requests' method name is *makePayment*. So, we create this table and make *method* column primary key. After all these all we need is to query this table by the payment type, in our case *makePayment*.

To get daily/monthly payment amount of a customer, we just query **requestByMsisdn** table. And the last query is to get company by id. In order to solve this issue, **companyById** table should be created. All the keywords after *'by'* in table names, represent the primary key column of the table. The primary key columns must be used in querying operation, unless we query for all data.

To summarize the queries in Cassandra:

1. get request data from **requestById** table
2. get payment request(s) of a customer by querying **requestByMsisdn** and **requestById** tables, respectively.
3. get not-succeeded payment requests data from **requestByMethod** table
4. get daily/monthly payment amount of a customer by querying **requestByMsisdn** table
5. get company data from **companyById** table

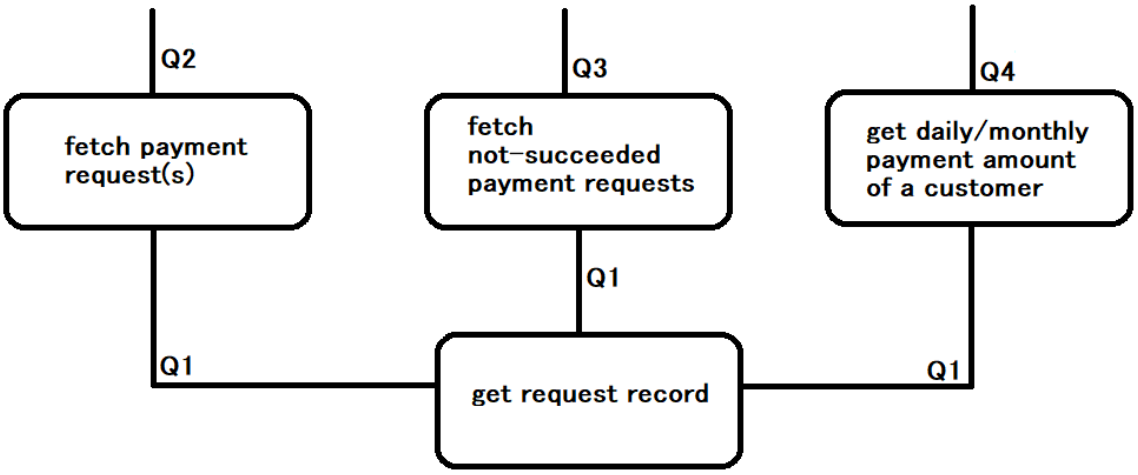

Figure 23. *MFS Query Diagram in Cassandra*

Fig 23 illustrates the potential query diagram of MFS project in Cassandra. As it seems all query results can be used to fetch request data from **requestById** table, it is not mandatory to query this table after all other queries, it just can be done. **Q2** requires customer number, and transaction id in case to get a single payment request. **Q3** requires list of not-succeeded request status. **Q4** requires customer number to get the payment amount of the customer.

Fig 24 shows the tables designed for MFS in Cassandra. There are five query tables. All tables have been designed by analyzing the queries used by application. If application needs to find a request, it firstly query the "request by msisdn" table, then "request by id" table. This process is same for other purposes. There is one table here which is not used for fetching request data, and it is "company by id" table. We can call it catalog table, it is used to get some data of a company at most of the time.

5.3.3 Design mapping entities in code

As we discussed this step earlier, any kind of dedication on coding part would not be necessary in case of migration to some other NoSQL database. But it is necessary in our case, as Cassandra is query-oriented database. That was the theory. In practice, researched realized that it is not a difficult task to do. We just need to create entities that represents our tables in Cassandra. And based on the programming language or framework it is up to us how to migrate data. We just simply fetched all data from Oracle database, and structured them in a suitable way for Cassandra, and inserted them in Cassandra database.

As we use Spring Boot for development, we will have to remove existing libraries for Oracle, and add new ones for Cassandra. Additionally, we have to re-think about code structure, as it will require lots of changes. Firstly, some entities will be deleted, and

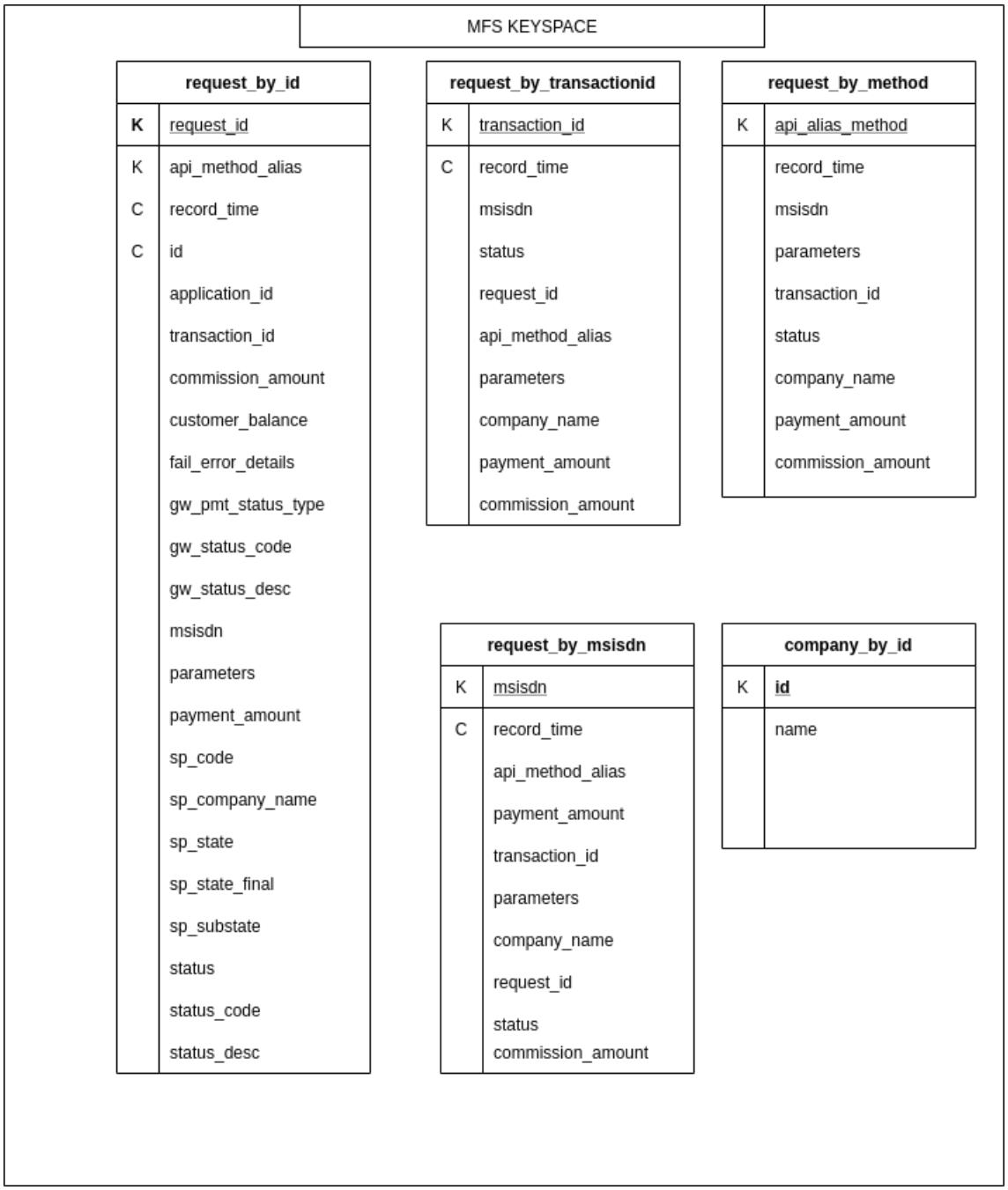

Figure 24. *MFS Tables in Cassandra*

some will be updated. The same will happen to repository classes as well. But the main code change will happen in service classes where all business logic happens. Storing new request data in multiple tables, running several select queries on multiple tables instead of having one complex SQL query, and writing its code and some other code changes are the challenges that waits for development team.

5.4 Performance Evaluation

In this section we discuss the performances of all queries in each database. Below there are two different figures which aimed to illustrate the performance values of MFS project queries in each database. In these tables, we can see query, database name, number of record and time required to perform the query.

System Properties

Before starting to performance evaluation, researcher would like to give some information about the system property values. In case of server options researcher is limited. Therefore, it has been used a single server with 16GB of RAM, and AMD Ryzen 5 2600 Six-Core Processor. Both simulations have been operated on the same system/server.

Select Queries in Oracle and Cassandra

In this section we discuss the queries that we are currently running on Oracle database. These queries are little bit complex, and requires time to understand what they are really doing. Additionally, almost all of them have JOIN clauses which makes the database do additional jobs in order to return the desired data. The select query below is being used to get all payments of a customer. To do this all we need is customer msisdn. The query joins three tables: company, requests and request history tables. Therefore it applies three JOINs. In the conditions part, we specified that we only need 'makePayment' requests, and requests with certain status.

```
SELECT *
  FROM (SELECT r.id request_id,
        r.transaction_id, r.application_id,
        c.id company_id, c.name company_name,
        r.customer_balance, r.payment_amount,
        r.commission_amount, r.record_time payment_date,
        h.status system_status, r.status_code, r.status_desc
        FROM MFS.REQUESTS r
        JOIN MFS.REQUEST_HISTORY h ON h.request_id = r.id
        LEFT JOIN MFS.COMPANIES c ON c.id = r.sp_company_id
        WHERE r.api_method_alias = 'makePayment'
        AND r.msisdn = <customer-msisdn>
        AND r.user_changed = 0
        AND h.status IN ('STARTED', 'CHECKING_STATUS',
              'GW_TIMEOUT_OR_OTHER_ERROR', 'GW_CONN_ERROR',
              'GW_UNSUCCESS_FINAL', 'GW_UNSUCCESS_NONFINAL',
```

```
        'CHARGE_TIMEOUT', 'REFUND_TIMEOUT',
        'REFUND_FAILED', 'REFUNDED',
        'ACCEPTED_BY_DEST', 'IN_DEST_PROGRESS', 'FINISH')
ORDER BY r.record_time DESC,
    h.hist_id DESC, h.record_time DESC);
```

The queries below do same job as the first one, but in a different way. As we only have msisdn of customer we are not able to search in request by id table. So, firstly we query request by msisdn table and get the request ids. Then we use these ids in order to get request data from request by id table. And the last part, to show company name that request belongs we use **spCompanyId** column value to run select query on company by id table.

```
SELECT *
FROM MFS.REQUEST_BY_MSISDN
WHERE msisdn = <customer-msisdn>
AND api_method_alias = 'makePayment';
```

Same or similar logic can be applied for other scenarios as well.

Performance Analysis

Fig 25 it is possible to see that almost most of the queries required more than 3 seconds to return a result. The best performance is of getting the company as company table has only 36 rows. The second place is getting request parameters. Request parameters table was created a long time later than the others, so it is OK it to have less data then others. We need to consider that it is possible for a (payment) request to have more than three (3) parameters. To get the parameters of a request requires 664 milliseconds. We perform a joining operation between parameters and requests table to perform this query. All following queries are performed in multiple tables - three (3) tables. Sum of all these table rows are more than 1.5 million, but we rounded this number to 1,5 million. To get all payment records of a customer requires more than three (3) seconds, but to get a particular one requires a little bit more than that. We need to wait 4 seconds for that as it makes transaction id comparison. In some payment requests we receive a response from 3rd party API to be informed that payment did not failed, but it in queue. So, we check those payments, and this operation takes three and half seconds (3.5). To get payment amount of a customer requires 3 seconds in Oracle database.

It is a little bit different in Cassandra. We do not have a query that requires more than a second. Actually all queries respond in less than 100 milliseconds. To get a request data we need to query a table with 2 million rows. This number is sum rounded. To

Query	Database	Record Number	Performance Time
fetch request parameters by request id	Oracle	60.000	664ms
get company by id	Oracle	36	50ms
get single payment request by customer number and transaction id	Oracle	1.5 millions	4s 203ms
get all payment requests of a customer	Oracle	1.5 millions	3s 595ms
get not-succeeded payment requests in last *n* days	Oracle	1.5 millions	3s 499ms
get payment amount of customer in last *n* days	Oracle	1.5 millions	3s

Figure 25. *MFS Query Performance in Oracle*

query 2 million row table we only need 47 milliseconds. To get payment request(s) of a customer we need to query one table only. That is the same for other purposes as well. All

Query	Database	Record Number	Performance Time
get all data of a request by request id	Cassandra	2 million	47ms
get company by id	Cassandra	36	36ms
get single payment request of customer	Cassandra	3 million	39ms (by transactionid tbl)
get all payment requests of a customer	Cassandra	3 million	36ms (by msisdn tbl)
get not-succeeded payment requests in last *n* days	Cassandra	3 million	62ms (by method tbl)
get payment amount of customer in last *n* days	Cassandra	850.000	38ms

Figure 26. *MFS Query Performance in Cassandra*

these are good. We designed a keyspace and we are able to get results quicker than we get them in Oracle. Cassandra does not support complicated queries. All queries above are complicated queries, and to get the desired result it is important use programming language. In code, we perform a query, then we get some values and based on these values we perform another query in another table to get the result we are pursuing from the beginning. As we design a structure based on an existing project, we are limited with business model, and we have a huge code base and it is not a good idea to re-write everything from beginning to design the database structure more freely. Maybe it can be done, it is up to company, and it requires resources to do that.

There can be a question here: why tables in Oracle have different amount of rows than tables in Cassandra? The answer is simple. Because we redesigned it in Cassandra, and there is a table in Oracle which is used as log table. All stages of a request is stored in "REQUEST HISTORY" table. We do not need this kind of concept in Cassandra. Instead, researcher decided to keep all stage information of a request in one table. The other thing is that "REQUEST PARAMETERS" table in Oracle. In Cassandra, researcher also does not see any reason to have a table for this purpose. Instead, request parameters can be stored inside requests table. There are some certain queries which used during application life-cycle. Therefore, comparing those queries in both databases is necessary. In this case, it should be understandable to compare two different tables those contain different number of rows.

6. Conclusion

This chapter summarize everything we did in this research. At the beginning our aim was to decide how to design a non-relational database in order to migrate data from relational one. There several types of non-relational databases out there, and our choice is Cassandra. Cassandra is column based and query oriented database. And our choice for RDBMS database is Oracle, as researcher uses Oracle database everyday as a software engineer at a company named Azercell.

It is not easy task to migrate data from one database to another. There has to be some realistic reasons for that. For example, current database does not perform well, or does not handle data well. There can be lots of reasons to change the database. In case somebody decided to do that, what are the steps this somebody is going to face? First of all, it is very important to analyze existing structure well. Otherwise new database design will fail in some phases. In case the first step is done, the following task to do is to design target database structure. As there is an existing project, we already have business model, and we know performance result of current database with this business model and we also know future plans. So, we should use these information to design target database structure. Then, it is important to handle entity mapping in codebase. The last step may be or may not be necessary as it depends on the programming language or framework we use for the application. These three (3) steps are the guideline of designing non-relational database structure to migrate data from relational one.

Not everything is done with just migration. Every database has different priorities and these priorities require some issues to handle. In our case, Cassandra has been developed to store huge amount of data, and it does not support joins like RDBMS. So, it should be expected to store same data in different tables. This case requires good network management. It is suggested to have multiple data centers to store data in Cassandra, and this requires a few good database administrators and network administrators. Paritioning strategy, number of physical and virtual nodes, primary keys of each table, queries of the application, replication factor and consistency levels should be analyzed and should be chosen very well in order to get the possible best performance from the Cassandra database.

Cassandra does not support complicated queries. Its strategy is having a table for

each query of the application, and running simple query in required tables rather than performing a complicated query in one table. This might require to handle some part of the a task in code. For example, it is not possible to have a sub-query in Cassandra, or using *in* clause on non-partitioning key column. If we use RDBMS, we are able to run a complicated query and we just receive a resultset, and we use it in codebase of the application. But in Cassandra, it is not like that. We might have to perform some filtering in code, or using application code to send multiple select queries to get the desired result.

As Cassandra stores data in denormalized structure, it means we can get same data from different tables. There is a question here: **how to perform *insert, update, delete*queries?** It is the decision that will be made by the team lead. There are some steps that can be followed here.

1. create one main table and multiple materialized views for each query. In this way, we are only responsible for the table. If we want too add, delete or update a record we only send a request to be performed in one table. The only issue here is materialized view is suggested to use in production environment yet. The other reason not to use it may require more time than a table.
2. create a structure like RDBMS. The issue here is Cassandra does not support joins. So, joining operations must be handled in client side - codebase of the application. Developers should be very careful. This way is not suggested as we treat Cassandra as RDBMS database. In Cassandra denormalization is ok, and it can perform very well even in this case. In this way, we do not benefit Cassandra's most properties.
3. create tables for each query. In this case, it is very important a software developer to be careful. There might be several insert, update and delete requests to Cassandra database for just a single record. To minimize that we suggest to design the structure in a way that allows you not to perform several requests for a single record.

Researcher suggest to use a single persistence API inside project to handle database operations. Not for only Cassandra, but for every database, but in our case it is very important to have this module. This module will receive some input data, and will perform database operation based on passed input parameters.

6.1 Future Work

As we discussed in previous section, data insertion and manipulation operations in Cassandra is the tricky and the hardest part. That can be a task to solve in future. A research that focuses on this subject would be very helpful for everyone who plans to use Cassandra for their applications. As Cassandra is query oriented database and it does not support

"joins", and in some cases "in" clause it makes difficult to choose Cassandra as application database. At the moment, Cassandras best use scenario is using it for reporting purpose, where mostly fetching operations are needed, and inserting is very limited. Additionally, in these cases updating and deleting are not operated at all. Some can see this as a failure when we consider huge amount of work behind Cassandra. Although reporting is very important process, researcher partially agree with that opinion. Having a database with these capabilities, and not being able to use it as main database is disappointing. Therefore as we mentioned earlier, future work could be how to configure Cassandra to make it efficient and effective for inserting/updating queries. It can be a new database with more capabilities developed based on Cassandra, or new methodologies to apply on Cassandra.

Bibliography

[1] Martin Kleppmann. In: *Designing Data-Intensive Applications*. 2019.

[2] Jan Paredaens et al. In: *The Structure of the Relational Database Model*. 2013.

[3] Masato Matsui Takao Bakuya. "Relational database management system". In: (2018).

[4] *Oracle Create Table Query*. URL: https://livesql.oracle.com/apex/livesql/file/tutorial_D39T3OXOCOQ3WK9EWZ5JTJA.html.

[5] Manas Ranjan Rabi Prasad Padhy. "RDBMS to NoSQL: Reviewing Some Next-Generation Non-Relational Database's". In: (2018).

[6] Ali Davoudian, Liu Chen, and Mengchi Liu. "A survey on NoSQL stores". In: *ACM Computing Surveys* 51 (Apr. 2018), pp. 1–43. DOI: 10.1145/3158661.

[7] *KeyValue Storage Architecture*. URL: https://aws.amazon.com/nosql/key-value/.

[8] S. Gilbert and N. Lynch. "Perspectives on the CAP Theorem". In: *Computer* 45.2 (2012), pp. 30–36.

[9] *CAP Theorem - Parts*. URL: https://cryptographics.info/cryptographics/blockchain/cap-theorem/.

[10] *CAP Theorem*. URL: https://medium.com/swlh/cap-theorem-in-distributed-systems-edd967e7bdf4.

[11] *Introduction Oracle*. URL: shorturl.at/mwQZ1.

[12] *Why-Cassandra*. URL: https://dzone.com/articles/the-top-10-reasons-to-use-cassandra-database.

[13] *Oracle History*. URL: https://en.wikipedia.org/wiki/Oracle_Corporation#History.

[14] *Oracle DB Query*. URL: https://docs.oracle.com/cd/B28359_01/server.111/b28310/create003.htm#ADMIN11080.

[15] *Oracle Table Query*. URL: https://www.oracletutorial.com/oracle-basics/oracle-create-table/.

[16] *Oracle Insert Query*. URL: https://docs.oracle.com/cd/B14117_01/appdev.101/b10807/13_elems025.htm.

[17] Prashant Malik Avinash Lakshman. “Apache Cassandra”. In: (2008). URL: cassandra.apache.org.

[18] *Cassandra Query Language*. URL: https://cassandra.apache.org/doc/latest/cql/ddl.html.

[19] *Cassandra Architecture*. URL: https://cassandra.apache.org/doc/latest/architecture/index.html.

[20] *Cassandra Data Modelling*. URL: https://cassandra.apache.org/doc/latest/data_modeling/index.html.

[21] Samah Bouamama. “Migration from a Relational Database to NoSQL”. In: *International Journal of Knowledge-Based Organizations* 8 (June 2018). DOI: 10.4018/ijkbo.2018070104.

[22] Feroz Alam. “DATA MIGRATION: RELATIONAL RDBMS TO NON-RELATIONAL NOSQL”. In: *M. Sc. Thesis, Ryerson University, Toronto, Ontario, Canada* (2015).

[23] *Azercell*. URL: https://www.azercell.com/en/company/.

I. Licence

Non-exclusive licence to reproduce thesis and make thesis public

I, Soltan Garayev

1. 1. herewith grant the University of Tartu a free permit (non-exclusive licence) to reproduce, for the purpose of preservation, including for adding to the DSpace digital archives until the expiry of the term of copyright, **Scheme Design in Non-Relational Model Database to Migrate Data from Relational Model Database**, supervised by Pelle Jakovits, PhD.
2. I grant the University of Tartu a permit to make the work specified in p. 1 available to the public via the web environment of the University of Tartu, including via the DSpace digital archives, under the Creative Commons licence CC BY NC ND 3.0, which allows, by giving appropriate credit to the author, to reproduce, distribute the work and communicate it to the public, and prohibits the creation of derivative works and any commercial use of the work until the expiry of the term of copyright.
3. I am aware of the fact that the author retains the rights specified in p. 1 and 2.
4. I certify that granting the non-exclusive licence does not infringe other persons' intellectual property rights or rights arising from the personal data protection legislation.

Soltan Garayev
10/AUG/2020