NTNU – Trondheim
Norwegian University of
Science and Technology

Supporting the Join Operation in a NoSQL System

Mastering the internals of Cassandra

Christian Peter

Master of Science in Informatics
Submission date: May 2015
Supervisor: Svein Erik Bratsberg, IDI

Norwegian University of Science and Technology
Department of Computer and Information Science

Abstract

The join operation is one of the most valuable operations found in traditional database management systems. With this operation, it is possible to join data from multiple tables. Today, most NoSQL systems do not support the join operation. One of the reasons for why these systems do not support this operation is that it is too time-consuming when the data is replicated across multiple nodes. However, it is possible to accomplish the same result with two other options, denormalizing of the data or joining at the application level. Denormalizing will result in more redundant data and both options will involve the user more in the execution of join. Support for the join operation in the query language of a NoSQL system may ease the change of database system for some users that only wants to use a NoSQL system where data can be joined.

This thesis presents an implementation of the equijoin in Cassandra since the two other options shown above are already covered by others. Cassandra is a NoSQL system classified as an extensible record store that is quite similar to the relational model used by, for example, MySQL. This implementation shows how the parsing, preparation and execution of the query are performed. Enabling support for queries that can be written in Cassandra Query Language (CQL) is done in the parsing step. A way of finding the join order that allows only one read of the table from memory or disk is also implemented. This join order is also slightly optimized where selections in a where clause are executed early on in the execution step. During execution, the nested loop join is used to accomplish the process of joining tables.

The implementation of join in Cassandra shows a significant worse execution time than MySQL. One of the problems with Cassandra is the underlying architecture that is not designed for the purpose of joining data from multiple tables. However, this thesis shows that it is possible to support the join operation in Cassandra, but it still need some further work to execute within a reasonable time.

Sammendrag

Join-operasjonen er en av de mest verdifulle operasjonene som finnes i tradisjonelle database systemer. Med denne operasjonen er det mulig å kombinere data fra flere tabeller. I dag har de fleste NoSQL-systemer ikke støtte for denne operasjonen. En av grunnene til at disse systemene ikke støtter denne operasjonen er at det er for tidkrevende når dataene blir replikert over flere noder. Imidlertid er det mulig å oppnå det samme resultatet med to andre alternativer, denormalisering av dataene eller join på applikasjonsnivået. Denormalisering vil resultere i mer redundante data og begge alternativene vil involvere brukeren mer i utførelsen av join-operasjonen. Støtte for operasjonen i spørrespråket til et NoSQL-system kan lette bytte av databasesystem for noen brukere som kun ønsker å bruke et NoSQL-system der data kan forenes.

Denne masteroppgaven presenterer en implementasjon av equijoin i Cassandra siden de to andre alternativene vist ovenfor allerede er undersøkt. Cassandra er et NoSQL-system klassifisert som et "extensible record store" som er ganske lik relasjonsmodellen brukt av for eksempel MySQL. Denne implementasjonen viser hvordan parsing, forberedelse og gjennomføring av spørringen utføres. Støtte for spørringer som kan skrives i Cassandra Query Language (CQL) er gjort i parsing-trinnet. En måte å finne join-rekkefølgen som gjør at det bare trengs én lese operasjon for hver tabell fra minnet eller disken er også implementert. Denne join rekkefølgen er også litt optimalisert der seleksjonene i en where-klausul blir utført tidlig i gjennomførings trinnet. Under gjennomføringen blir nested loop join brukt til å utføre prosessen med å forene tabeller.

Implementasjonen av join i Cassandra viser en betydelig dårligere kjøretid enn i MySQL. Ett av problemene med Cassandra er at den underliggende arkitekturen ikke er beregnet for det formål å forene data fra flere tabeller. Denne masteroppgaven viser at det er mulig å støtte join-operasjonen i Cassandra, men at det fortsatt trengs noe videre arbeid for å gjennomføre operasjonen innen rimelig tid.

Preface

This master thesis was written over an one year period at the Norwegian University of Science and Technology (NTNU). It is assumed that the reader of this thesis has basic knowledge about traditional database systems including transaction management and the ACID properties.

I would like to thank Svein Erik Bratsberg for valuable guidance through this master thesis. I would also like to thank the contributors of Cassandra making it open source and answering my questions on IRC (Internet Relay Chat). This help has been crucial for finishing my master thesis.

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

The join operation is an important service that a database management system (DBMS) should support. The most well-known providers of database management systems provide this operation, however, most NoSQL (often referred to as Not only SQL or Not relational) systems do not. When a user switches from a traditional relation DBMS to a NoSQL system, it may be confusing to understand an entirely new data model where the join operation (among others) is not supported. If the join operation had been introduced to the different NoSQL stores, it could ease the change of database system.

NoSQL is a common designation for many database systems that have emerged because of big data. These systems often support good scalability (horizontal) and the support for simple write/read operations. However, different from the most well-known relation database management systems, each NoSQL system has it own interface. Two NoSQL systems, Cassandra and MongoDB have completely different query interfaces. Cassandra uses a query language (referred to as CQL) very similar to SQL while MongoDB uses a query language based on JSON that differs a lot from SQL.

When creating a database, different tables are created to best define the real world. A manufacturer may want to store all its products and the customers. This case will also include the trades between the customer and manufacturer. To find all the trades between the manufacturer and the customer, a join of the different tables must be executed. A web application may want to have a forum where different users can write posts. A join is then required to find all the posts from one particular user. As shown here, there are endless of opportunities for the join operation. It is used for all types of applications where there exist relations between tables.

This master thesis looks at how the join operation can be implemented in one NoSQL store. Since each NoSQL store has different interfaces, the implementation of join is not generalizable for all NoSQL systems, but it is possible to show that the join operation can be supported in a NoSQL system.

1.1 Background and motivation

The join operation is used by many applications, and it is very useful in many cases where you want to combine data from multiple related tables. NoSQL systems can somehow be the future of database systems where scalability with high read/write throughput is supported. However, today many NoSQL systems do not support the join operation. With a join implementation in a NoSQL system, the transfer from a relational DBMS (for example MySQL) is simplified. The interest for this subject is also shown in Chapter 3, where already existing solutions exists on top of one or more NoSQL systems.

To accomplish a join in a NoSQL system, a denormalization of the data is needed to place all data that is related in the same table. A denormalization requires that the user has knowledge about the query before starting the data modelling. In SQL, there is possible to define different tables and to create the queries afterwards if they are needed. For example a person and the cars that is owned by this person. It is then possible to create multiple tables that contain the relation between person and cars. However, in a NoSQL system, this data has to be placed at the same table and thereby creating a lot of redundant data. With a join implementation, this redundancy is avoided, and it is also possible to define queries afterwards if needed. In addition to this, there is impossible for a user to know all queries that may be needed in the future. A developer in a business may not need the join query, but maybe another employee need it later on. Support for the join operations may remove this problem.

Creating a join functionality in a NoSQL system will help the end user. It is still possible to do the join with denormalization, but the user should also have the possibility to join tables without denormalization. The main motivation for this thesis is based on this. Implementing an operation in a NoSQL system that has the potential to helping users of NoSQL systems. In addition to this motivation, the join operation has had a significant influence on the database world (for example, the number of research articles on this operation is high). Having the opportunity to implement a join operation in a NoSQL system shows how complex this operation is, but also the mechanics laying behind a successful database system.

1.2 Definition and goals

For this master thesis, a set of goals has been determined including a problem definition. The definition is first presented, and then some formalized goals related to the definition is given. The problem can be defined as following:

Problem definition. *The purpose of this thesis is to evaluate the possibility of incorporating the join operation in a NoSQL system. This includes how to formulate joins in the query language or API. We would like to know if the lack of join operation in NoSQL databases is due to architectural decisions, making join hard to implement, or if it is straightforward to include it. The work will focus on a single node since distributed join is considered hard to implement, and will not bring forward issues which are specific to NoSQL databases only.*

The implementation should be seen as a proof of concept (POC). This POC should be a basis for further work on the join operation in a NoSQL system. An evaluation of the implementation should be executed and compared to an already existing database management system that supports join. A conclusion based on the findings found during the evaluation is made.

With this definition, a set of goals can be defined for this thesis. A NoSQL system that is well-known should be used. The first NoSQL system that was investigated was MongoDB. However, as seen in Chapter 3.2, some work has been done where MongoDB is used at the main NoSQL store. Besides this, the query language is quite different from SQL. A NoSQL system that has a similar language as SQL is Cassandra. Cassandra is one of the most well-known extensible record stores that can be found on the market. With a query language similar to SQL, it is possible to define join queries similar to SQL. Cassandra is therefore used as the NoSQL system in this thesis. Implementing the join operation in Cassandra is a general goal that consists of multiple small goals. In addition to this, the implementation must be tested, and compared against MySQL. This will reveal possible strengths or weaknesses of the implementation. The following goals has been set for this thesis:

Goal 1: Implement support for the join operation on one node in the Cassandra source code.

- **Goal 1.1:** Implement support for defining joins in the query language of Cassandra.
- **Goal 1.2:** Implement a possibility to add column families to a query that does not belong to an N-Way-Join.
- **Goal 1.3:** Implement a method for finding the correct join order.
- **Goal 1.4:** Implement the possibility for multiple N-Way-Joins.
- **Goal 1.5:** An implementation that supports the equijoin operation between N column families.

Goal 2: Test and evaluate the implemented join functionality in Cassandra.

Goal 3: Comparison of the join operation in Cassandra to the well-known MySQL database.

These goals were used as the main guidelines through this thesis. Some goals have been created at a later stage since some problems were found at different stages in the process. Finding a join ordering (Goal 1.3) is an example of this, since the first implementation did not take this into account. However, the three goals (Goal 1-Goal 3) have been very clear from the beginning of this thesis.

1.3 Report outline

This master thesis has eight chapters, but can be seen as three parts. These are: (1) background and related work, (2) design and implementation and (3) results and conclusion including further work.

Chapter 2 presents the background theory for the relational DBMS and NoSQL. It will also pay particular attention to the join operation and Cassandra. The next chapter that also belongs to part (1), is Chapter 3. This chapter presents some related work focusing on the join operation in NoSQL systems. Part (2) includes Chapter 4 and 5. Chapter 4 presents three different ways of accomplishing a join in Cassandra including the reason for why implementing the join functionality in the source code of Cassandra was chosen. The implementation of the join operation is described in Chapter 5. At last, a conclusion based on the evaluation and some further work is presented in part (3). Chapter 6 presents different execution times for the join operation in Cassandra MySQL. These results are then discussed and based on this discussion a conclusion is presented in Chapter 7. Since this is a thesis with limited time, a lot of work remains before this could be a production ready functionality in Cassandra. This work is presented in Chapter 8.

Chapter 2

Background

This chapter looks at the background for how join and query processing are done today. It will also look into Cassandra and the different tools associated with it. Other NoSQL systems are also included, for example, MongoDB. In addition to this, some general background on NoSQL and relational database management systems are presented in this chapter.

2.1 Query processing

Query processing is a way of processing a query that is issued to the database. The different steps of query processing are parsing, optimization, code generation and execution [47, 30].

The first step is parsing, where the syntax of the query is checked (against the rules of grammar). If no errors were found in the query, the next step is the optimization step [47, 30]. This step makes sure that a good plan for execution is created. However, this name can be misleading because the optimal plan may not be generated. It is more important to create a suboptimal execution plan and avoiding the worst [30]. After the optimization step is done, the next part is the code generator. It will generate code to execute the plan created by the optimizer [47, 30]. In System-R, the code generation replaces the parse tree with executable machine code [47]. The code generated can either be executed immediately or stored in the database for later execution [47, 30]. The last step is simply executing and returning the results to the end user.

2.1.1 Query optimization

Query optimization is the part of query processing that has the biggest effect on the execution time. As described in Chapter 2.4, an N-Way-Join consists of multiple 2-Way-Joins. If we define the two relations in a 2-Way-Join as outer and inner relation, the outer relation is usually (the first 2-Way-Join must join between

relations) the composite of the already joined relations and the inner relation is added to the composite result. This process continues for each 2-Way-Join. In System-R, a mix of joining algorithms (explained in Chapter 2.3.1) may be used. If there is a 3-Way-Join, the first 2-Way-Join may be executed with sort-merge join and the last 2-Way-Join may be executed with nested loop join. Also, the join ordering has a significant impact on how fast a join is completed. If it is n relations that are being joined, the number of different orders for this join is n!. For example will $n = 7$ result in 5040 possible different join orderings. However, Selinger et al. presents a heuristics method that reduce the number of join orders that are possible. For example, let's say relations R_1, R_2 and R_3 exists. R_1 and R_2 is joined on the same attribute, but R_2 and R_3 is joined on a different attribute. Then the plans $R_1 \bowtie R_3 \bowtie R_2$ and $R_2 \bowtie R_3 \bowtie R_1$ are excluded. This method is used whenever it is possible [47].

The next step is to find an optimal plan. Multiple trees are constructed which contains different solutions, and each tree is built with some steps:

Step 1 Each access path for a single relation is scanned. The cheapest one is kept with any ordering in mind. For example, if the sort-merge join or group/order by clause is used.

Step 2 This step then examines all joins of two relations. The results that were found in step 1 is used, and the cheapest solution is kept.

Step 3 The next step is to repeat step 2, but with three relations (with the results from step 2).

Step 3 continues (with four or more relations) until it finds all complete solutions, and the optimizer choses the cheapest solution [47]. This process is also defined as the enumeration of left-deep plans [46]. However, the optimizer has more to it than explained here. The reader is advised to the explanation of the optimizer in System-R [47] for further reading.

2.2 ANother Tool for Language Recognition

Cassandra uses ANother Tool for Language Recognition (ANTLR) version 3.2 as the parser and lexer. ANTLR is a parser generator that uses an underlying top-down parsing strategy, called LL(*) [43].

ANTLR makes it possible to construct recognizer, compilers, and translators. This construction is done by creating grammatical descriptions that contain actions. It is also possible to add code snippets to the grammar. This means that recognizer becomes a translator or interpreter [1].

2.3 Join

Join is an operation that is very useful in the relation algebra. It combines results from two or more relations. It can be viewed as a cross-product (R x S) between

relations where projections and selections are executed. However, a cross-product will have a much larger result (number of rows) than a join [46].

In this chapter, different join types are presented and different algorithms to accomplish a join are presented. Three popular join algorithms and variations of these are described below.

2.3.1 Join types

The join operation can be defined as the cross-product (or Cartesian product) with a following selection. Ramakrishnan and Gehrke [46] describes the join operation as following:

$$R \bowtie_c S = \sigma_c(R \times S)$$

where c is the condition that usually are the attributes of both R and S. This is the most general type of the join operation and this chapter includes some descriptions of some special cases for the join operation. Mishra [39] also defines this as a theta join where it is possible to use the theta operators, which are: $=, \neq, >, <, \leq, \geq$.

To exemplify the difference between the join types described in this chapter, an example schema exists with two relations, owner (ownerno, name, age) and car (regno, type, ownerno). The data in each relation is listed in Table 2.1. However, the row in table car that has ownerno 1 is unrealistic (there is no owner with ownerno 1), but it is necessary to exemplify the different join types.

regno	type	ownerno
AB1234	Sport	74
CC3344	Family	96
RR3333	Sport	1

(a) Car data

ownerno	fname	age
74	Kari	27
85	Hugh	36
96	Jack	55
99	Lars	77

(b) Owner data

Table 2.1: Example data for owner and car

Equijoin and nonequijoin

Equijoin consists of using the equal operator to join two or more relations. If there are two relations A and B, an equijoin can be written on the form A.name = B.name [39, 46]. However, if the theta operator equal is not used, the join is defined as a nonequijoin [39]. An example of equijoin between car and owner can be seen in Table 2.2.

ownerno	fname	age	regno	type	ownerno
74	Kari	27	AB1234	Sport	74
96	Jack	55	CC3344	Family	96

Table 2.2: $Owner \bowtie_{Owner.ownerno=Car.ownerno} Car$

However, as mentioned earlier, there also exists nonequijoins. An example of this type of join can be seen in Table 2.3.

ownerno	fname	age	regno	type	ownerno
74	Kari	27	RR3333	Sport	1
85	Hugh	36	AB1234	Sport	74
85	Hugh	36	RR3333	Sport	1
96	Jack	55	AB1234	Sport	74
96	Jack	55	RR3333	Sport	1
99	Lars	77	AB1234	Sport	74
99	Lars	77	CC3344	Family	96
99	Lars	77	RR3333	Sport	1

Table 2.3: $Owner \bowtie_{Owner.ownerno>Car.ownerno} Car$

Natural join

Another special case is the natural join that uses an equijoin. The equality is specified in all the fields that have the same name. For instance, the join result that is shown in Table 2.2 is a natural join (because the field ownerno is the same in Owner and Car). It can be expressed in relational algebra like this:

$$Owner \bowtie Car$$

In another example, two relations may not have any attributes that have the same name. If this happens and a natural join is executed on these two relations, the cross-product will be found instead [46].

Semi join

Semi join differs from the joins explained so far, because equijoin contains all attributes of the joined relations, whereas semi join only contains the attributes of the first relation. It is a join between two relations, R and S, with a projection where all the attributes of S is dropped. It is expressed like this:

$$R \ltimes S = \pi_{a_1,...,a_n}(R \bowtie_{a\theta b} S)$$

where $a_1, ..., a_n$ is all the attributes from relation R [39]. For the example data in Table 2.1, the result of a semi join can be seen in Table 2.4.

ownerno	fname	age
74	Kari	27
96	Jack	55

Table 2.4: $Owner \ltimes_{Owner.ownerno=Car.ownerno} Car$

Outer join

SQL supports a join type that is known as the outer join. Three different types of the outer join are described here. These are left, right and full outer join. A standard join is expressed as following in relational algebra:

$$Owner \bowtie_{Owner.ownerno=Car.ownerno} Car$$

This join only adds matching rows from the two relations to the result. However, this is not the case for the outer join. If an outer join is executed on the tables in Table 2.1, the result differs a lot from the standard join expressed above. A left outer join will add all rows from the left table and the corresponding rows from the right table. If there are no corresponding rows, null values are added instead [46]. A left outer join of owner and car can be seen in Table 2.5. This shows that all rows from owner are in the result and the matching rows from car. However, two rows does not have matching rows in the car table, and the fields (from table car) are therefore null.

ownerno	fname	age	regno	type	ownerno
74	Kari	27	AB1234	Sport	74
85	Hugh	36	null	null	null
96	Jack	55	CC3344	Family	96
99	Lars	77	null	null	null

Table 2.5: Left outer join for tables owner and car

For right outer join, it is vice versa. All the rows from the right table are added and only the matching rows from the left relation. If there are no matching rows in the owner table (for the ownerno in the car table), the fields from the table owner are set to null [46]. An right outer join is shown in Table 2.6.

ownerno	fname	age	regno	type	ownerno
74	Kari	27	AB1234	Sport	74
96	Jack	55	CC3344	Family	96
null	null	null	RR3333	Sport	1

Table 2.6: Right outer join for tables owner and car

The last version of outer join is the full outer join. This version will present the rows that do not have any match from both the left and right table (including those who have a match) [46]. A full outer join between owner and car can be seen in Table 2.7.

2.3.2 Nested loop join

This is the simplest join algorithm of all the three algorithms explained in this chapter. A nested loop consists of two relations, for example, A and B. Relation

ownerno	fname	age	regno	type	ownerno
74	Kari	27	AB1234	Sport	74
85	Hugh	36	null	null	null
96	Jack	55	CC3344	Family	96
99	Lars	77	null	null	null
null	null	null	RR3333	Sport	1

Table 2.7: Full outer join for tables owner and car

A is the outer relation and B is the inner relation. Relation A consists of multiple tuples, where each tuple $a \in A$ and for each tuple a in A, all tuples in B are iterated through. If there is a match, the tuples are added to the result R [46, 23]. An example of this type of join can be seen in Figure 2.1.

for each $a \in A$ **do**
  for each $b \in B$ **do**
    if $a_i == b_j$ **then**
      Add tuples a and b to R

Figure 2.1: Simple nested loop join [46]

In this nested loop, the outer relation is the smallest, and the inner is the biggest. This is because the total I/O volume will be less when the outer relation is the smallest [46, 23]. However, this simple algorithm does not utilize the buffer pages efficiently. The optimal solution is if there are two buffer pages left over when the smaller relation A is in memory. Then the smaller relation can be read and one buffer page can be used to read the other relation B. Then matching tuples from A and B can be added to the result. An output buffer is created with the second buffer page. This solution will have an optimal I/O cost compared to the simple nested loop join. However, the memory may be too small to hold the outer relation. It is then possible to break up the outer relation A into blocks (where each block fits into memory, and A is typically stored in blocks on disk). For each block from A, the inner relation B is scanned. This algorithm is called the block nested loop join [46].

Bratbergsengen [23] also presents a possible data structure for matching of rows from two relations which is shown in Figure 2.2. As seen in the figure, a bloom filter is used to filter away rows with keys that are not found in the work space (an area in memory that should not be larger than relation A). The data structure consists of a hash table and binary trees. The purpose of the hash table is to create smaller problems that come from a big problem. Then the binary trees can be used to search for the correct record. Each binary tree consists of four parts that are left and right subtree pointer, a record pointer, and a signature. Then the signature can be used to avoid comparison with the record if there is no match [23]. This data structure can be used to hold the blocks described above and, therefore, reduce the execution time of nested loop.

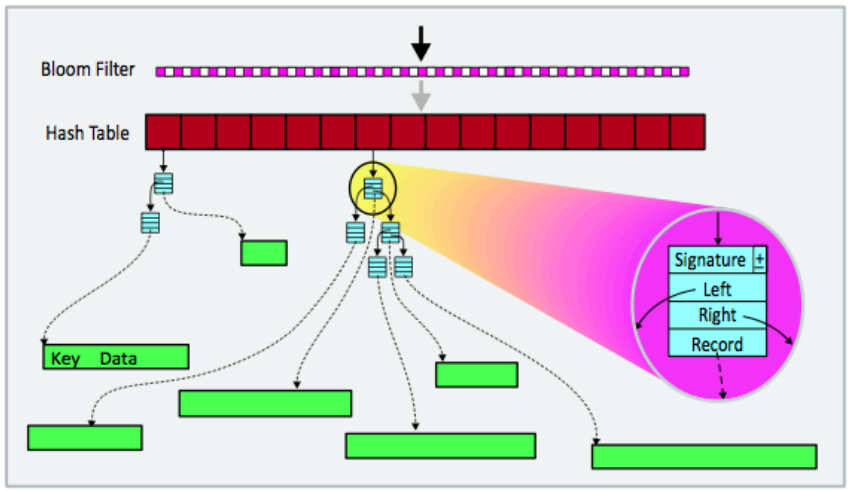


Figure 2.2: Layout for data structure in work space [23]

Another approach is called the index nested loop join where one of the relations has an index on the join attribute(s). The relation with the index will be the inner relation. For each outer relation, it is possible to find the matching tuples from the inner and outer relation by retrieving all tuples from the inner relation with the use of an index. This algorithm does not, unlike simple and block nested loop join, enumerate the cross product of the inner and outer relation [46].

2.3.3 Sort-merge join

The sort-merge join consists of two phases, sort, and merge. The first phase consists of sorting the two relations, A and B, on the join attribute. All tuples from A and B that have the same value (in the join column) are grouped together during the sorting step. This makes it easy to find partitions with tuples with the same value. Then it is possible only to compare tuples from A with tuples from B within the same partition and avoiding enumeration of the cross-product between A and B. However, this partition strategy works only for equality operations [46, 27]. To sort the relations, A and B, an external sorting algorithm (for further reading, a possible external sorting algorithm is described by Ramakrishnan and Gehrke [46] and Bratbergsengen [23]) can be used. However, if the relation is already sorted on the join attribute, no sorting is needed.

During the merge step, the relations A and B are scanned with the purpose of finding matching tuples. For example Ta in A and Tb in B where $Ta_i = Tb_j$. It is two scans, where each starts at the first tuple in each relation. The scan of A is continued as long as the tuple (current in A) is less than the tuple (current) in B. This check is done on the join attribute. This is the same for B, as it continues to scan as long as the current tuple in B is less than current tuple in A. It changes

between such approaches until matching tuples are found. When these are found, the joined tuple needs to be output. However, there may be multiple tuples in A and B, with the same join attribute value. These tuples are referred to as current A partition and current B partition. Then for each tuple a in current A partition, all tuples in current B partition is scanned. The tuples that are joined are written as output. The scanning of A and B is continued, where it begins at the first tuples that follow the partitions just processed [46].

2.3.4 Hash join

There are three types of hash join algorithms described by DeWitt et al. [27], and Ramakrishnan and Gehrke [46] describes only one of them, namely grace hash join. The two others are simple hash and hybrid-hash (which is also described by Bratbergsengen [23] as partial partitioning) join algorithm.

If there are two relations A and B, the simple hash join builds a hash table in memory (if there is enough space). After the hash table is created, it scans B, hashing each tuple b in B for a match in the hash table. However, if the hash table for A does not fit in memory, the simple hash join algorithm will fill the memory with a part of A, scan B as above, then create a new hash table for the next part of A, and scan the remainder of B. The algorithm will end when there are no more tuples from A that are passed over [27].

The grace hash join algorithm is a bit different since it executes in two phases. The first phase consists of using a hash function h_1, same for both A and B, to hash the join attributes. For both A and B, the hash function h_1 is used to hash the tuples and place them in the correct output buffer. When the buffer is filled, it is flushed to disk creating a partition. The second phases, also referred to as probing or matching is started. A partition of A (for example A_1) is read and an in-memory hash table is created with the hash function h_2. This hash function is different from h_1 because the tuples should be uniformly distributed in the hash table. Then the partition B_1 is read from disk and matched against the table using the hash function h_2 (same used to create the hash table). Tuples that match from A and B are written to the output. This matching is repeated for all partitions of A [46, 27].

The third hash join algorithm is called the hybrid hash join algorithm. If there is more memory available, this algorithm performs better because of minimized disk traffic. During the first step, only as many buffers as needed is used to hold k partitions for A (similar for B). A is scanned and hashed with the hash function h_1. If it belongs to partition 0, it is stored in a hash table in-memory. If the tuple does not belong to this partition, it is written to disk on the correct partition. Then B is scanned and hashed with the hash function h_1. If this tuple belongs to partition 0, it is matched against the hash table. When there is a match, a result tuple is created. The tuple is tossed if there is no match and it belongs to partition 0. However, if it belongs to another partition, it is written to disk to the correct partition. The second step reads each partition on the disk. For example is partition A_1 read from disk and a hash table is created with the hash function h_2. Then partition B_1 is scanned and probed against the hash table from A_1. If

there is a match, a result tuple is created. Otherwise, the tuple is tossed. This is repeated for all partitions of A and B on disk. This algorithm saves I/O because partition A_0 and B_0 does not need to be read from or written to disk. However, it requires enough memory to be executed properly [46, 27].

As for the block nested loop join described in Chapter 2.3.2, a similar data structure (see Figure 2.2) in memory can used to match partitions from two relations.

2.4 N-Way-Join

Each join between two tables is called a 2-way-join. In the case of join between N tables, an N-Way-Join exist. Ramakrishnan and Gehrke [46] defines this as a multiway join where multiple tables are joined to create one result table.

Selinger et al. [47] describes it as N-Way-Joins, and this is also the name that will be used further in this master thesis. An N-Way-Join will consist of multiple 2-way-joins between two tables [47].

2.5 Database management system

This chapter focuses on transactions and the ACID properties that are used in a database management system (DBMS). For further reading, Ramakrishnan and Gehrke [46] and, Elmasri and Navathe [30] describes different parts of a DBMS.

2.5.1 Transaction management

A transaction can be seen as a set of actions or an executing program. This program or actions performs reads or writes against the database. If there are multiple transactions, for example, transaction T_1 and T_2, all their actions is part of a schedule. The ordering of the actions in a transaction T must be same as the ordering in the schedule [30, 46]. The different types of operations or actions can be seen here:

- begin_transaction (b): When a transaction is started, this operation is used.
- read (r): Specifies a read operation.
- write (w): Specifies a write operation,
- end_transaction (e): When a transaction is finished, this operation is used, and it may be needed to check if the transaction should be committed or aborted.
- commit (c): If the transaction is successful, a commit operation is used.
- abort (a): On the other hand, if the transaction is unsuccessful, an abort operation is used.

The operations above are also used as transaction states for a possible recovery if something fails [30]. A possible schedule can be seen here:

$$S_1 : r_1(a), w_1(a), r_2(b), w_2(b), c_1, c_2$$

Schedule S_1 has two transactions, T_1 and T_2 where T_1 first reads item a and then writes item a. The same happens for transaction T_2, but with item b. In the end, both transactions commit their actions. This is a complete schedule. By complete, it is meant that all the transactions are either committed or aborted. Another type of schedule, know as serial schedule, is that each transaction is executed from start to finish. Schedule S_1 is also known as a serializable schedule, which means that S_1 is equivalent to a complete serial schedule (with the same transactions) [46]. The schedule above is serial because transaction T_1 could have been executed alone first, and then transaction T_2 could have been executed. Another and more complex schedule is:

$$S_2 : r_1(a), w_1(a), r_2(a), w_2(a), r_1(b), w_1(b), r_3(b), w_3(b), c_3, c_2, c_1$$

Schedule S_2 is also serializable because transaction T_1 could have been executed first, then T_2 and lastly T_3. However, when a transaction aborts, it may be impossible to undo the changes from an aborted transaction. Ramakrishnan and Gehrke [46] says that it not enough with a complete serial schedule, but that the transactions must also be committed. For example, a unrecoverable schedule is this:

$$S_3 : r_1(a), w_1(a), r_2(a), w_2(a), r_3(b), w_3(b), c_3, c_2, a_1$$

Transaction T_1 reads and writes item a, T_2 then reads and writes item a and T_3 read and writes item b. Then T_3 and T_2 is committed, however T_1 aborts. Then T_2 has written to item a based on what T_1 wrote, but T_1 aborted, and the value from this transaction is not valid. The opposite of a unrecoverable schedule is a recoverable schedule. For a schedule to be recoverable, each transaction in a schedule only commits their changes after all the other transactions that it has read anything from commits [30, 46]. It is also possible to classify a schedule as avoid cascading aborts, which means that that a transaction only reads the changes from committed transactions [46].

Having serializable and recoverable schedules is possible with the use of locks in a DBMS. Then the committed transactions do not lose their changes because another transaction aborts. There is possible to use different types of locks, where one of them are named Strict Two-Phase Locking (Strict 2PL). However, these are not described in this chapter. There are two mechanisms, blocking and aborting, that are used in locking [46]. These mechanisms bring a performance penalty, but as Ramakrishnan and Gehrke [46] says, blocking is the primary reason for the overhead.

2.5.2 ACID

ACID stands for Atomicity, Consistency, Isolation and Durability. These four properties should be supported by transactions in a DBMS. The DBMS manages

these properties with the use of recovery methods and concurrency control [30, 46]. In this chapter, each property is described and the enforcement of these.

Atomicity

Atomicity means that a transaction should be fully executed or not at all. In a DBMS, the transaction recovery subsystem is responsible for ensuring this property. A system may crash (for example power break), or a transaction may fail during execution. Then all operations executed by a transaction on the database must be undone with a recovery technique. However, if there was a committed write operation, this change has to be written to disk (eventually). This is possible due to the logging that happens in a DBMS [30, 46].

Consistency

With consistency, it is meant that a transaction should take the database from one consistent state to another when it is completely executed without any other transaction interfering. This property is ensured by the user (programmers that write database programs) of the DBMS. Let's say that the database is in a consistent state and the transaction has not been executed yet. Then the transaction is executed, and the database program should make sure that the database is in a consistent state after the execution is finished [30, 46].

Isolation

By isolation, it is meant that a transaction is executed in isolation from other transactions. No other transactions should interfere with the execution of a transaction. The concurrency control subsystem in a DBMS makes sure of this. It can be seen as execution all transactions in a serial order, even though they are not [30, 46].

Durability

A committed transaction and the changes from this should be sustained, even if there is a system crash. This property is maintained by the recovery subsystem of the DBMS, and it is possible to retain the system by reading the log [30, 46].

2.5.3 SQL

SQL stands for Structured Query Language [48, 46], which is a database language. It was first developed at IBM in the projects SEQUELXRM and System-R. This was the start and the other DBMS vendors started using this language shortly after [46]. Formal and mathematical theory is the ground foundation for SQL, and this is theory is based on the relation model that was first introduced by Codd in 1970 [48].

SQL supports multiple operations to be expressed and sent to the database. Some operations that are supported is the possibility to select, insert, delete and

modify rows. This is also known as a subset of SQL, known as the Data Manipulation Language (DML) [31]. Another subset of SQL is the Data Definition Language (DDL). It supports creation, deletion, and modification of definitions in tables and views. Some SQL statements may be executed at a later time. This is known as triggers, which will be executed by the DBMS. It will be executed if it meets the specified condition in the trigger. Another aspect is to allow a host language (for example Java) to call the SQL code, which is known as embedded SQL. Another type is Dynamic SQL, which makes it (SQL) possible to be constructed at run-time. There is also possible to specify how clients can connect to the database server. The user also has the possibility to specify commands (in SQL) for transaction management and different possibilities for user access are also possible to specify in SQL [46]. There exist more advanced features in SQL, but these are not covered in this chapter.

2.5.4 MySQL

MySQL [14] dates back to 1979, but was first public released in October 1996. However, the most interesting part for this thesis is the indexing used in MySQL. However, this is different based on which storage engine that is used. There are multiple storage engines, ranging from MyISAM, Memory, and InnoDB [42].

InnoDB is the storage engine that was used in this thesis, and it is therefore only InnoDB that is described here. It is the most complex storage engine in MySQL, where transactions, foreign keys, row-level locks and multi-versioning are supported. However, the most interesting part is how the data is stored. InnoDB uses something called clustering index, and this is a B-tree where the primary key is the key in the tree and the record in the data part. This implies that each table must have a primary key, however if it is not specified by the user, InnoDB creates one not visible to the user [42].

Besides this, it was mentioned logging in Chapter 2.5.2 to achieve the properties of ACID. InnoDB has two types of logs, undo and redo. The undo log makes it possible to undo all the actions of an aborted transaction while a redo log makes it possible to recover from a crash [42].

2.6 NoSQL

The definition for NoSQL, which stands for either "Not Only SQL" or "Not Relational", is not agreed upon [25]. However, Cattell [25] lists up six key features for NoSQL database systems. A NoSQL store can scale horizontally and distribute the data over many servers. The interface that the NoSQL stores have is usually simple and not like SQL. The concurrency model supported by ACID transactions is weaker in NoSQL. Distributed indexes and RAM are also used in a more efficient way, and it is possible to add attributes dynamically. However, all NoSQL stores are different, and the features mentioned above are not identical for all of the different types.

The three types of architectures for NoSQL that is described in this chapter are Key-Value, Document and Extensible record (also referred to as column-family store [38, 33, 21]) stores. As Cattell [25] describes, some authors use a broad definition of NoSQL. Some systems that fall under this definition are graph, object-oriented and distributed object-oriented databases. These are not explained in this chapter. For example, graph databases are defined as NoSQL system by Abramova [21].

In the NoSQL world, the ACID properties (explained in Chapter 2.5.2) are not supported (usually) [25]. Instead of the ACID properties, the BASE properties have been introduced which are explained in this chapter. Another term that is introduced in this chapter is the CAP theorem.

2.6.1 CAP theorem

The CAP theorem was first introduced by Eric Brewer [32], and it says that a system only can have two out of three properties (see example in Figure 2.3). These properties are Consistency (C), Availability (A) and Partition-tolerance (P). With consistency, it is meant that all the nodes contain the same data (at the same time). Availability means that if a request is sent, a response should be sent back and the last one, Partition-tolerance, implies that if one node fails, the system should not fail or crash [21, 44]. For example, a DBMS is most likely to be in the CA (consistency and availability) part of Figure 2.3. Which of these three properties that should be chosen are different from system to system [25, 21], but as Cattell [25] describes, most NoSQL gives up consistency, but the trade-offs are complex.

2.6.2 BASE

BASE stands for Basically Available, Soft state, Eventually consistent and is a term that is used for NoSQL systems because the ACID properties are not present in these systems. This is because giving up the ACID properties; it should be possible to have higher performance and scalability [25, 21]. The first part, basically available, means that if the data is distributed and a node fails, it still works. Soft state means that there is no consistency guarantee, but this does not mean that the data never will be consistent, hence eventually consistent [21, 44].

BASE is connected to the CAP theorem because, as explained in Chapter 2.6.1, two of the three properties can only be chosen, and BASE follows this. For example, the properties availability and partition-tolerance can be selected. Then the BASE properties hold because the database only needs to be eventually consistent, but there is no guarantee of the consistency. As may been noticed, between ACID and BASE, the most noticeable difference is the consistency. If this property is essential, a relational DBMS may be better to choose [21]. As Pritchett [44] describes: ACID is pessimistic, and BASE is optimistic. This is because the consistency with BASE changes over time (hence, eventually consistent), but ACID makes it so that each transaction leaves the database in a consistent state.

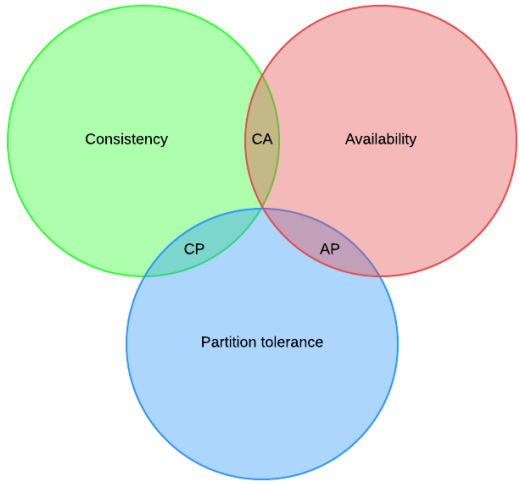


Figure 2.3: CAP theorem

2.6.3 Key-value stores

Key-value stores are, as the name suggest, a data store where the value is indexed, and it can be found by the use of a key [38, 25, 21]. This type of store includes features like transactions, locking, replication, persistence mechanism and other features [25]. It is also possible to hold both structured and unstructured data in key-value stores [38]. Some examples of key-value stores are Riak [17], Project Voldemort [19] and Redis [16].

2.6.4 Document stores

A document store is a bit different than key-value stores. Document stores make it possible to store more complex data than key-value stores. This type of store can contain documents or objects where each object may have different fields and data [38, 25, 21]. In addition to this, different types of documents can be used (for example XML or JSON) [25, 21]. Two well-known document stores that exists are MongoDB [13] and CouchDB [4].

2.6.5 Extensible record stores

Extensible record (or column-family) stores have the data model that is most similar to the relational data model. The data model for extensible record store consists of columns and rows [38, 25, 21]. Both rows and columns are distributed over multiple nodes. Sharding on the primary key is applied to split the rows on multiple

nodes, and a range split is often used instead of a hash function. Columns are also splitted across nodes and grouped together in column groups, and these column groups are pre-defined [25]. Some different extensible record stores that exists are Cassandra [3], HBase [8] and HyperTable [9].

2.7 Cassandra

Cassandra is a distributed storage system where it is possible to manage huge amounts of data (structured) over multiple nodes. There is no master-slave architecture, so there is no single point of failure [36, 20]. Cassandra was created for the Facebook inbox, where it was designed to handle high write throughput (billions of writes per day). Another aspect that was put into Cassandra was that it should scale well with the number of users [36].

In this chapter, some key techniques and components are mentioned. Not all components that are used to create Cassandra is mentioned here, but the most important. The data model in Cassandra is almost unrecognizable [28] from the data model presented by Lakshman and Malik [36]. The data model today is similar to the data model of tables and rows in SQL.

Cassandra needs to dynamically partition the data over multiple nodes because it should be possible to scale incrementally [36]. To manage this, the partition key is hashed, and a token is created. This partition key is unique for each row, and the token is used to distribute the row on the correct node. Consistent hashing is used to create this token. If then a node is removed or added, the reorganization is minimized (only neighbor nodes may be affected). Each node is therefore responsible for a range of hash values, as shown in Figure 2.4 [36, 20].

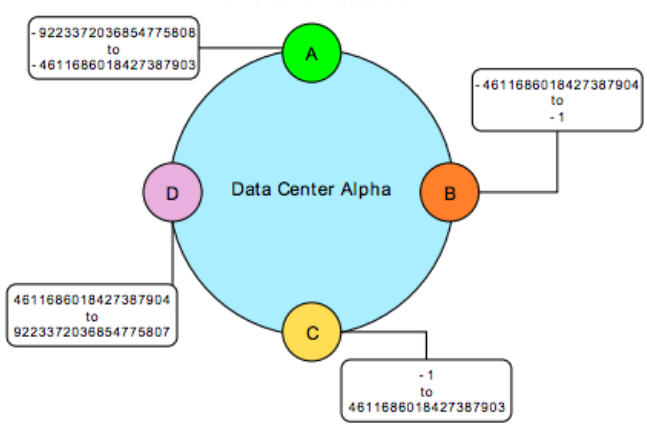


Figure 2.4: Partition responsibility for four nodes [20]

There is also used replication in Cassandra, to achieve high availability and durability. There is possible to configure the replication factor in Cassandra. For example, a replication factor of three means that each row is copied three times and

placed on different nodes. However, the key for a row is managed by a coordinator node. This node is also responsible to replicate the data (and all the other keys that it handles) [36, 20]. There is also possible to select two different strategies for replication that is based on the topology (racks and datacenters). These are [20]:

- SimpleStrategy: Used if there is only one data center (Nodes in the same collection). The replicas are placed clockwise in the ring structure. This strategy does not consider the topology.
- NetworkTopologyStrategy: This strategy is used when it is expected that the data is going to be replicated over multiple data centers. This strategy tries to place replicas on different racks (nodes on the same rack tends to fail at the same time).

Cassandra is also using a technique called gossiping (communication protocol that is peer-to-peer). This communication protocol helps Cassandra to share state and find location information. This information is found on other nodes in the cluster (all datacenters). A node does also exchange information about location and state for itself and other nodes (only those that the node knows about). There are sent multiple gossip messages, and a versioning number is associated with each gossip message. This number is a way of discarding old messages [20]. Gossip state and history can, for example, be used for failure detection. It is determined locally if a node is up or down. This enables Cassandra to avoid request being sent to nodes that have failed. Also, when a node starts up for the first time, a random token is chosen, and this token is the position in the ring. Gossiping is then used to send the information about the token to the other nodes that makes it possible to know about the other nodes and their position in the ring. When it comes to bootstrapping, a node that tries to join a cluster reads a configuration file that contains some nodes referred to as seeds. These seeds are the contact points for new nodes [36, 20].

Besides the components described above, Chapter 2.7.1 describes how data is stored in Cassandra, Chapter 2.7.2 presents the different keys and indexes used in Cassandra. The query language for Cassandra is also described in Chapter 2.7.3.

2.7.1 Storage

In Cassandra, the data can be stored on both SSTables and Memtables. As seen in Figure 2.5, when data is written to Cassandra, the operation is first written to a commit log (which ensures durability). If this is successful, the data is written to a memtable. When the size of the memtable hits a threshold (it is configurable in Cassandra), the data is flushed to an SSTable [36, 20]. The process is also referred to as a minor compaction by Chang et al. [26]. In this chapter, these two types of tables, which is very similar to the BigTable design [26], are described.

SSTables

Sorted String Table (SSTable) is used to store data on disk, and it is immutable. It has a mapping between keys and values. The keys and values are both strings

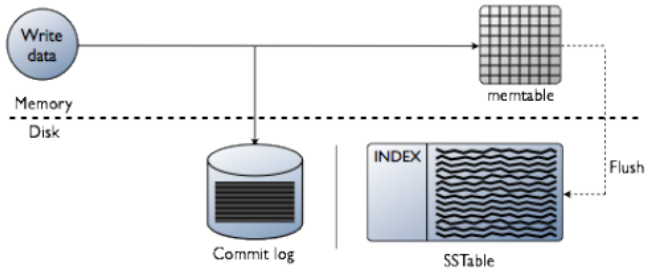


Figure 2.5: Memtable and SSTable architechture [20]

that can have an arbitrary byte size [26]. The SSTables in Cassandra are also append only, and they are maintained for each table in Cassandra. An SSTable in Cassandra also has some structures associated with it. Each SSTable has a list of partition keys (referred to as partition index) and the position for the start of a row. A partition summary is kept in memory that is the sample of the partition index, and a bloom filter is used to find data without needing to do any disk I/O [20].

In addition to this, a lot of SSTables could exist over time (because each minor compaction creates one new SSTable [26]). Cassandra uses a compaction strategy that is very similar to the one found in BigTable [36]. This process happens in the background and is periodical. This will remove data that is unnecessary. The result of this compaction is a new SSTable [20, 26]. In Cassandra, the data is not deleted in place, but a tombstone is created (which indicates that the data should be deleted). This is done because the SSTables, as described above, are immutable. Then when the compaction process starts, the tombstones are evicted, and deleted data is removed. After this, the SSTables are merged. During this process, the old and the new SSTable will exist at the same time which will give some more disk and I/O usage. As mentioned above, the compaction happens in the background, and the purpose is to minimize aggravated read speeds [20].

Memtables

As mentioned earlier, all write operations are written to a commit log and then to a memtable. Memory table (Memtable) is a sorted buffer, and the data is written to this after logging is done. If this structure reaches a threshold, the data is flushed to an SSTable (minor compaction) [20, 26].

Memtables are also a part of the compaction process described for SSTables. The memtable that was a part of the compaction can be deleted after the process is done [26].

2.7.2 Keys and indexes

In Cassandra, it is possible to define different keys when a column family is created. In addition to this, a secondary index can be created on different columns. The different types of keys and indexes are described in this chapter.

Primary key

Cassandra requires that at least one column is defined as a primary key. If it is only one column, a primary key keyword after the column can be used as shown in Listing 2.1

```
CREATE TABLE user (
  username varchar PRIMARY KEY,
  age int,
  first_name varchar,
  last_name varchar
);
```

Listing 2.1: Create table with a single primary key

However, there is also possible to define it as compound primary keys (described below) does it (with parentheses and only one column in between these). However, only one column is used between the parentheses. A single primary key will also act as the partition key (it is also referred to as the row key) for the data [6].

Compound primary key

In this type, a primary key consists of multiple columns. The first part of the compound primary key is defined as the partition key. The other columns are referred to clustering columns. The partition key will decide which node the data is stored on. The clustering column(s), on the other hand, will cluster (a process where an index is created and an order for the data is made with this index) the data on each partition [6]. This type of key can be created as shown in Listing 2.2.

```
CREATE TABLE user (
  username varchar,
  age int,
  first_name varchar,
  last_name varchar,
  PRIMARY KEY (username, age)
);
```

Listing 2.2: Create table with a compound primary key

In this example, the username column is the partition key, and the column age is the clustering column.

Composite partition key

A composite partition key is similar to the compound primary key described above. However, the partition key can contain multiple columns. The clustering columns are created identical as the compound primary key [6]. An example of how this key can be created can be seen in Listing 2.3.

```
CREATE TABLE userposts (
  user_id int,
  post_id int,
  content text,
  post_date timestamp,
  PRIMARY KEY ((user_id, post_id), post_date)
);
```

Listing 2.3: Create table with a composite partition key

In this example, if the user_id is the same for multiple columns, but post_id is different, the columns will be partitioned to different nodes. However, if user_id and post_id is the same, the columns will be on the same node [6].

Secondary Index

A secondary index in Cassandra makes it possible to retrieve data using other attributes than the partition key (described above). With an index, it is possible to have fast and efficient lookup when matching data based on a condition. An index can be created as shown in Listing 2.4.

```
CREATE INDEX ON books(author);
```

Listing 2.4: Create a secondary index on a column in a column family

This statement will create a secondary index on the column author in column family books. However, a secondary index should not always be used. For example, an index should not be created on the book title. A query against Cassandra will return a small number of results (assuming there are many books in the column family). There should neither be created indexes in column families where there are used counter columns or columns that are often updated/deleted. Besides this, indexes should not be used if a query is used to look for a row in a large partition. However, it is possible to this if the query also narrows down the search area (for example by an id) [6].

2.7.3 Cassandra Query Language

Cassandra Query Language (CQL) is the query language for Cassandra and is similar to SQL (described in Chapter 2.5.3). CQL does not support join operations

or subquries like SQL does. However, Cassandra makes it possible to denormalize the data through features like collections and clustering. It is possible to use CQL by either start up the interface, named cqlsh or use a driver to access Cassandra (DataStax has multiple drivers, including a Java driver). All information in this chapter is based on the DataStax documentation for CQL. [6].

A small example is shown in this chapter. This example shows how some basic operations work in CQL. In a relational DBMS (described in Chapter 2.5), it is possible to have separate tables and define foreign keys (based on the id in the other table) in one table to join the data when queried. However, in Cassandra the data is denormalized to one table (column family). This example involves books and the authors of them. First, we need to create a keyspace that is a namespace for the cluster. It is possible to set the name, replication strategy, replication factor and support for durable writes. Let's say we only have one data center and with only one node. We, therefore, set the replication factor to one and uses the SimpleStrategy described above. The durable write option is not defined (it is possible to set it to false and skip the writes to commit log) and it is, therefore, true. It should also not be used when SimpleStrategy is used. An example statement is shown in Listing 2.5. This statement is used to create a keyspace in Cassandra.

```
CREATE KEYSPACE bookKeyspace WITH REPLICATION = { 'class' : '
    SimpleStrategy', 'replication_factor' : 1 };
```

Listing 2.5: Create keyspace statement in CQL

This statement only needs to be executed the first time when the keyspace is initialized. Before creating any column families in this keyspace, a use statement must be executed. This is shown in Listing 2.6.

```
USE bookKeyspace;
```

Listing 2.6: Use keyspace statement in CQL

The next step is to create a column family in Cassandra. An example of how a column family can be created can be seen in Listing 2.7.

```
CREATE TABLE book_authors (
  bookauthor_id int,
  author_id int,
  book_id int,
  author_name text,
  book_title text,
  book_date timestamp,
  book_category text,
  popularity int,
  PRIMARY KEY (bookwriter_id, popularity)
);
```

Listing 2.7: Create table statement in CQL

In this column family, a compound primary key (described in Chapter 2.7.2) is used where bookauthor_id column is the partition key, and the popularity column is a clustering column. Besides this, it should be possible efficiently to execute a query with a condition on the author name. A secondary index is therefore created on this column as shown in Listing 2.8.

```
CREATE INDEX ON book_authors(author_name);
```

Listing 2.8: Create index statement in CQL

An insert statement is created to insert data to this column family. An example of this type of statement can be seen in Listing 2.9 (assuming a popularity value of 0 are the most popular books).

```
INSERT INTO book_authors
  (bookauthor_id, author_id, book_id, author_name, book_title,
       book_date, book_category, popularity)
VALUES
  (1, 33, 46, George R. R. Martin, A Game of Thrones,
      06.08.1996, Fantasy, 0);
```

Listing 2.9: Insert statement in CQL

When some data has been inserted to the column family, different queries can be executed on Cassandra. A simple query is shown in Listing 2.10.

```
SELECT *
FROM book_authors;
```

Listing 2.10: Simple select query in CQL

This query will return all rows from the column family book_authors. If the

number of rows is larger than ten thousand, only the first ten thousand rows will be returned. However, it is possible to specify the limit in a query by the use of the keyword LIMIT. A query can be expressed as shown in Listing 2.11 to find all the books that have the author George R. R. Martin.

```
SELECT *
FROM book_authors
WHERE author_name = 'George R. R. Martin';
```

Listing 2.11: Select query with one where relation in CQL

There is also possible to count the number of rows as done in SQL. A simple example of this is shown in Listing 2.12.

```
SELECT COUNT(*)
FROM book_authors;
```

Listing 2.12: Select query with count in CQL

This query will return the number of rows that matches this query. Besides the different commands shown above, there is possible to update, drop, alter and truncate a column family. It is also possible to define different security options for users and list the different users/permissions. For a full overview of the different CQL commands in Cassandra, a look at the DataStax documentation for CQL is recommended [6].

2.8 Other NoSQL stores

Some other NoSQL stores are also briefly described in this chapter. Since Cassandra is explained in Chapter 2.7, no other extensible record stores are presented here. Two of the most well-known document and key-value stores are described in this chapter. These are MongoDB and Redis, respectively.

2.8.1 MongoDB

MongoDB [13] is a document store defined as a NoSQL database. MongoDB is written in C++ and developed by 10gen. It is also open source as Cassandra and Redis. Master-Slave replication is supported by MongoDB [25, 21]. The master can write and read data. A slave can only read data. If a master node fails, the slave node that has the most recent data is replaced as the master [21]. The replication is also asynchronous, which means that the updates are not spread instant, and some updates may be lost during a crash. However, this gives the advantage of high-performance [25, 21].

The data in MongoDB is stored as BSON (binary JSON-similar format). It supports integers, strings, booleans, dates and binaries [25, 21]. As Cattell [25]

says, MongoDB is lockless. However, he said this in 2010 and version 2.2 that came after this, introduces locks to ensure consistency of data. Besides this, indexes are supported and used to increase performance (as relational DBMS). The structure used by these indexes are a B-tree. Read, create, delete and update operations are also supported in MongoDB [21].

2.8.2 Redis

Redis [16] is written in C and is licensed under the BSD license as an open source project. However, it started with only one person. Redis is a key-value store where it is possible to use operations like insert, delete, and lookup. There is also support for sets and lists, not only blobs or strings. In Redis, the client does the distributed hashing to servers and each client is connected with the use of a library. This library contains a wire protocol to the servers, and the servers have the data in memory. However, the data can also be copied to disk (in case of shutdown). There is support for atomic updates in Redis. This is possible with the use of locking [25, 33].

It has been shown that Redis can make one hundred thousand requests per second with the use of the memory [25, 33]. However, Redis has a limit on the physical memory, and it will therefore execute best on smaller amounts of data [33].

Chapter 3

Related work

In this chapter, solutions related to implementation of join in NoSQL are investigated. The purpose of this chapter is to show that the join operation in or on NoSQL systems is investigated.

3.1 Equijoin in a Ring Architecture Key/Value Database

Wang et al. [49] have proposed an approach for executing equijoins on a ring architecture. As Wang et al. [49] mention, different solutions to perform a join on the master-slave architecture exists with the use of MapReduce, but not for the ring architecture. Their proposal for an efficient parallel execution of equijoin on a ring architecture without MapReduce is the following:

1. A Column Value Index (CVI). The consistency hash algorithm is utilized to store the CVI on the cluster.
2. Build a Memory Index(MI) to improve efficiency. This memory index is also used to avoid/reduce useless disk access.
3. A Pre-Join Table Generator (PTJG) uses the CVI and MI to process equijoin tasks on the ring architecture.

This implementation makes use of the column family data model that is illustrated in Figure 3.1. With this data model, three join conditions are represented. These are the following, equality of two row keys, equality of one row key against a column value (for one column key) and equality of two column values (for two different column keys). Based on this information, the PTJG algorithm is proposed, and this contains three phases:

1. Build a Column Value Index.
2. Build a Memory Index.

3. Accomplish the equijoin and generate the join result.

RowKey	Columns		
	ColumnKey	ColumnValue	Timestamp
k_1	c_1	v_1	t
	c_2	v_2	t
k_2	c_1	v_3	t
	c_2	v_4	t

Figure 3.1: ColumnFamily data model [49]

3.2 UnityJDBC

Unity is a generalizable SQL query interface for both relational and NoSQL systems described by Lawrence [37]. This system translates SQL queries to the underlying API of the data sources (for example, MySQL and MongoDB). Unity allows queries to span over multiple sources where each source will use their query engine to perform joins across the different sources. It is therefore defined as an integration and virtualization system.

The architecture of unity can be seen in Figure 3.2. This architecture consists of one SQL query parser that converts an SQL query into a parse tree and validates the query. This query parser supports standard SQL-92 syntax for SELECT, INSERT, UPDATE, and DELETE statements including inner and outer joins. This parse tree is sent to the query translator that converts it into a relational operator tree that consists of selection, projection, grouping and join operators. Besides this, it also validates field and table names. The join ordering is found in the optimizer with the use of the parse tree. It also discovers which parts of the query plan that should be executed on individual sources. Lawrence [37] describes different techniques to perform a join between multiple sources:

1. Push down filers: Selection operators are sent to the data source for execution.

2. Join ordering: The ordering of the join is found by using a cost based optimizer.

3. Push-down, staged joins: If a join is across two systems, the result from the first system is used to modify the query sent to the second system.

After the execution plan is generated, the execution engine takes over. It interacts with the data sources to submit queries and retrieve results and then perform any additional operations.

Experimental results for Unity shows that the performance is a bit slower and that the overhead is minimal in the SQL translation process. Further work is benchmarking the performance of other NoSQL systems, like Cassandra.

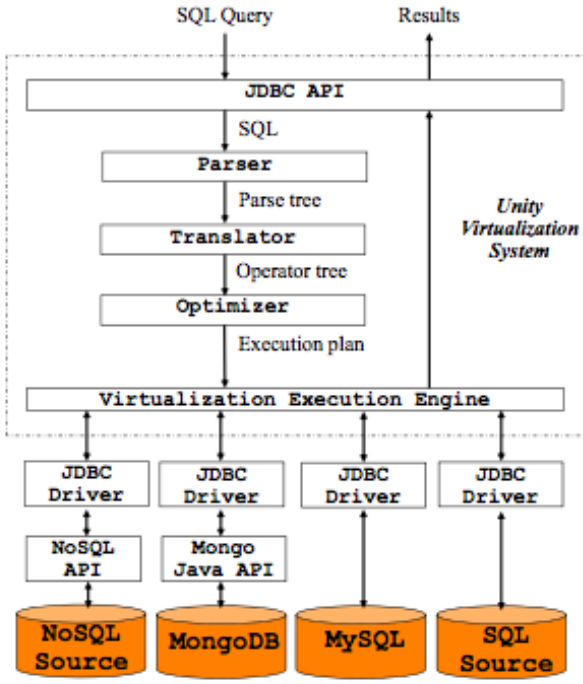


Figure 3.2: Unity architecture [37]

3.3 SimpleSQL

Calil [24] proposes a relational layer over Amazon SimpleDB, called SimpleSQL. It has an SQL interface for performing operations on SimpleDB where it is storage and operation transparency.

The four standard operations known from SQL are supported, which are: SELECT, UPDATE, DELETE and INSERT. A select statement in SimpleSQL can contain join operations. When a join operation is executed, each attribute must have the format "table.attribute".

When a command is executed on SimpleSQL, the first step is to decompose the SQL command and convert it to SimpleDB syntax. Once converted, is the command converted to a SimpleDB REST method call. The architecture of SimpleSQL is shown in Figure 3.3.

The results are returned to SimpleSQL after processing in SimpleDB. Evaluation of SimpleSQL shows a slight increase in processing overhead compared to pure SimpleDB. The processing time for select statements shows a 40% increase, but this was expected since SimpleSQL has to process the data retrieved from the

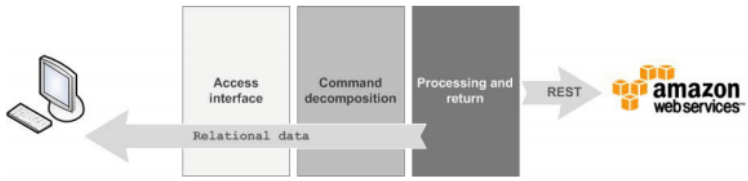


Figure 3.3: Architecture for SimpleSQL [24]

cloud.

3.4 CloudTPS

Wei et al. [50] present join query support in CloudTPS. CloudTPS is a middleware layer. This layer is between the web application and the corresponding cloud data source (BigTable, SimpleDB and Cassandra). CloudTPS supports one type of join queries. This is equijoins where foreign keys are used. This is also known as a foreign-key equijoin (a relationship between two tables where a primary key is equal a foreign key). This design chose is done because this is one of the most common types of join.

The data model that CloudTPS defines is a collection of tables. Each table in CloudTPS contains a set of records. These records then again contain one primary key and an arbitrary number of attribute-value pairs. The foreign keys are attributes that refer to a primary key in another table (or the same).

A Java client-side library is used by web applications to access the API of CloudTPS. Join queries can then be sent to this API. Each join query in CloudTPS is then a collection of JoinTable (the tables in the join) and JoinEdge (the relation between two tables) objects in Java.

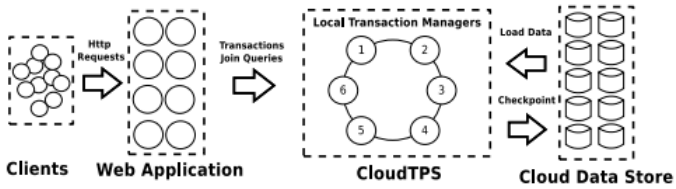


Figure 3.4: System model for CloudTPS [50]

As seen in Figure 3.4, each client can send an HTTP request to a web application and this uses CloudTPS to send the queries and transactions. There are multiple Local Transaction Managers (LTMs) in CloudTPS. Each of the LTMs is responsible for a subset of all data items. When CloudTPS is used by an application, the LTM

that is requested is referred to as the coordinator. If some data is needed and this is missing from the main memory of the LTM, the LTM that holds this data will load it from the cloud data store. CloudTPS was compared against PostgreSQL, and Wei et al. [50] shows that it performs better than replicated PostgreSQL, when the amount of data is large.

Chapter 4

Design space

This chapter looks at how a join can be implemented in Cassandra. It will not focus on how the architecture can be designed because no design process was needed to start the implementation. Cassandra is already designed, and the primary focus is to justify why the join functionality is implemented in the query language. Three different possibilities are described in Chapter 4.1. Certain limitations have been made for this thesis, and they are presented in Chapter 4.2. Each possibility has it pros and cons; these are highlighted in Chapter 4.3. These possibilities are also discussed in this chapter and a justification on why an implementation in the query language was chosen is presented.

4.1 Different roads - same destination

It is possible accomplish a join in different ways with the same result, hence different roads. In this chapter, three different possibilities that could have been used to accomplish a join operation in Cassandra are presented. These are materialized view (Chapter 4.1.1), application layer (Chapter 4.1.2) and query language (Chapter 4.1.3).

4.1.1 Materialized view

In SQL, a materialized view is a precomputed view where the query has been executed in advance and the result is stored in a table [41]. This is also used in decision support as Ramakrishnan and Gehrke describe [46] where fast response time is needed. With this technique, all results are already processed when a query is executed. This will enable faster retrieval of results. Different types of materialized views exist [45, 46]. The only type of materialized view considered here is the join view since the others (aggregation and projection) is out of scope.

In Cassandra, there is no way to create materialized views (Rabl and Jacobsen [45] presents an implementation of view creation in Cassandra) or join data be-

Id	Name	Age
5353	Ola Nordmann	50
7154	Kari Hansen	63
7235	Rolf Kittelsen	42
9476	Jens Jensen	38

Table 4.1: Example column family for professor

tween column families in the query language. However, it is possible to manually denormalize the column families and create a new column family. Planning your data model, so each column family answers a query, helps Cassandra because the data for one query is gathered in one column family [15]. This will give less disk seeking and network traffic, but increases the use of disk space (which is cheap).

In an SQL design, the normal forms are used to give guidance on how good design the relation schema has [46] and normalize the database if the design needs to be better. This is not the case in Cassandra, where it is more important to look at how a query can be answered with only one column family.

The denormalized column family must be maintained somehow. A straightforward approach to this problem could be to update the column family when one of the underlying column families are changed. This can be managed in the application layer where a change is registered. All rows that are affected in the denormalized column family is inserted, deleted or modified.

Another method is only to have the denormalized column families. All reads and writes are done on this denormalized column family. This means that only one column family has to be updated. With this design (denormalized), no base column families are required, and extra maintenance is not needed. All column families (if it was normalized) from the biggest join that is needed is stored together to avoid multiple column families that contain the same values. With biggest, it is meant the number of column families. If there is a row where there is no match in the other column families (in a normalized design), the values from these column families are null. When a select statement is issued against the denormalized column family, only results not containing null is retrieved.

An example of a denormalization is that you have two column families, professor, and course. To get all courses that are thought by one professor, a denormalization of these two column families could be performed. The first column family is professor, and this can be seen in Table 4.1.

Since a professor can have multiple courses, each course has a professor id (it is assumed that there is only one professor per course). This column family is shown in Table 4.2.

To answer the query "Which courses does professor Ola Nordmann teach?", a denormalization of these two column families is executed, and the result can be seen in Table 4.3. This column family can be updated at the application level when something changes. This fits well with Cassandra, which is designed to handle high write throughput [36].

Courseid	Title	Professorid
TDT4186	Operating Systems	5353
TDT4117	Information Retrieval	5353
TDT4175	Information Systems	9476

Table 4.2: Example column family for course

Id	Name	Age	Courseid	Title	Professorid
5353	Ola Nordmann	50	TDT4186	Operating Systems	5353
5353	Ola Nordmann	50	TDT4117	Information Retrieval	5353
9476	Jens Jensen	38	TDT4175	Information Systems	9476

Table 4.3: A denormalization of professor and course

4.1.2 Application layer

A possible way to join different column families in Cassandra is to perform it on the application layer. Examples of this type of join are presented in UnityJDBC [37] and SimpleSQL [24]. Both solutions are presented in Chapter 3, where they perform joins between tables on the application layer.

For this thesis, a possible design for this type of join design would be to create an application layer where it is possible to submit CQL queries. If the query does not contain a join, it is forwarded to Cassandra and the result is returned to the user through the application layer. Otherwise, a preparation has to be done before multiple select queries can be executed on Cassandra. A possible design for this can be seen in Figure 4.1. Cassandra data store refers to the memtables and SSTables (described in Chapter 2.7.1) for the column families in the join.

During join preparation, multiple select queries have to be generated, where all rows for one column family are fetched. These rows can then be used to create new select statements with a where clause. This where clause will contain relations on the correct column with values from the retrieved rows. If it is an N-Way-Join, one or more column families are reused in the relations. The result from these can be stored temporarily and used again without needing to access Cassandra. A column family should only need to be read once.

4.1.3 Query language

By query language, it is meant that the join can be written in CQL and executed in Cassandra. No materialized views or join in the application layer is needed.

Today, SQL supports join in the query language. For example, a join between the two tables (called column families in Cassandra.) in Table 4.1 and 4.2 can be expressed as shown in Listing 4.1.

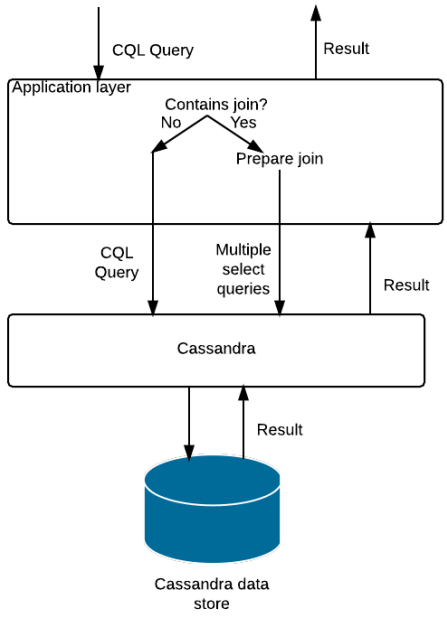


Figure 4.1: A possible design for join in the application layer

```
SELECT *
FROM   Professor P, Course C
WHERE  P.id = C.professorid;
```

Listing 4.1: Traditional style of writing join queries

This is the traditional way of writing join queries in SQL, called theta style [35]. Another SQL style, is the ANSI style [35, 31]. An example of this type is shown in Listing 4.2.

```
SELECT *
FROM   Professor P JOIN Course C ON P.id = C.professorid;
```

Listing 4.2: ANSI style of writing join queries

This will produce the same results as the traditional way. The difference is that the join is explicitly stated in the from clause [31] and not in the where clause as the traditional style.

In Cassandra, it is no support for this type of complex queries. To support a join through the query language in Cassandra, a change in the implementation has to be done. The design of this possibility is based on the query processing steps,

described in Chapter 2.1. These steps are as following:

1. Parsing
2. Optimization
3. Code generation
4. Execution

These steps may differ a bit from the steps in Cassandra. The select statement that is already implemented in Cassandra is the model for how the join statement should be designed. It is three steps that can be found in this class, and these are:

1. Parsing
2. Prepare
3. Execution

These three steps are almost the same as the four steps for query processing. For the join in the query language, the prepare step will contain both the optimization and code generation step.

The first step is the parser, that has to be changed to support joins. Only the traditional way of writing queries (also called theta style), is considered for this design possibility. The join is separated from the where clause to a new clause called JOINON. This separation is done to avoid confusion between new functionality and already implemented functionality. The SELECT clause must also be changed to support a projection on different columns in different column families. A new clause for this is also needed, called SELECTJOIN. The FROM clause must also be changed. However, this change is minimal, so no new clause is required here. A list of column families must be possible to add here since standard Cassandra only supports one column family per query. The WHERE clause is also almost unchanged, and the only difference will be that each column must be represented with the column family. An example of this type of query is shown in Listing 4.3.

```
SELECTJOIN *
FROM       professor, course
JOINON     professor.id = course.professorid;
```

Listing 4.3: Possible layout for a join query

This is a query where the where clause is not used. A query that contains a where clause can be written like Listing 4.4.

```
SELECTJOIN *
FROM       professor, course
WHERE      course.courseid = 'TDT4175'
JOINON     professor.id = course.professorid;
```

Listing 4.4: Possible layout for a join query with where relations

Both queries, executes a join between the two column families on the columns id and professorid. However, the latter query will only return the rows that have TDT4175 as their value in the column courseid. The first query will return all rows from the join. There should also be possible only to select some columns as the result. An example can be seen in Listing 4.5.

```
SELECTJOIN professor.name, course.title
FROM       professor, course
JOINON     professor.id = course.professorid;
```

Listing 4.5: Possible layout for a join query with projections

All rows that came through the join will be returned. However, only two columns will be returned. The examples above shows how a join could be written in the query language of Cassandra.

After the parsing step has been executed, the query must be prepared. This will involve optimization and code generation. The first part is the optimization. This step will discover which column families that are biggest and create a plan for execution for each N-Way-Join. This plan is based on query optimization described in Chapter 2.1.1. However, it only finds one plan and does not enumerate different left deep plans. It is based on that there must be a connection between each 2-Way-Join and reducing volume early if there is possible (for example with the use of where relations). This plan will be used during the execution to avoid unnecessary reads from the disk.

In the current implementation of Cassandra, the raw select statement is prepared and returned to the query processor (this class makes use of the different statements supported in Cassandra). This can be seen as the code generation in Cassandra, and the same operation must be done with the join statement. Besides this, each join relation generates a join restriction that is stored in the prepared join statement. This also happen in the current implementation with the relations in the where clause. However, if this can be classified as code generation is not critical and it differs a lot from the code generation in System-R (see Chapter 2.1).

During execution, different approaches (described in Chapter 2.3) for join can be chosen. In this design, the nested loop join algorithm is used to execute the join. This is the simplest join algorithm [46] of the three described in Chapter 2.3 and is therefore used in this design.

The design of the nested loop join may be different in Cassandra because some modifications may be done to make it work with existing implementation. If there

exist methods or lists that can be used by the join implementation, the current code base of Cassandra is something that should be used for this possibility. It is unnecessary to implement new functionality if it already exists.

4.2 Limitations

Some limitations have been made concerning the join types and algorithms including which index types that are supported in a join (for Cassandra). These are presented in this chapter.

4.2.1 Only primary keys and secondary indexes

Only single primary keys and secondary indexes (explained in section 2.7.2) are supported when a join statement is issued. The two other types of keys described, composite partition and compound primary key is not supported. In this first prototype, it was only important to show that join could be executed in Cassandra. However, this should be supported if the join operation is used and the data is distributed over multiple nodes.

4.2.2 Only equijoin

Only equijoin will be considered, since this is by far the most used type of join in web applications as stated by Wei et al. [50]. Other types of join, like semi join, natural join and outer join (described in Chapter 2.3.1) are out of scope.

4.2.3 Only index nested loop join

One of the join algorithms described in Chapter 2.3 must be used to implement the join functionality in Cassandra. Originally, the simple nested loop join was thought of as the algorithm to use. However, the design of Cassandra made it easier to implement the index nested loop join algorithm. Sort-merge and hash join are excluded since there exists a time limit on this thesis, and these algorithms would require more adjustments of the underlying architecture to implement.

4.2.4 Locally supported

This version is only supported and tested at a single node. Implementing and testing a join operation on multiple nodes are out of scope and should be further work.

4.3 Discussion

Query language (described in Chapter 4.1.3) is the one possibility that was chosen for this thesis. In this chapter, this possibility is discussed with the two other possibilities described in Chapter 4.1. These three options are compared with

speed, memory, redundancy, and maintenance in mind. A justification for why the query language was chosen is presented in Chapter 4.3.5.

4.3.1 Speed

The first possibility presented was materialized views. This is most likely the possibility that is the fastest of these three. The reason for this statement is that no join has to be done by the user and only one column family has to be read to answer a query. As explained in Chapter 4.1.1, Cassandra performs best when an answer for a query is stored in one column family (denormalized). This gives less disk seeking and network traffic, which is read in this case. Cassandra does not handle reads as good as it does with writes [40, 36]. However, it still has good read speeds. But, the materialized view option will have fewer disk reads than the two other options. This will result in a slower execution if a join is performed in the application layer or query language. However, it is assumed that the query language will execute faster than the application layer. This is because the network traffic for the application most likely will be higher. For example, there are two column families A and B. Column family A has 1,000,000 rows and B has 500,000 rows. The result of a join between A and B is 300,000 rows. In the application layer, column family B must be read in its entirely first, which implies that 500,000 rows must be transmitted over a network. After this, as explained for the application layer, each of these 500,000 rows is iterated through and a select statement is sent against Cassandra (gives 500,000 more reads). Each select statement will return 0 or more rows that are combined the row used to create the select statement (300,000 rows will be returned in total). In the query language, the back and forth operation is not needed because the complete result of the join is found before it is sent to a user. Only 300,000 rows must be sent over the network while the application layer will need to transmit 800,000 rows (500,000 from column family B and 300,000 from column family A). The application layer will also need to prepare and interpret a query as the query language. So both the application layer and query language must almost do the same operations like parsing, preparation and iterating through each row. The biggest difference is that the number of rows from the query language will be significantly smaller. With this example, 300,000 rows must also be returned from the materialized view. However, the join is already executed, and no operation is needed to combine the results as the query language.

4.3.2 Memory

If the join is executed in the application layer, results during the join process are stored in the virtual memory. If the size of the result is too big, it may exceed the virtual memory size and the application layer may crash. With this functionality implemented in the query language, a possibility is to store the temporary results on disk. These results can then be used when they are needed.

4.3.3 Redundancy

Since the materialized view is denormalized, some data will be stored redundantly. In the query language or application layer, this is not a problem. In the materialized view, this requires maintenance of the column family. Since there is no implementation of this in Cassandra, this requires the developer of the application that uses Cassandra to handle this.

4.3.4 Maintenance

Materialized views and the application layer requires more actions to complete a join than the query language does. The materialized view requires the user to create and maintain the view. An implementation in the application layer can require the user to implement some functionality for the join. The user may use a library for the join functionality and this does not require much more work for the user. However, it still requires more work than the query language because the library must be included in the application code. Besides this, the library may break if something is changed in Cassandra if the library is developed by a third-part. Then the user has to manually change the library to be compliant with Cassandra. Instead of letting the user do the work, the query language makes it possible for the user to send queries that contain joins to Cassandra. No maintenance or extra implementation is needed to execute the join.

4.3.5 Possibility implemented

The query language is the one possibility that is implemented for this thesis, and the implementation is described in Chapter 5. Materialized views would probably be faster than the query language, but it will have more maintenance and redundancy. Creating a join functionality in the application layer is a solution that needs to transmit more data over the network than the query language. Besides this, trying to implement a join functionality into CQL is interesting to look at because this is how SQL works, where it is possible to write queries that contain joins.

Join functionality in the application layer has already been investigated [37, 24]. It has also been investigated on the implementation of materialized views in Cassandra and CQL [45]. However, no work has been found on join implementation in Cassandra (query language) after searching with best efforts. With this information, it is known that the application layer and materialized views are possible. Therefore was a join implementation in Cassandra, which support joins defined in the query language, chosen as the subject for this thesis.

Chapter 5

Implementation

This chapter will explain how equijoin was implemented in the source code of Cassandra. All sections and subsections are chronologically ordered in the way they occur when a user executes a query against Cassandra. Chapter 5.1 looks at what tools that were used during the implementation period and how the code is organized. Chapter 5.2 and 5.3 looks at how the actual implementation for join was done. Chapter 5.4 describes how column families that are not in any join are executed and Chapter 5.5 explains how all the results are merged. Chapter 5.6 will look at an example execution with the purpose of showing what happens in Cassandra.

The reader of this chapter should also know that the term "reading from the data store" is referred to as reading from both memtables and SSTables. Besides this, selections refer to projections in this chapter because the object referring to projections in Cassandra is named "Selection". The term selection known from relation algebra is also avoided because of this. Instead, terms like where relations and restrictions are used.

5.1 Organization and explanation of the code

This implementation is done in Cassandra source code for version 2.0 that can be found on the GitHub page for Cassandra [22]. All code implemented in this thesis tries to use the same name convention as Cassandra when it is possible. However, sometimes the naming convention is a bit different because it does not exist any similar functionality in the source code of Cassandra.

GitHub [7] is used as a tool for source code management and as a backup solution for the implementation. This tool also gave the possibility to create stable versions and experiment with new functionality without destroying the existing implementation. All code implemented during this thesis can be found on GitHub [10]. Most of the work in this thesis was done in Java, but ANTLR version 3 [1] was used for the parser and lexer.

Further, the tool used for building the project was Apache Ant [2]. Some of the main commands used for this implementation are (see Cassandra build file at GitHub [22] for documentation on all ant commands):

ant build	Compiles all Cassandra classes.
ant artifacts	Generate artifacts for all Cassandra classes.
ant test	Executes all unit tests.
ant clean	Removes all locally created artifacts

The most used command was ant build. This command was used each time something in the code was changed. After this command was executed, the server could be started up, and users could connect to the server. If the project was pulled from the GitHub directory to a new computer, ant clean had to be run because locally created artifacts had to be removed. Ant build could then be run after this. Ant test was used to check that this implementation did not destroy any existing code. During the implementation of join in Cassandra, some documentation was used to understand the existing code. The command for retrieving this information was ant artifacts. Due to the lack of documentation in Cassandra, there was not so much information to extract here. To understand how the existing implementation worked, printing and logging at different places in the code was done. This gave some information on how the flow was and what was done. Trying to change different variables in the code and see if it failed was also done.

Since there already existed an organization of the code, no changes were done here. All statements are under the cql3 package and then under the statements package. The join statement was also implemented in this package. All other classes created was saved under packages where the other classes were similar (in means of what they do). Most of the implementation was done in the package cql3.

5.2 Parser and prepare

This chapter describes how ANTLR was used to implement a parser that supports join statements in Cassandra and how the prepare step was implemented. The final result of this chapter is a prepared statement that will be used by the query processor in Cassandra to execute the join statement.

5.2.1 Selectjoin clause

Selectjoin is similar to the select clause found in standard CQL. A join select can contain multiple join selectors separated by a comma, shown in Figure 5.1. This figure also shows that it is possible only to write '*' (asterisk). This will select all columns from all column families in the query at a later stage. JoinSelector returns a raw join selector and join select clause returns a list of raw join selectors. This list is returned only if the user has selected one or more columns and not when the asterisk is used.

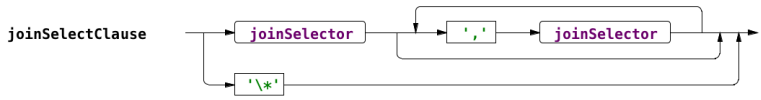


Figure 5.1: Syntax diagram for the select clause

If a query contains selections on columns, there are some differences from a standard CQL query. Each column must be represented on the form "column-family.column-name" as the syntax diagram in Figure 5.2 shows. A representation in this form is chosen because some column families may have the same column names. Join selectable will set the column family and column.

Figure 5.2: Syntax diagram for the join selector

5.2.2 From clause

Since a join statement must support the use of multiple column families in one query, a rule in ANTLR was created so multiple column families could be used. Each column family is separated by a comma, shown in Figure 5.3. A list of column family names is returned from the from clause (named "columnFamiliesName" in Figure 5.3).

Figure 5.3: Syntax diagram for the from clause

5.2.3 Standard where clause

A standard where clause should be possible in this implementation. As in a select in Chapter 5.2.1, each column must be represented with the column family. Only single column relations will be allowed. An extra method for storing the column family name was implemented in the single column relation class. This information is used by the prepare step. As seen in Figure 5.4, a where clause can contain one or more single column relations which is found in "joinSingleColumnRelation".

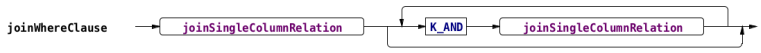


Figure 5.4: Syntax diagram for the where clause

5.2.4 Joinon clause

In a select statement, the where clause may contain one or more relations, for example, "WHERE id=6". Based on the input, the relations are either single column or multi column. A single column relation is a standard relation, for example, "id=3". A multi column relation is a relation that spans over multiple columns, for example "(a, b, c) IN ((1, 2, 3), (4, 5, 6), (7, 8, 9))". The implementation of equijoin in Cassandra is a single column relation on the form "columnfamily1.id=columnfamily2.id", but it is not called a single column relation. The relation is called join relation because a relation in a join requires other variables than a standard single column relation.

Figure 5.5 shows that a joinon clause can have multiple relations separated by a "K_AND" (this is and). Each relation is created in equijoin which returns a join relation. A query that contains a join will use a join relation that contains a

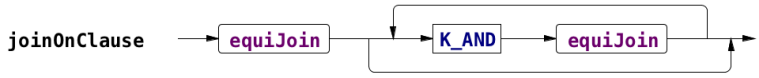


Figure 5.5: Syntax diagram for the joinon clause

left and right column family. A left and right column identifier is also stored in the relation. As explained in Chapter 5.2.1, each column is represented with the column family. This can be seen from Figure 5.6. EachSideOfJoin checks each side of the equal operator. By the use of a boolean, correct column family and column name is stored either on the left side or right side. This method will also check that only letters allowed by Cassandra is used. A list of join relations will be created

Figure 5.6: Syntax diagram for equijoin

and used by the prepare step in the join statement.

5.2.5 Other clauses

Clauses like ORDER BY or GROUP BY exists in standard CQL. These clauses are not implemented in the join statement because it is not important to support these operations to show that a join statement can work in Cassandra. Aggregate functions like MIN and MAX are neither implemented.

5.2.6 Example query

An example can be shown with four column families that should be joined. The example case is: there are many persons in Norway, where many persons own one or more houses (multiple persons can own a house). These houses are located by

Column family	Primary key	Secondary index	Attributes
Person	personid	–	givenname, familyname and adress
HouseOwner	id	houseid and personid	–
House	houseid	postalcode	color, size
PostalCode	postalcode	–	cityname

Table 5.1: Example column families for person, houseowner, house, and postalcode.

a postal code and a city. The different column families will contain a primary key. It may contain one or more secondary indexes or attributes as shown in Table 5.1. Four create table statements must be executed first and they are listed in Listing 5.1.

```
CREATE TABLE person (
  personid int PRIMARY KEY,
  givenname text,
  familyname text,
  adress text
);

CREATE TABLE houseowner (
  id int PRIMARY KEY,
  houseid int,
  personid int
);

CREATE TABLE house (
  houseid int PRIMARY KEY,
  postalcode int,
  color text,
  size text
);

CREATE TABLE postalcode (
  postalcode text PRIMARY KEY,
  cityname text
);
```

Listing 5.1: Create table statements for column families person, houseowner, house and postalcode in CQL

When each column family has been created, all columns that will be used in the join must either be a primary key or secondary index. A secondary index is created as shown in Listing 5.2.

```
CREATE INDEX ON columnfamily (column);
```

Listing 5.2: Create index on column in CQL

After the secondary index is created, some test data is inserted into the column families. It is required that the whole column family name is written and not with an alias since this is not supported by join statements. A query that contains a join looks like Listing 5.3.

```
SELECTJOIN *
FROM person, houseowner, house, postalcode
JOINON person.personid=houseowner.personid AND houseowner.
    houseid=house.houseid AND house.postalcode=postalcode.
    postalcode;
```

Listing 5.3: A join query in CQL

This query returns all four column families with all columns with all rows that match the restrictions in the joinon clause. A query containing a join can also include different columns in the selectjoin clause. A query with a projections and join is written as shown in Listing 5.4

```
SELECTJOIN person.familyname, person.givenname, postalcode.
    postalcode, postalcode.cityname
FROM person, houseowner, house, postalcode
JOINON person.personid=houseowner.personid AND houseowner.
    houseid=house.houseid AND house.postalcode=postalcode.
    postalcode;
```

Listing 5.4: A join query with projections in CQL

This query returns all queries as the first example, but only four columns. A join query can also contain single column relations. An example of this is shown in Listing 5.5.

```
SELECTJOIN *
FROM person, houseowner, house, postalcode
WHERE postalcode.postalcode='7030'
JOINON person.personid=houseowner.personid AND houseowner.
    houseid=house.houseid AND house.postalcode=postalcode.
    postalcode;
```

Listing 5.5: A join query with where relations in CQL

This query returns all rows where the column postalcode in column family postalcode is 7030. All other rows are excluded from the result.

5.2.7 Selection

When a user executes a join statement, all column families that the user has selected have to be validated by Cassandra. This validation generates a column family definition (CFDefinition) that holds metadata on a column family preprocessed for use by CQL queries.

In standard Cassandra, the select clause only contained the column identifiers since there was only possible to select from one column family per query. Since a join statement requires joins from multiple column families (at least two), is there a possibility that this can create a conflict if some of the column names are the same. To solve the potential problem with identical column names, each column name must have a column family represented with it, in the form "column-family.column-name". All columns from one column family in the selectjoin clause is stored in a hash map with the corresponding CFDefinition as the key and a join or simple selection object as the value. If there is one column that is used in the selectjoin clause, there will exist a join selection object for each column family in the join. However, all these objects will have an empty list of columns except the column family that has the column in the selectjoin clause. If it is used an asterisk in the selectjoin clause, all column families will have a simple selection object as the value in the hash map. This object is already an implemented class in the source code of Cassandra [22]. Under the execution, when the result is processed for each column family, the correct selector is found by looking up the CFDefinition in the hash map. If a column family does not have any columns in their join selection object, no columns are shown from this column family. However, all columns are used if the column family has a simple selection object in the hash map. As explained in Chapter 5.2.1, the parser ensures that all selections in the selectjoin clause are on the form "column-family.column-name" if not an asterisk is used.

5.2.8 Join restrictions

A standard restriction in Cassandra contains all conditions for a column in a select statement [22]. A restriction can either be a single column or multi column restriction.

A relation is converted to a restriction when the query has been parsed. If it is a single column or multi column restriction depends on if it was a single column or multi column relation. What type of relation that is used, is decided in the parser step, explained in Chapter 5.2.4. Since a join statement does not get the join value before one of the column families have been read, a specialized join restriction class is implemented. This restriction can store two column families and two columns.

When each join restriction is prepared, there are some rules for which columns that can be used in a restriction. Both columns in a restriction must be a primary key or secondary index (see Chapter 2.7.2). A reason for this choice is that an index provides a means to access data with the benefit of efficiency [6, 5]. If a column is indexed is found in the metadata for the column family.

As explained in Chapter 2.3.2, the smallest table (in Cassandra, the term table is referred to as a column family) is in the outer loop. To find the smallest column family in Cassandra, a method that checks if the left column family in the relation is bigger than the right column family is implemented. This implementation can be seen in Code 5.6. A method, called "getTotalDiskSpaceUsed" is used to decide if the left column family is biggest. This method returns the total disk space used by all SSTables that belongs to either left or right column family [22]. If left column family (in the relation) is biggest, this will be the left column family in the restriction and the right column family will be the right column family. Otherwise will the right column family be the left column family in the restriction and the left column family will be the right column family. In addition to containing this, information about the column types (if they are primary keys or secondary indexes) are stored for the left and right column family. The information found Code 5.6 and the type of column (primary key or secondary index) is used during the execution step.

Code 5.6 Check if the left column family in the relation is biggest

```
1  private boolean isLeftColumnFamilyBiggest(ColumnFamilyStore left,
       ColumnFamilyStore right){
2    return left.getTotalDiskSpaceUsed() > right.getTotalDiskSpaceUsed
         ();
3  }
```

After each relation is processed and transformed to a join restriction, it is added to a hash map. This hash map contains the column family as the key and a list of join restrictions for each key. Since a join restriction contains two column families, the join restriction will be added twice in the hash map (at different keys). This hash map is then used in Chapter 5.2.10 to find the ordering for each N-Way-Join.

5.2.9 Standard restrictions

Because this thesis looks into joins in Cassandra, a simple implementation of the standard where clause was implemented in this prototype. Only the equal operator is allowed in a relation. If a user tries to use something else, an exception will be

thrown. Greater or smaller than operators could also have been implemented, but for testing purposes, it is enough with the equal operator.

Since the parser only uses single column relations (see Chapter 5.2.3), all the single column relations are transformed into two types of single column restrictions. These types are either a join single column restriction or a modified standard single column restriction (which is a modified class already implemented in Cassandra). The reason for separating these types are because the usage is different in the implementation. If a relation is used on a column in a column family which is also used in the join, a join single column restriction created. This object contains a single column restriction, boolean variables about the column (if it is a primary key or secondary index) and the column name including the column family name. The single column restriction will contain the value that have been specified by the user. Only one column in a column family can be restricted if the column family is part of the join because this functionality is not important to show that a join is possible in Cassandra. All join single column restrictions are added to a hash map as a value where the key is the column family name. The column family name is also added to an array list that will be used for the join ordering (usage is described in Chapter 5.2.10).

If the column family is not part of the join, a standard modified single column restriction is created. By modified, it means that some additional variables have been added to the class "SingleColumnRestriction" found in Cassandra. These variables are the same boolean variables as in a join single column restriction. These restrictions are further used, either in the standard execution or the nested loop execution.

5.2.10 Find join order

When a query that contains join restrictions is executed by a user, the ordering of the different join restrictions has to be decided. This ordering is found because the number of column families read from the data store should be kept at a minimum.

When the join restrictions were transformed from join relations, a hash map was created. As seen in Figure 5.7, this hash map has a set of column family names that are the keys. Each column family name ($CF_1, CF_2, ..., CF_N$, where N is the number of column families in one N-Way-Join) has one corresponding array list that contains all join restrictions. Each restriction connected to a column family is represented as res-1, res-2, ..., res-R where R is the number of join restrictions.

If there exist a column family that is used in the where clause and belongs to a join restriction, it is beneficial to have this join restriction first in the join order. A lot of rows may be excluded at an early stage, and unnecessary iterations are avoided in the nested loop. The list, let's say L, created in Chapter 5.2.9, which contains all column family names that is used in the where clause and belongs to an N-Way-Join, is iterated over first. A temporary list, let's say TL, is created before the iteration starts. During the iteration, all column families found in the hash map (described above) that matches any column family name in list L, is added to the list TL. After this, all other column families are added to the list TL. This temporary list is then used when each N-Way-Join object is created.

An iteration over the temporary list, TL, is executed. For each column family in TL, a method is called and the N-Way-Join is created. All column families belonging to that N-Way-Join is added in this method. However, this only happens if the hash map contains a list for that column family with size bigger than zero. If the size is zero, it means that the column family has already been added to an N-Way-Join. In an N-Way-Join, each restriction must have a connection to at least one other restriction. This connection is represented by the use of column family names. With this connection, it is possible to start at one column family, iterate over each corresponding restriction. At each restriction, the second column family is checked. If there is only one restriction at the other column family, this must be the same restriction, and it is removed from the hash map. Another possibility is that the second column family contains more than one restriction. If this happens, the restriction is removed from the second column family and a new iteration of the restrictions found on the second column family is done in the same way as described above. This operation continues until all restrictions have been assigned to an N-Way-Join, and each N-Way-Join contains a list of join restrictions.

After the N-Way-Join is created, the ordering of the different restrictions has to be decided. An unordered and ordered list of all restrictions is maintained in each N-Way-Join. The unordered list has to be ordered so that only the first restriction has to read both column families from the data store. All other restrictions only need to read one column family from the data store. The ordered list is achieved by having a nested for loop where the same unordered list is used on the inner and outer for loop. A join restriction that contains a column family that is also used in the where clause will be first in the unordered list and the ordered list as well. However, it is only guaranteed that the first element of the unordered list will be the first element of the ordered list. In the outer loop, the first restriction is added to the ordered list. In the inner loop, all the join restrictions that matched on either the left or right column family in the outer loop is added to the ordered list. This process is repeated until all elements are added. Duplicates in the ordered list are avoided with a check that ensures that only one instance of the join restriction is added.

An example of this operation can be shown from Figure 5.7. This example does not have any column family that was used in the where clause and at the same time belongs to a join. The first column family that is read from the list L (described above) is CF_1 and the method creating the N-Way-Join is called. This column family has the corresponding restrictions Res-1, Res-2, and Res-3. CF_2 is the second column family, but it only contains a list of size one. Res-1 is therefore removed from CF_2, and the next restriction is Res-2. Res-2 also belongs to CF_4 and is removed from this list. Since CF_4 originally had a list of size two, Res-4 is read next. Res-4 has CF_5 as the second column family and Res-4 is removed from this list. Since this list has size one, it jumps back to Res-3. This restriction also has column family CF_3 which is size one. The operation is done and Res-1, Res-2, Res-3, and Res-4 belongs to the same N-Way-Join. This N-Way-Join have this list of restrictions after the operations is done: [Res-1, Res-2, Res-4, Res-3]. Since all column families have been added to an N-Way-Join, the next operation is to order

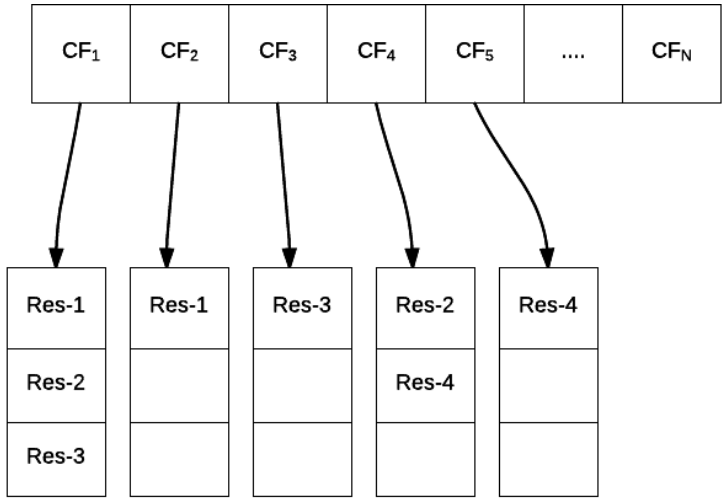


Figure 5.7: Hash map containing all column families with corresponding restrictions

the N-Way-Join to create a join tree. Res-1 is the first element in the list, and, therefore, the first restriction to be added to the list. Res-2 is then checked against Res-1 in the inner loop. CF_1 is a match and Res-2 is therefore the next element to be added. Res-3 is also added because of CF_1. In the outer loop, Res-2 and Res-3 will not be added again since they already are in the list. Res-4 will be the last restriction to be added. As shown in Figure 5.8, a join tree has been created. At the execution stage, this enables only one reading of a column family in the join restriction from the data store except the first.

5.2.11 Column families outside the join

When a query that contains both column families in an N-Way-Join and outside an N-Way-Join, two different lists are created. The first list is an N-Way-Join list containing each N-Way-Join described in Chapter 5.2.10 and the second list is all column families outside any N-Way-Join. When the join restriction is prepared, the column family name (both left and right side of the equijoin) is checked against hash map as seen in Code 5.7. This code uses the CFDefinition metadata to check if the hash map contains the column family name. If it contains the name, the insertion is removed from the hash map. After all column families appearing in the query is prepared, all the column families in any N-Way-Join is removed and the hash map only contains column families outside an N-Way-Join.

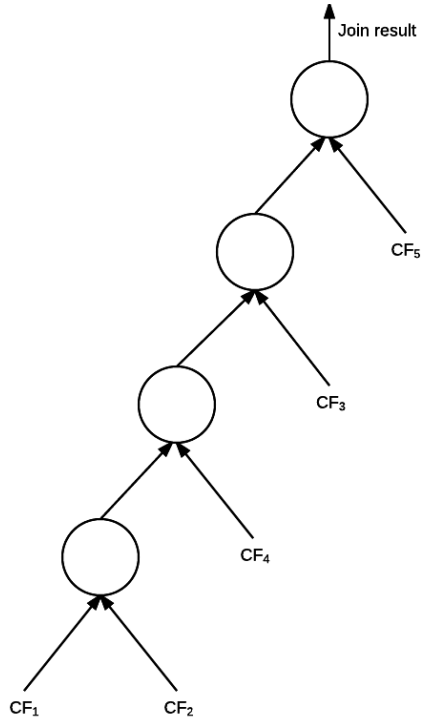


Figure 5.8: Join tree

Code 5.7 If check for column families outside a join

```
1  if(columnFamiliesOutsideJoin.containsKey(leftCdef.cfm.cfName)){
2    columnFamiliesOutsideJoin.remove(leftCdef.cfm.cfName);
3  }
4  if(columnFamiliesOutsideJoin.containsKey(rightCdef.cfm.cfName)){
5    columnFamiliesOutsideJoin.remove(rightCdef.cfm.cfName);
6  }
```

5.3 Execute join query

This chapter looks at how an N-Way-Join is executed. A list of join restrictions was ordered in each N-Way-Join as described in Chapter 5.2.10. This list is used to iterate through each join restriction and execute reads against the column families to create a final result for an N-Way-Join. All subsections of this section are ordered chronologically as they happen in the code.

5.3.1 Decide inner and outer column families

Originally, the right column family is the smallest and should, therefore, be in the outer loop, and the left column family should be the inner. This rule only applies to the first join restriction in the list and no column families from this join restriction are used in the where clause. The decision on the inner and outer column family can be separated in two sections:

(a) The join restriction is the first element in the list and, therefore, the first join restriction that is executed in the N-Way-Join.

(b) All other join restrictions are dependent on a join restriction that is already executed.

In case (a), the join restriction may include a column family that is also used in the where clause. If this is the case, this column family should be in the outer loop. This is because the size will most likely be significantly smaller than the other column family (in the join restriction). If neither the left or right column family are used in column family, the left column family is employed in the inner loop and the right column family is used in the outer loop. Three if checks are used to enforce this rule as shown in Code 5.8. In addition to this, a boolean variable is set to true and used in the outer loop.

If a join restriction is not the first element in the list, case (b), either left or right column family has been read earlier. This requires two if checks (shown in Code 5.8), to find out which column family that have been already read. If none is read earlier, an exception is thrown. In the first case, if the left column family has been read before, it will be the outer column family and the right column family will be the inner. In the second case, it will be the opposite. A boolean variable will also be set, but to false. This boolean variable is then used in the outer loop to decide if the column family should be fetched from the data store or not. Each join restriction is executed after this step.

Code 5.8 Discover which column families that are in the outer and inner loop

```
1  for (int i = 0; i<joinRestrictions.size(); i++){
2    //All elements in the list except the first
3    if(i != 0){
4      if(previousCfs.contains(joinRestrictions.get(i).joinFields().left
           .cfName)){
5        //Left column family in the outer loop
6      }
7      else if(previousCfs.contains(joinRestrictions.get(i).joinFields()
           .right.cfName)){
8        //Right column family in the outer loop
9      }
10     else{
11       throw new InvalidRequestException("Something wrong on nested
             loop with: " + bigColumnFamily.cfm.cfName + " and: " +
             smallColumnFamily.cfm.cfName);
12     }
13   }
14   //First element in the list, no results from prior CFs exists yet.
15   else{
16     if(columnFamiliesInWhereClause.contains(smallColumnName.cfName))
           {
17       //Left column family in the outer loop
18     }
19     else if(columnFamiliesInWhereClause.contains(bigColumnName.cfName
           )) {
20       //Right column family in the outer loop
21     }
22     else {
23       //Right column family in the outer loop
24     }
25   }
26 }
```

5.3.2 Big row

Big row is an object implemented in this prototype that contains one row from each column family in the N-Way-Join when the join is finished. These rows are stored in a hash map where the column family is the key, and the value is a row fetched from that column family. Each big row only contains one standard row per column family. The result is also stored in byte buffers in the same column order as the rows. For example, a 3-Way-Join has three restrictions as shown in Table 5.2, where each restriction is on two column families. Join restriction 1 is executed first. This execution retrieves M rows and creates M big rows. Each big row has a hash map containing one row from CF1 and one row for CF2. Only those rows that

have matching column values between CF1 and CF2 will be used by join restriction 2. This operation is repeated with join restriction 2 and 3. In the end, a set with big rows survived and is therefore used in the result. This implementation is used by the outer and inner loop implementation described in Chapter 5.3.3 and 5.3.7, respectively.

Join restriction 1		Join restriction 2		Join restriction 3	
CF1	CF2	CF2	CF3	CF1	CF5

Table 5.2: A set of join restrictions with corresponding column families (CF) in a 3-Way-Join.

5.3.3 Outer loop

Outer rows are the rows that are looped through in the outer loop in nested loop join. This nested loop algorithm can be defined as an index nested loop join (described in Chapter 2.3.2). It is defined as this type since, as described below, all inner rows are found in the outer loop and only processed with the outer rows in the inner loop. The outer rows will always be big row objects, which is explained in Chapter 5.3.2. If the outer column family has been read before, a boolean variable is set to true (all join restrictions in the N-Way-Join except the first). The boolean variable is set to false (first join restriction executed in the N-Way-Join) if it has not been read before. False will require a read of the outer column family from the data store. This boolean variable is set when the outer and inner column family in the loop is decided (described in Chapter 5.3.1). The boolean variable makes it possible to either retrieve old results or read the column family from the data store. In the first case, where the outer column family has been read earlier, the big rows that survived the previous join restriction are used. In the second case, if the outer column family is first, it has to be read from the data store. All rows from both column families in that join restriction will be read from the data store. The rows from the outer column family (also the smallest) are converted to big rows that will be used to remove rows that do not match any other rows in the inner column family (biggest). When the rows are converted to big rows, the method "hasSingleColRestriction" (explained in Chapter 5.3.6) is also performed if there exists a join single column restriction on the outer column family. Only the rows that match the restriction (from the where clause) are converted to big rows.

It is the row object for the outer column family that is going to be used in the join (a big row may contain one row from multiple column families). To retrieve this, a get operation on the hash map in the big row with the column family name is done, and the correct row is fetched, shown in Code 5.9. This row is then used to fetch the inner rows, which will be row objects. A command to find all rows are used in Code 5.9 (This method is described in Chapter 5.3.4). This command is sent to the method "ExecuteJoinRows" which returns all inner rows matching the join value. These rows are further processed in the

inner loop together with the outer rows which are described in Chapter 5.3.7.

Code 5.9 Use of get command method and finding all inner rows

```
1 Row row = outerRow.getRows().get(outerColumnFamily.cfm.cfName);
2 innerCommand = getCommand(joinRestriction,variables,
      innerColumnFamily,innerColumnName, row, outerColumnName);
3 innerRows = executeJoinRows(innerCommand, cl, new ArrayList<
      ByteBuffer>(), limit, System.currentTimeMillis());
```

5.3.4 Get correct command

Whenever a column family is read, three different read commands can be used based on the query type [29]. These are *SliceFromReadCommands*, *SliceByNamesReadCommands*, and *RangeSliceCommand*. A standard select statement in Cassandra only enables the user to select from one column family for one query, and it is, therefore, only one command per statement. A join statement needs to have multiple commands since there are two or more column families in a join. As shown in Code 5.10, if the column in a column family is a secondary index, a range command will be returned. Otherwise, a command for use on single primary keys are issued. The boolean variable "isKeyRange" will always be the same as "UsesSecondaryIndexing" in this implementation since this applies to all keys except the first key in the primary key. As explained in Chapter 4.2.1, only secondary indexes or single primary keys are allowed in this prototype. The join value (how the join value is found is explained in Chapter 5.3.5), which is used in the command is also fetched before this happens. This value is stored in the array list "temp" seen in Code 5.10. These commands are generated with the use of already implemented code (line 2 and 5 in Code 5.10).

Code 5.10 Returning correct command for join

```
1   if(isKeyRange || usesSecondaryIndexing){
2       return getRangeCommand(temp, joinRestriction, limit, System.
            currentTimeMillis(), columnFamily, column, false);
3   }
4   List<ReadCommand> commands = getSliceCommands(temp, limit, System.
        currentTimeMillis(), columnFamily, joinRestriction);
5   return commands == null ? null : new Pageable.ReadCommands(
        commands);
```

This command is used in the method "ExecuteJoinRows" (see Code 5.9) and the rows are read from this method. If the command is null, an empty list is returned from this method. If not, the rows are returned.

5.3.5 Column value

Code 5.11 shows how the value for a column in an outer row is found. This value is then used in Chapter 5.3.4 to create a command that will fetch the inner rows that matches the outer rows. There are two ways of finding the value of the column in the outer row. These options are:

1. The column is a primary key and the row key is returned (line 1-3).
2. A secondary index is used on the column, and the value needs to be found and returned (line 4-10).

If the column that is being checked is a primary key (option 1), it will return the row key. If the column is a primary key is checked with the boolean variables "isKeyRange" and "usesSecondaryIndexing". These variables are false if the column is a primary key. If the column is a secondary index (option 2), another approach is needed to get the value for a column in a row. Each column is stored in a row as a null prefix terminated string. This means that each column starts with a null, then the length of the column name, then the column name and it is ended with a null. For example, a column that is named personid will have a corresponding hex value: "0008706572736f6e696400". It starts with 00 which is null and then 08, which is the length of personid. The string personid alone has this hex value: "706572736f6e6964". The hex value ends with 00 again. This hex string is created for the column and is then converted to a byte buffer (see Code 5.11). The correct column is retrieved based on this byte buffer. The value for the column in the outer row will be returned to the get command method, explained in Chapter 5.3.4.

Code 5.11 Get column value for one row

```
1  if(!isKeyRange || !usesSecondaryIndexing){
2    return row.key.key;
3  }
4  int columnNameLength = column.toString().length();
5  int nullTerminator = 0;
6  //Since the columns are stored as null prefix terminated strings:
7  String columnWithTerminator = "" + (char) nullTerminator + "" + (
       char) columnNameLength + "" + column.toString() + "" + (char)
       nullTerminator + "";
8  ByteBuffer bb = ByteBufferUtil.bytes(columnWithTerminator,
       StandardCharsets.UTF_8);
9  Column joinOnCol = row.cf.getColumn(ByteBufferUtil.bytes(
       columnWithTerminator, StandardCharsets.UTF_8));
10 return joinOnCol.value();
```

5.3.6 Has single column restriction

This method compares the value of the column that a row contains against a join single column (described in Chapter 5.2.9) restriction where the row and the restriction belong to the same column family. A hash map that was created in Chapter 5.2.9, contains all the restrictions (values) on column families (keys) that are also present in an N-Way-Join. Before this method, "hasSingleColumnRestriction", is called, a check against this hash map is executed. However, it is only called if there exists a restriction on the column family. This method is used at two places in the code. The first place is the outer loop where outer rows are transformed into big rows (described in Chapter 5.3.3). The second place is the inner loop where the inner rows are processed (described in Chapter 5.3.7).

When a column value is checked, it first checks if there even exist a single column restriction. If not, true is returned because the column family has not been used in the where clause. Even though this method is not called if there does not exist a single column restriction, a check is used to avoid null pointer exceptions. This method returns true if there exists a column in a row that matches the value of the single column restriction. The column that is used in the where clause can either be on the same column as the join restriction or it can be a different column. However, both cases are executed in the same way. The column value is sent to this method and is used to compare with the value in the single column restriction. This column value is found in the same way as described in Chapter 5.3.5. Both values are translated from bytes to a string with hexadecimals. This translation ensures that not only integer values can be checked, but also strings. If there is a match, true is returned. False is returned if there is no match and this will exclude the row from the result and thereby the big row explained in Chapter 5.3.2.

5.3.7 Inner loop

In the inner loop, the outer row is merged with all the inner rows that matched on the join value. Since the column family of the inner row also can be in the where clause, a check for this is also needed here. The method "hasSingleColRestriction" explained in Chapter 5.3.6 is used for the inner rows (the same method is used for the outer rows). There is also a possibility that the column family is not used in the where clause. These checks are performed by the if and else clause as shown in Code 5.12. If the column family is not found in the hash map (created in Chapter 5.2.9), the variable res will remain null. The inner and outer rows will just be merged without removing some of the rows. If res is assigned, some rows may be removed during execution.

Code 5.12 Check if the row matches the value in the where restriction

```
1  if (singleColumnRestrictionsOnJoinCFs.containsKey(innerColumnName.
       cfName)) {
2    res = singleColumnRestrictionsOnJoinCFs.get(innerColumnName.cfName
       );
3  }
4  if (res != null) {
5    ByteBuffer colValue = getValue(innerRows.get(i), res.getColumn().
       toString(), res.isKeyRange(), res.isUsesSecondaryIndexing());
6    if (hasSingleColRestriction(res, colValue, innerColumnName,
       variables, res.getColumnFamily(), res.getColumn().toString(),
       innerRows.get(i))){
7      BigRow tempBig = getBigRow(outerColumnFamily, outerColumnName,
         innerColumnFamily, innerColumnName, innerRows.get(i),
         specifications, variables, outerRow, outerByteBufferRows,
         limit);
8      surviningBigRow.add(tempBig);
9    }
10 }
11 else {
12   BigRow tempBig = getBigRow(outerColumnFamily, outerColumnName,
       innerColumnFamily, innerColumnName, innerRows.get(i),
       specifications, variables, outerRow, outerByteBufferRows,
       limit);
13   surviningBigRow.add(tempBig);
14 }
```

As can be seen in Code 5.12, method "getBigRow" is mentioned twice. This is because both cases (column family is or is not in the where clause) must do the same operation. What this method does, is that it creates a new big row object containing the old results. The new result is first merged into a new list containing byte buffers and then added to the big row. When the result is printed, column specifications are used to print the column name at the end (Column specifications contains keyspace name, column family name, column name, and type [22]). It is, therefore, necessary to store these column specifications during execution for later use when the result is returned (see Chapter 5.3.8). After the specifications have been added, the big row is added to a list called surviving big row. When the execution is done for one join restriction, the list with the surviving big rows are returned (this happens outside the outer loop). A list that is accessible by all methods in the class will be cleared, and the new big rows are added. All join restrictions in an N-Way-Join except the first relies on the results found in the previous join restriction. The list of big rows can be used by all these join restrictions. After the join restriction is finished, the method described in Chapter 5.3.1 goes to the next join restriction in the N-Way-Join.

5.3.8 Return query results

So far, each section in Chapter 5.3 has been focusing on each join restriction. In the end, all results for a single N-Way-Join must be put together. Under the inner loop execution, the big rows that survived were returned and added to a new list. Since each row is a list with bytes, and these are stored in the big row objects, these lists are added to a new list that will be used to create the result message. An implementation of this solution can be seen in Code 5.13.

Code 5.13 Create result message

```
1  ArrayList<List<ByteBuffer>> endResult = new ArrayList<List<
       ByteBuffer>>();
2  for (BigRow bigRow : lastBigRows){
3    if(bigRow.getResult().size() == specifications.size()) {
4      endResult.add(bigRow.getResult());
5    }
6    else{
7      throw new InvalidRequestException("Error");
8    }
9  }
10 ResultSet returnResultSet = new ResultSet(new ResultSet.Metadata(
       specifications), endResult);
11 return new ResultMessage.Rows(returnResultSet);
```

Result messages are already implemented in Cassandra and are used by standard select statements. This implementation of join also uses the result message object found in Cassandra, so it fits with already existing architecture. A result message will be used when all the N-Way-joins and standard select statements are merged, creating a Cartesian product described in Chapter 5.5.

5.4 Execute standard select statements

A join statement can also contain column families who are not part of an N-Way-Join. The list created in Chapter 5.2.11 are used in this implementation. Each column family is iterated through, and each column family is processed where a result message is returned. This result message is used when the results are merged.

Since the standard select statement from Cassandra only supports one column family, there are some modifications needed for this implementation. In Cassandra, all restrictions belong to one column family with all restrictions stored in single arrays [22]. With multiple column families, a hash map with the column family definition as the key and the array as the value is needed instead. When each column family is executed, a lookup on the key in the hash map can be done to execute it as a standard select statement in Cassandra. As described in Chapter 5.2.9, the restrictions contains boolean variables about the columns that are affected by the where clause. These boolean variables are used to separate between

restrictions on primary keys and secondary indexes. When this execution is done, a result message is returned.

5.5 Merge all results

When all results have been found, a Cartesian operation is performed. As shown in Code 5.14, this method is recursive because each row from one result message must be combined with all other rows from the other result messages. The list resultRows is accessible by the whole class and all new rows are added to this class. Variable k keeps track on which result message that is accessed. For example, one row from result message one is combined with all other rows in result message two.

Code 5.14 Cartesian product of all result messages

```
1  private void cartesianProduct(List<ResultMessage.Rows>
       resultMessages, List<ByteBuffer> newRow, int k ){
2    List<ByteBuffer> prev = new ArrayList<ByteBuffer>();
3    prev.addAll(newRow);
4    if(k==resultMessages.size()){
5      resultRows.add(newRow);
6    }
7    else{
8      for (int j = 0; j < resultMessages.get(k).result.rows.size(); j
           ++){
9        newRow.addAll(resultMessages.get(k).result.rows.get(j));
10       cartesianProduct(resultMessages, newRow, k + 1);
11       newRow = new ArrayList<ByteBuffer>();
12       newRow.addAll(prev);
13     }
14   }
15 }
```

This final result is returned as a result message to the query processor class that was already implemented in Cassandra (a select statement also uses this class). After this, the result is printed to the user and the operation is done.

5.6 Example execution

For a better understanding of the execution of a query that contains a join, an example is given here with the use of the column families in Table 5.1. To exemplify the method in Chapter 5.5, only houseowner, house, and postalcode is joined. There is also a restriction on the postalcode column in column family postalcode. Person is just added to the query in the from clause and the following query is shown in Listing 5.15.

```
SELECTJOIN *
FROM person, houseowner, house, postalcode
WHERE postalcode.postalcode='7030'
JOINON houseowner.houseid=house.houseid AND house.postalcode=
    postalcode.postalcode;
```

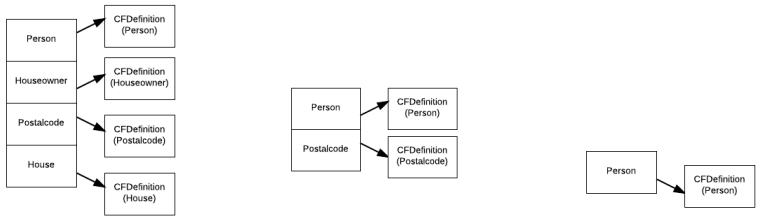
Listing 5.15: Example CQL query

The first thing that happens is that the parser checks if the syntax is correct. Since it is an asterisk in the selectjoin clause, an empty list of raw join selectors is returned. If there had been some variables in the selectjoin clause, the list would not be empty. All column families in the from clause are returned in a list that is used to find the CFDefinitions.

To convert the column families to CFDefinitions, the list of column families is iterated over. During this iteration, selections from the selectjoin clause are also created. Since no columns are defined in the selectjoin clause, each column family will have a simple selection in the hash map where the CFDefinition is the key. If there had been a selection on a column, each column family would have had a corresponding join selection with an empty list of columns except the column family with the column in the selectjoin clause. The column family in the selectjoin clause would also have a join selection, but with a list that contained the column. During this iteration, all column families are also added to a hash map that will consist of the column families that are not part of the join.

In the joinon clause, there are three column families involved where two join relations are returned from the parser. These column families are houseowner, house, and postalcode. Houseowner and house are the first join relation and is, therefore, prepared first. The first thing that happens is that the column family name is checked against the hash map (see Figure 5.9a). This hash map will contain the column families outside the join when all column families in the join are prepared. House and houseowner will be removed from the hash map with the resulting hash map only containing two column families as shown in Figure 5.9b. In the next step, postalcode and house are the two last column families. House has already been removed, but postalcode has to be removed resulting in the hash map in Figure 5.9c. This hash map will be used under the standard execution and not in the join. The second step explained here will not happend before it is done with the join relation containing house and houseowner.

The next step is to check if the left column family in the join relation is bigger (in means of megabytes) than the left column family. Houseowner is the left column family in the join relation and is biggest. With this result, houseowner is assigned to the left column family in the join restriction and house will be the right column family (left column family in the join restriction is always the biggest). The column houseid in column family houseowner is a secondary index, and both "isKeyRange" and "usesSecondaryIndex" is true. This information is stored in the join restriction as the left boolean variables. In the right column family, "isKeyRange" and "usesSecondaryIndex" is false because the column houseid in column family house



(a) Initial hash map (b) Removal of house and houseowner (c) Removal of postalcode

Figure 5.9: Finding column families outside any N-Way-Join in the query

is a primary key. This information is also stored in join restriction, but as the right boolean variables. This join restriction (Res-1 in Figure 5.10) is then added to a hash map, where the column family name is the key, and a list of join restrictions is the object. The final hash map when all join restrictions have been created is shown in Figure 5.10. This hash map is used when the ordering of the join is decided. This process is repeated with the second join relation, with left column family house and right column family postalcode where the outcome is a join restriction (Res-2 in Figure 5.10).

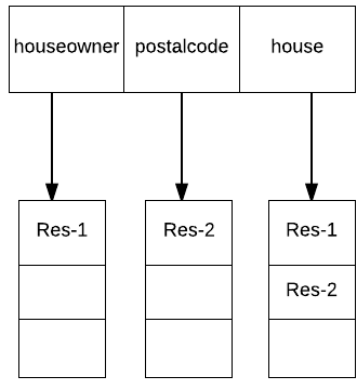


Figure 5.10: Example hash map containing all column families with corresponding restrictions

As stated in the query, all final rows must have 7030 as their value in the column postalcode. Since this restriction is on a column in a column family that is used in the join, a join single column restriction is created. If a column family outside the join was in the where clause, a modified single column restriction would

have been created. Boolean variables about the column is also stored in the join single column restriction (if it is a primary key or secondary index). Since the column postalcode in the column family postalcode is a primary key, both boolean variables are false. Column and column family name is stored in the join single column restriction too. In addition to this, the column family postalcode is added to an array list that contains all the column families present in both the where clause and an N-Way-Join. This array list is used when the join ordering is found.

It is desirable remove data early if it possible. If a column family is used in an N-Way-Join and in the where clause, some data may be removed early. The list created when the relations in the where clause was iterated is used here. A temporary list TL is created that will contain all column families in all N-Way-Joins. Then the array list that contains column families present in both an N-Way-Join and the where clause is iterated. A match is found on the column family postalcode and this is therefore added to the list TL. The next step is to add all the column families in the hash map in Figure 5.10. When the keys are iterated through, houseowner and house will be added to the list TL. Since postalcode is already added, it will skip this column family.

The next step is to create the N-Way-Joins where each join restriction belongs to only one N-Way-Join. To accomplish this, an iteration over the list TL is done. Column family postalcode is read first from TL. Each join restriction in the hash map in Figure 5.10 with column family postalcode (the other column family in the join restriction) is iterated through in a recursive method. Since this join restriction (Res-2 in Figure 5.10) is the only one, it is added to the N-Way-Join. The second column family in the join restriction is found, which is house. The method is recursively called, but there are two possible scenarios. If the size of the list for the column family house is one, the join restriction is removed from this hash map. However, the list for house is two so Res-2 is removed from this list and the method is recursively called. Then all the join restrictions that uses the column family house is iterated through. The first join restriction (Res-1 in Figure 5.10) is then added to the N-Way-Join and the method finds out that the second column family is houseowner. It then checks the hash map in Figure 5.10 and discovers that the size is one. Res-1 is just removed from the hash map. The method then returns to the iteration of TL. It will also iterate over house and houseowner, but they will be ignored since they are already added to an N-Way-Join.

After creating the N-Way-Joins, all join restrictions in each N-Way-Join is sorted based on matching column families with the previous join restriction. This creates a join tree seen in Figure 5.11. For the one N-Way-Join created above, the first join restriction added to the ordered list is Res-2 (house and postalcode column family). This join restriction is matched against the other join restrictions. However, it is only one other join restriction and this is Res-1 (house and houseowner). This is then added to the ordered list since there is a match on the column family house. No more checks are done since only two join restrictions are used in the N-Way-Join.

As Figure 5.11 shows, Res-2 will be executed first, and then Res-1 will be executed where the result from column family postalcode will be used. In the end,

a list with all N-Way-Joins (in this case, only one) is completed and stored in the join statement for use under the execution step. After the prepare step is done, a prepared statement is returned to the query processor that will execute the join statement.

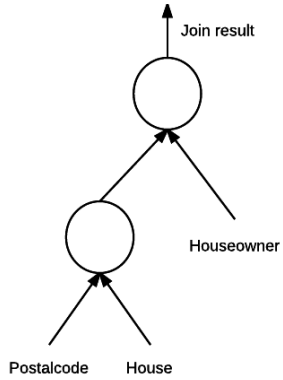


Figure 5.11: Example of join tree for an N-Way-Join

During the execution, three different operations will be executed. First, each N-Way-Join is executed. Second, all column families outside the join is executed and third, all results are merged where the result is a Cartesian product. First, each N-Way-Join is executed and in this case, there only exists one N-Way-Join. When this N-Way-Join is executed, each join restriction is looped through and the first join restriction is executed. Since it is the first join restriction, both column families (postalcode and house) has to be read from the data store. Column family postalcode is in the outer loop and is read from the data store first. This is only a simple read operation where all rows (see Table 5.3) in postalcode are fetched.

postalcode	**cityname**
7030	Trondheim

Table 5.3: All rows in column family postalcode with postalcode value 7030

There is only one row that is converted to big row because only one row has a match on the value 7030 for the column postalcode. To find the inner rows that match with the column postalcode, each outer big row (in this case, one) has a hash map where the key is the column family and a single row from that column family as the object. For the big row described above, a get operation on the hash map with the key postalcode is executed. The correct command is then generated where the join value (7030) is found. This command is then used to fetch all the inner rows from the column family house. When all inner rows have been found for one outer row, an iteration over each inner row is done. A new big row is generated where the outer row from postalcode and one row from house are merged. During iteration, each big row that is created in the inner loop is added to a surviving big

row list that will be used by the next join restriction. When this join restriction is done, the result contains two rows as shown in Table 5.4.

postalcode	**cityname**	**houseid**	**color**	**postalcode**	**size**
7030	Trondheim	2	Blue	7030	Big
7030	Trondheim	3	Green	7030	Medium

Table 5.4: All big rows for $postalcode \bowtie_{postalcode.postalcode=house.postalcode} house$

For the next join restriction (between house and houseowner), the results from the previous join restriction (between postalcode and house) are used as the outer big rows. The results from the join between postalcode and house can be seen in Table 5.4. The column family house does not need to be read from the data store again, and this is, therefore, the outer column family in the nested loop. When each outer big row is iterated through (two iterations in this case), a get operation on the hash map in the big row is executed, and the correct outer row is fetched. With this row, a join value for the column houseid (2 and 3 as seen in Table 5.4) in column family house is found. This value is then used to find the correct command and fetch the matching rows from the column family houseowner. The matching rows from houseowner is merged with the matching big row, and the final result can be seen in Table 5.5. A result message is returned and stored in a list for later use under the merge operation.

postal-code	**city-name**	**house-id**	**color**	**postal-code**	**size**	**id**	**house-id**	**person-id**
7030	Trond-heim	2	Blue	7030	Big	3	2	1
7030	Trond-heim	3	Green	7030	Med-ium	4	3	4

Table 5.5: All big rows for $postalcode \bowtie_{postalcode.postalcode=house.postalcode} house$ and $house \bowtie_{house.houseid=houseowner.houseid} houseowner$

The next operation is to read all column families that are outside any join. Since it does not exist any restrictions on the column family person, all rows are read. The result from this column family can be seen in Table 5.6. A result message is created for this column family, and it is also stored in the same list as the result message from the N-Way-Join. The last step is to merge all results from all result messages. It is only two result messages from this query since each N-Way-Join creates one result message and each standard select also generates one result message. As shown in Table 5.6, there are four rows from column family person. The N-Way-Join has two rows, as shown in Table 5.5. When the result is merged, a Cartesian product from the result messages is created. Each row in the N-Way-Join is combined with each row in the column family person that creates eight rows as the final result. This result is returned to the query processor, and the final result is printed to the user.

personid	**adress**	**familyname**	**givenname**
1	P.A. Munchs Gate 6	Peter	Christian
2	Riisalleen 26	Nordmann	Ola
4	Osloveien 66	Jensen	Jens
3	Riisalleen 26	Nordmann	Kari

Table 5.6: All rows from column family person

Chapter 6

Evaluation

This chapter looks at how the join implementation presented in Chapter 5 performs compared to the well-known MySQL. In Chapter 6.1, the method for evaluating the performance of this join implementation is presented. Chapter 6.2 presents the test data and Chapter 6.3 describes the equipment used during the tests. The result is presented in Chapter 6.4, where different queries are tested. The execution times for insertion of rows are also given here because this gives an interesting look at the differences between MySQL and Cassandra. All results are discussed in Chapter 6.5.

6.1 Method

To test the join implementation, five warm up runs are done first before twenty test runs are done. The warm up runs are used to warm up the servers in case of caching on the server side. It is the average of the twenty test runs that is used as results, but all runs can be found on GitHub [10].

Different queries were also tested to check performance, including insertion and selection to both Cassandra and MySQL. This gives a broader picture on the strengths and weaknesses of Cassandra, compared to an SQL database system.

6.2 Test data

For this test, a data generator created for this master thesis was used. This source code can be found on GitHub [18]. This source code was also used to perform the test.

For this test, five column families/tables are used to create a 5-Way-Join. These tables can be seen in Figure 6.1. Each customer can have zero or more trades. One trade can include multiple products, but one product can belong to multiple trades.

This requires an additional column family/table to store this relation called producttrade. This table contains an id and two columns that are either secondary indexes (Cassandra) or foreign keys (MySQL). In MySQL, the table trade also contains a foreign key for customer (customernr), and the table product has one foreign key for the manufacturer (manufacturernr). In Cassandra, secondary indexes are used instead of foreign keys. All datasets can be found on GitHub [18].

Each column family/table is created six times, one for each number of rows. For example, will the column family/table customer exist as customer_10, customer_100 and up to customer_1000000. This makes it easier to test since an insertion to the database is not needed for each test.

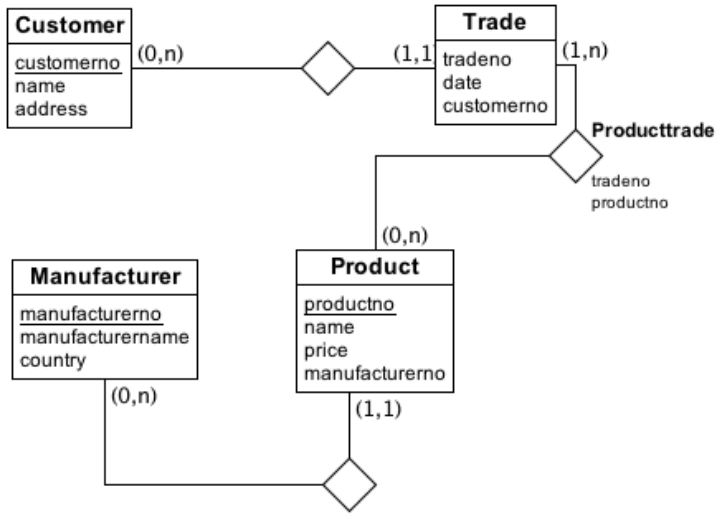


Figure 6.1: All five tables/column families used during testing

6.3 Equipment used under testing

This test was executed on Cassandra version 2.0 and MySQL version 5.5.41 with a single node. This node is running on a local Ubuntu machine with eight Intel i7-4770 CPU cores with 3.40GHz and 16 gigabyte of ram. Also, the heap size for Cassandra has been set to 8GB and 800MB per physical CPU core.

The read timeout option in Cassandra has also been changed (originally 5000 milliseconds), on the client and server side. This was done because this enables large column families to be joined without having timeouts. Since this is a testing environment, the timeout value was set to the max value of the integer object in

Java, but in a production setting this should be set to a reasonable value or not changed at all.

6.4 Results

All results generated from the performance testing are presented in this chapter. The results from a join without any where relations or projections are presented, but also with projections and where relations. Performance testing of insertion and a select statement with one column family/table are also included in this chapter. Each graph presented uses a logarithmic scale (log_{10}) on the y-axis to show better the difference between Cassandra and MySQL.

6.4.1 Join without where relations and projections

For this test, two queries are used (one for MySQL and one for Cassandra). The query used for Cassandra can be written as shown in Listing 6.1.

```
SELECTJOIN *
FROM customer, product, trade, producttrade, manufacturer
JOINON customer.customernr = trade.customernr AND trade.
    tradenr = producttrade.tradenr AND producttrade.productnr
    = product.productnr AND product.manufacturenr =
    manufacturer.manufacturenr;
```

Listing 6.1: Join query in CQL without where relations and projections used during evaluation

MySQL uses a different query and it can be seen in Listing 6.2.

```
SELECT *
FROM customer, product, trade, producttrade, manufacturer
WHERE customer.customernr = trade.customernr AND trade.
    tradenr = producttrade.tradenr AND producttrade.productnr
    = product.productnr AND product.manufacturenr =
    manufacturer.manufacturenr;
```

Listing 6.2: Join query in SQL without where relations and projections used during evaluation

Figure 6.2 shows how a join query without where relations and projections perform in Cassandra and MySQL. It can be seen from this graph that MySQL performs better from ten rows to one million rows. However, when the number of rows is only ten, the difference is minimal. Cassandra uses 7 milliseconds to perform the join, but MySQL uses 0 milliseconds (close up to 1 millisecond). When the number

of rows is increased to one hundred rows, MySQL performs even better than Cassandra. Cassandra uses 50 milliseconds, and MySQL uses close up to 1 millisecond. When the number of rows is increased to one thousand rows, Cassandra uses 301 milliseconds, and MySQL uses 1 millisecond. At ten thousand rows, MySQL only uses 50 milliseconds. The execution time for ten thousand rows in MySQL is 75 times faster than Cassandra, which uses 3772 milliseconds. When one hundred thousand rows are used, MySQL uses 587 milliseconds and performs only 69 times faster than Cassandra (40689 milliseconds). For one million rows, Cassandra uses 669985 milliseconds. MySQL uses 33014 milliseconds. As seen in the graph, the biggest difference between Cassandra and MySQL is at one thousand rows.

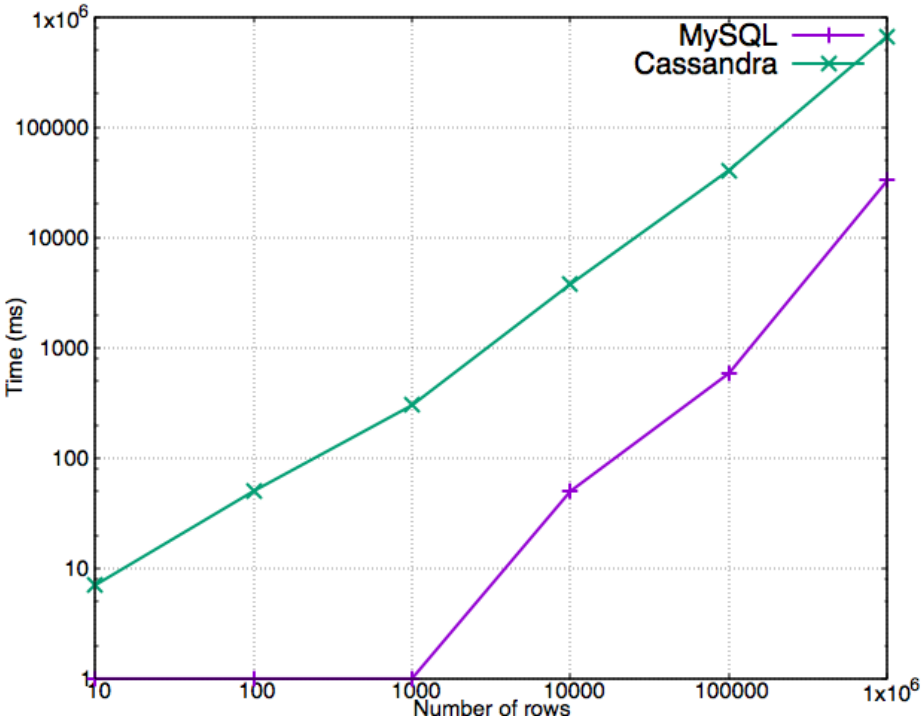


Figure 6.2: This graph shows the execution time of a join without any where relations or projections in Cassandra and MySQL for different number of rows

6.4.2 Join with one where relation

In both Cassandra and MySQL, there is possible to decrease the number of relevant rows to a query by using one or more where relations. For this test, two queries are used creating two performance tests. One query will make use of a column that is a foreign key/secondary index, and the other query makes use of a column that is a primary key. Both types use the same query for Cassandra and MySQL shown in Chapter 6.4.1. The only difference is that MySQL adds another relation (including

Number of rows	Customer number
10	7
100	59
1000	33
10000	5335
100000	47000
1000000	862912

Table 6.1: Different customer numbers used for each number of rows.

the join relations) in the where clause. Cassandra adds the relation in the where clause and the join relations are still placed in the joinon clause. When the data set was generated, different customer numbers were used (one match at least in the column family/table trade). The different numbers can be seen in Table 6.1. The queries that are used for Cassandra and MySQL can be seen below (for both primary key and secondary index/foreign key). Since each query is unique for each number of rows, the question mark in the queries is one of the values represented in Table 6.1. Both queries also have a column family/table T represented in the where relation. This is either the column family/table customer or trade. The query used for Cassandra can be seen in Listing 6.3. MySQL uses a different query and it can be seen in Listing 6.4.

```
SELECTJOIN *
FROM customer, product, trade, producttrade, manufacturer
WHERE T.customernr=?
JOINON customer.customernr = trade.customernr AND trade.
    tradenr = producttrade.tradenr AND producttrade.productnr
    = product.productnr AND product.manufacturenr =
    manufacturer.manufacturenr;
```

Listing 6.3: Join query in CQL with one where relation used during evaluation.T is either customer or trade. The question mark is one the values in Table 6.1

```
SELECT *
FROM customer, product, trade, producttrade, manufacturer
WHERE T.customernr=? AND customer.customernr = trade.
    customernr AND trade.tradenr = producttrade.tradenr AND
    producttrade.productnr = product.productnr AND product.
    manufacturenr = manufacturer.manufacturenr;
```

Listing 6.4: Join query in SQL with one where relation used during evaluation. T is either customer or trade. The question mark is one the values in Table 6.1

With primary key

This query uses the column customernr in column family/table customer. Figure 6.3 shows how Cassandra and MySQL perform when a query with a where relation on the primary key is executed. MySQL uses 0 or 1 millisecond on all the number of rows. When Cassandra executes the similar query on ten rows, it uses 1 millisecond. When the number of rows is increased to one hundred, the execution time does not change from ten rows. At one thousand rows, it takes 3 milliseconds to complete the query. With ten thousand rows, the execution time is 24 milliseconds and at one hundred thousand rows it is 213 milliseconds. One million rows in each column family make Cassandra use 2920 milliseconds. As seen in the graph, the difference gets larger and larger when the number of rows is increased for Cassandra.

With foreign key/secondary index

This query uses the column customernr in column family/table trade. As seen from the graph in Figure 6.4, MySQL has the same execution time (0 or 1 millisecond) as the query using a where relation on a primary key. For Cassandra, the execution time is almost the same. However, at one hundred rows Cassandra uses 10 milliseconds. This is because one of the test runs used 169 milliseconds and all the other runs used 1 or 2 millisecond(s). At ten thousand rows, Cassandra uses 35 milliseconds and the execution time for one hundred thousand rows is 246 milliseconds. For one million rows, the execution time is 2982 milliseconds that are almost the same as the query with the where relation on a primary key.

6.4.3 Join with projections

The graph in Figure 6.5 shows the execution time for Cassandra and MySQL for a query containing projections. The queries for Cassandra and MySQL can be seen in Listing 6.5 and 6.6, respectively.

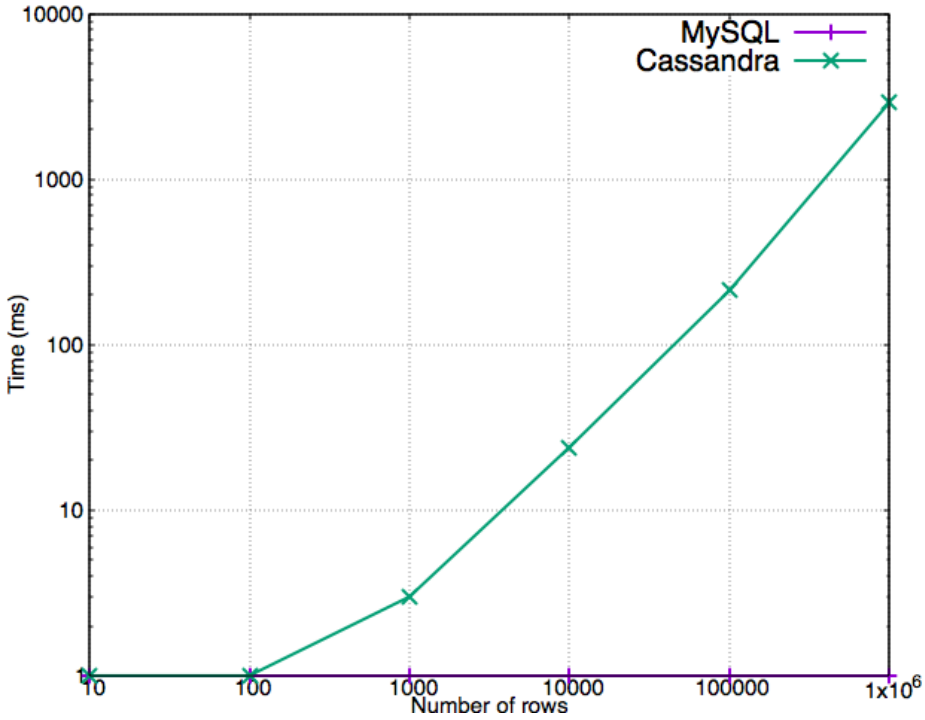


Figure 6.3: This graph shows the execution time of a join with a where relation on the primary key customernr in Cassandra and MySQL for different number of rows

```
SELECTJOIN customer.name, customer.address, manufacturer.
    manufacturer.name, manufacturer.country
FROM customer, product, trade, producttrade, manufacturer
JOINON customer.customernr = trade.customernr AND trade.
    tradenr = producttrade.tradenr AND producttrade.productnr
    = product.productnr AND product.manufacturenr =
    manufacturer.manufacturenr;
```

Listing 6.5: Join query in CQL with projections used during evaluation

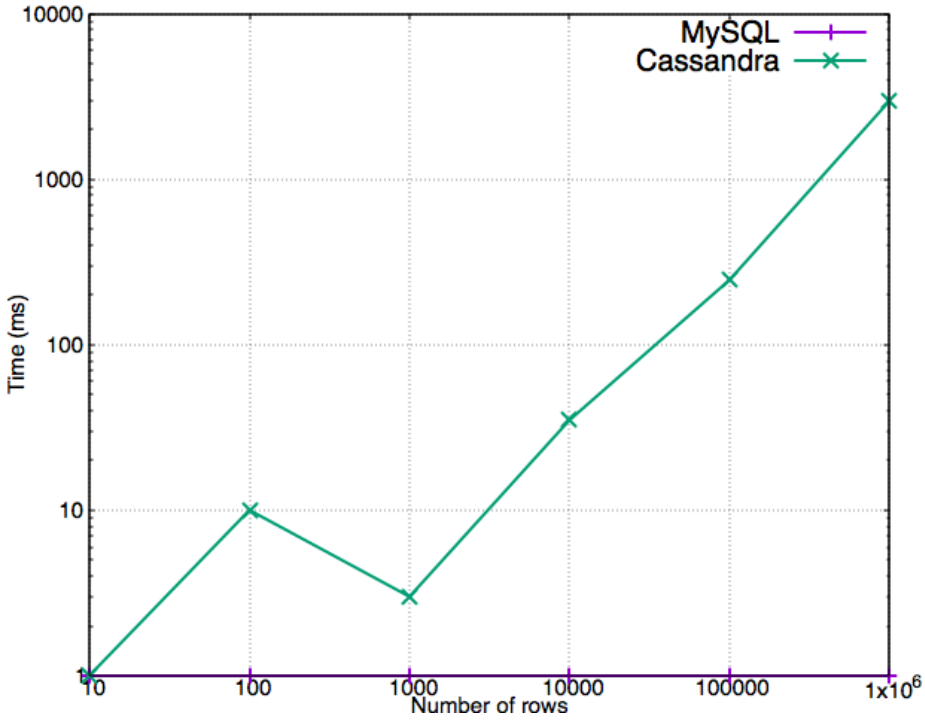


Figure 6.4: This graph shows the execution time of a join with a where relation on the secondary index/foreign key customernr in Cassandra and MySQL for different number of rows

```
SELECT customer.name, customer.address, manufacturer.
    manufacturer.name, manufacturer.country
FROM customer, product, trade, producttrade, manufacturer
WHERE customer.customernr = trade.customernr AND trade.
    tradenr = producttrade.tradenr AND producttrade.productnr
    = product.productnr AND product.manufacturenr =
    manufacturer.manufacturenr;
```

Listing 6.6: Join query in SQL with projections used during evaluation

It can be seen that Cassandra uses 4 milliseconds for ten rows, and MySQL uses 0 milliseconds. At one hundred rows, Cassandra uses 31 milliseconds whereas MySQL still uses 0 milliseconds. It is the same execution time for MySQL at one thousand rows, but Cassandra uses 250 milliseconds. When the number of rows is increased to ten thousand, MySQL has an execution time of 2 milliseconds while Cassandra has 3660 milliseconds. For one hundred thousand rows, Cassandra uses 39746 milliseconds while MySQL executes on 438 milliseconds. One million rows

takes Cassandra 673211 milliseconds and MySQL 25394 milliseconds.

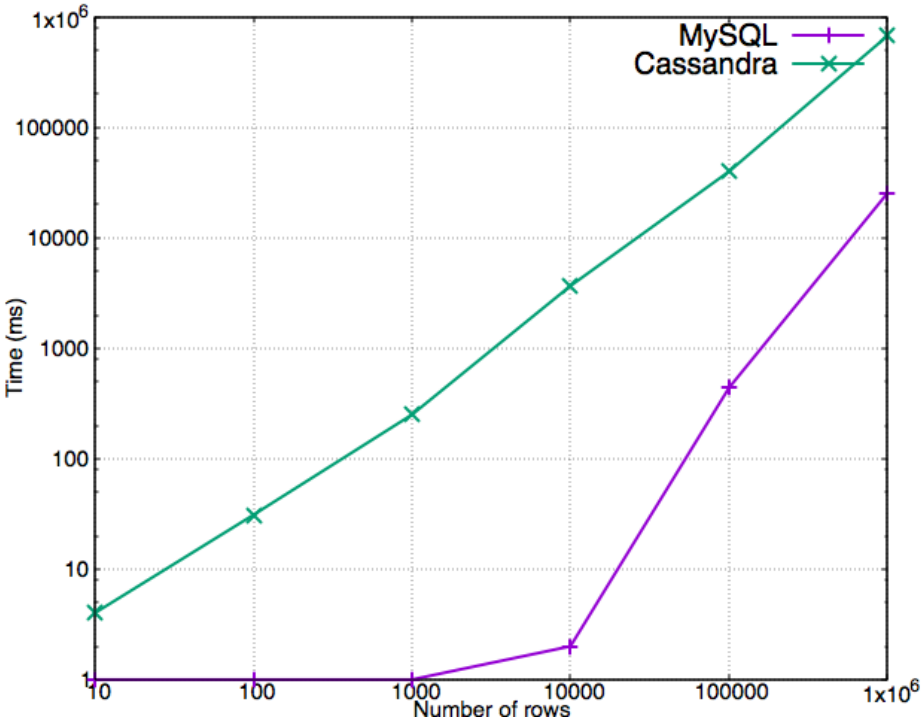


Figure 6.5: This graph shows the execution time of a join with projection in Cassandra and MySQL for different number of rows

6.4.4 Single SELECT statement

For this test, a simple select query was used. It is identical for CQL and SQL and can be seen in Listing 6.7. The variable k in the query represents the number of rows, for example ten or one million (it must be specified in Cassandra since it is default ten thousand if no other limit has been set [6]).

```
SELECT *
FROM trade
LIMIT k;
```

Listing 6.7: Select query for CQL and SQL used during evaluation

As seen in the graph from Figure 6.6, Cassandra returns ten rows in 0 milliseconds, and MySQL does the same. When the number of rows is increased to one hundred, Cassandra uses 1 millisecond and MySQL still uses 0 milliseconds. At one thousand rows, the execution time is still 0 milliseconds for MySQL, but Cassandra executes

in 4 milliseconds. For ten thousand rows, MySQL uses 1 millisecond, and Cassandra uses 18 milliseconds. However, when the number of rows is one hundred thousand, MySQL (23 milliseconds) uses longer time than Cassandra (18 milliseconds). The difference is increased when the number of rows is one million. Cassandra uses 17 milliseconds, and MySQL uses 375 milliseconds.

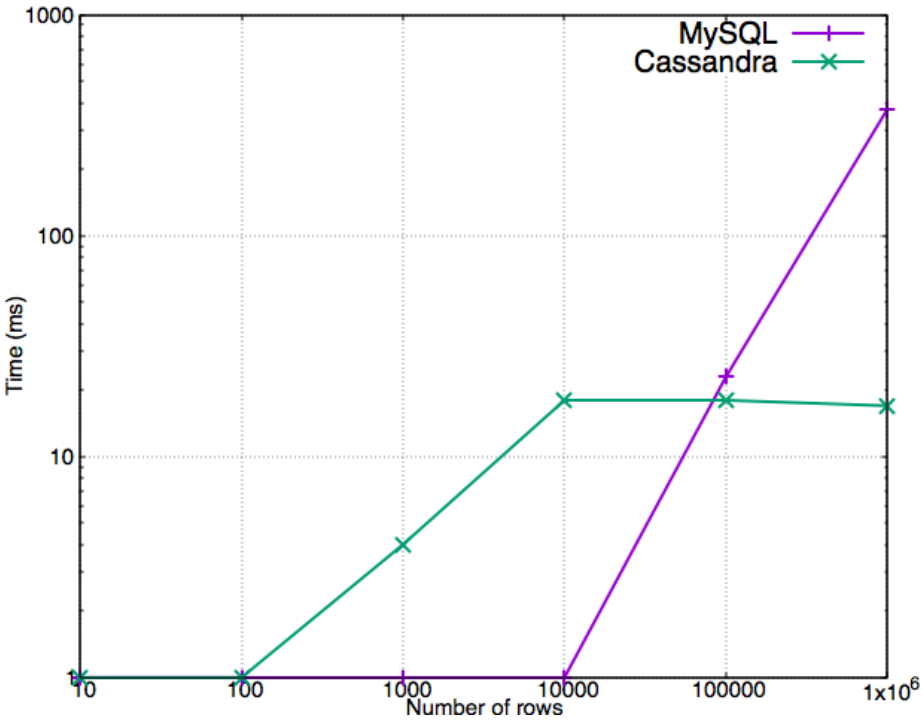


Figure 6.6: This graph shows the execution time of a select statement with only one column family/table in Cassandra and MySQL for different number of rows

6.4.5 Inserting rows

For the insertion of data, five different insert statements are used. Figure 6.7 shows how Cassandra performs compared to MySQL when different number of rows are inserted. In both Cassandra and MySQL, the insert statements are executed one by one. In MySQL, it is also possible to execute insert statements as a batch or use a file as the input to insert rows.

As seen from the graph, Cassandra (14 milliseconds) performs slightly better than MySQL (28 milliseconds) when inserting ten rows. When the number of rows is increased to one hundred rows, Cassandra uses 51 milliseconds, and MySQL uses 274 milliseconds. The difference increases when the number of rows is increased to one thousand and up to one million rows. This difference can also be seen in the graph in Figure 6.7.

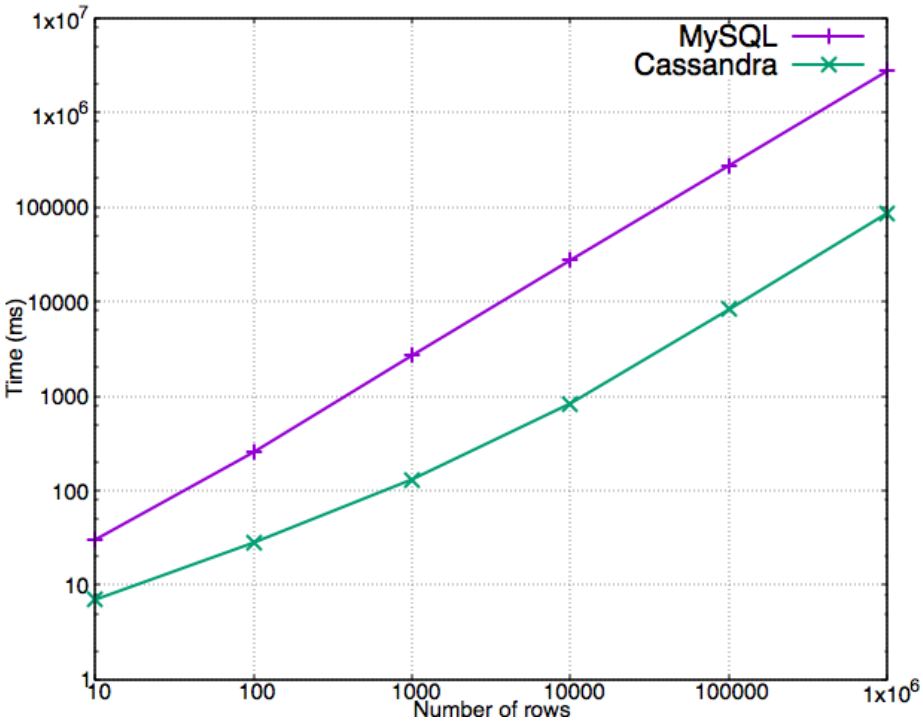


Figure 6.7: This graph shows the execution time of insertion in Cassandra and MySQL for different number of rows

6.5 Discussion

There may be different reasons for why Cassandra is slower than MySQL when a join is executed. There may be reading from the disk that is the bottleneck, or it may be the garbage collector in Java that makes Cassandra slower than MySQL for join. However, it is most likely the implementation of join in Cassandra that makes it slow compared to MySQL. In this chapter, all of these possibilities are discussed with the purpose of trying to discover the cause of the poor performance of Cassandra compared to MySQL.

6.5.1 JVM options

To better understand what happen with the garbage collector for Java during execution, both the jstat command was used and the VisualVM plugin Visual GC. The command used for monitoring the garbage collector was the Java Virtual Machine Statistics Monitoring Tool (jstat) [12]. The command is

```
jstat -gc <pid> 250ms 0
```

where pid is the process identification and a sample is printed each 250 milliseconds. The last number represents number of samples to print out (0 indicates infinite or until the user aborts the program). Figure 6.8 shows a part of the result from the jstat command when a query is sent to Cassandra.

S0C	S1C	S0U	S1U	EC	EU	OC	OU	MC	MU	CCSC	CCSU	YGC	YGCT	FGC	FGCT	GCT
81920.0	81920.0	0.0	81920.0	655360.0	463387.1	7569408.0	3600459.4	27232.0	26598.9	3172.0	2978.8	99	5.760	2	0.062	5.822
81920.0	81920.0	0.0	81920.0	655360.0	463726.5	7569408.0	3600459.4	27232.0	26598.9	3172.0	2978.8	99	5.760	2	0.062	5.822
81920.0	81920.0	0.0	81920.0	655360.0	464068.3	7569408.0	3600459.4	27232.0	26598.9	3172.0	2978.8	99	5.760	2	0.062	5.822
81920.0	81920.0	0.0	81920.0	655360.0	464407.8	7569408.0	3600459.4	27232.0	26598.9	3172.0	2978.8	99	5.760	2	0.062	5.822
81920.0	81920.0	81920.0	0.0	655360.0	118824.4	7569408.0	3746850.2	27232.0	26643.7	3172.0	2978.8	100	5.908	2	0.062	5.970
81920.0	81920.0	0.0	81645.6	655360.0	0.0	7569408.0	3828279.9	27232.0	26645.9	3172.0	2978.8	101	5.997	2	0.062	6.058
81920.0	81920.0	0.0	81645.6	655360.0	562028.8	7569408.0	3828279.9	27232.0	26645.9	3172.0	2978.8	101	5.997	2	0.062	6.058
81920.0	81920.0	35388.3	0.0	655360.0	452604.5	7569408.0	3869031.5	27232.0	26645.9	3172.0	2978.8	102	6.042	2	0.062	6.104
81920.0	81920.0	0.0	10754.8	655360.0	351820.4	7569408.0	3876658.7	27232.0	26645.9	3172.0	2978.8	103	6.059	2	0.062	6.121
81920.0	81920.0	9073.6	0.0	655360.0	255784.1	7569408.0	3883813.4	27232.0	26645.9	3172.0	2978.8	104	6.077	2	0.062	6.139
81920.0	81920.0	0.0	7287.7	655360.0	186567.6	7569408.0	3890917.3	27232.0	26645.9	3172.0	2978.8	105	6.094	2	0.062	6.156
81920.0	81920.0	9470.0	0.0	655360.0	125857.5	7569408.0	3897749.5	27232.0	26646.2	3172.0	2978.8	106	6.111	2	0.062	6.173
81920.0	81920.0	0.0	8629.6	655360.0	11636.9	7569408.0	3904740.4	27232.0	26646.2	3172.0	2978.8	107	6.128	2	0.062	6.190
81920.0	81920.0	0.0	8629.6	655360.0	536403.3	7569408.0	3904740.4	27232.0	26646.2	3172.0	2978.8	107	6.128	2	0.062	6.190
81920.0	81920.0	11774.8	0.0	655360.0	423929.2	7569408.0	3911491.3	27232.0	26646.2	3172.0	2978.8	108	6.150	2	0.062	6.212
81920.0	81920.0	0.0	8393.6	655360.0	322836.8	7569408.0	3918280.0	27232.0	26646.2	3172.0	2978.8	109	6.167	2	0.062	6.229
81920.0	81920.0	12319.2	0.0	655360.0	220265.0	7569408.0	3925178.5	27232.0	26647.2	3172.0	2978.8	110	6.185	2	0.062	6.247
81920.0	81920.0	0.0	7998.4	655360.0	142558.4	7569408.0	3931819.7	27232.0	26647.2	3172.0	2978.8	111	6.211	2	0.062	6.273
81920.0	81920.0	11893.6	0.0	655360.0	77128.9	7569408.0	3938498.8	27232.0	26647.2	3172.0	2978.8	112	6.232	2	0.062	6.294
81920.0	81920.0	11893.6	0.0	655360.0	634788.7	7569408.0	3938498.8	27232.0	26647.2	3172.0	2978.8	112	6.232	2	0.062	6.294
81920.0	81920.0	0.0	7821.7	655360.0	530628.9	7569408.0	3945045.1	27232.0	26647.2	3172.0	2978.8	113	6.250	2	0.062	6.312
81920.0	81920.0	11177.7	0.0	655360.0	452225.0	7569408.0	3952340.8	27232.0	26648.5	3172.0	2978.8	114	6.266	2	0.062	6.328
81920.0	81920.0	0.0	8559.2	655360.0	387293.5	7569408.0	3958906.6	27232.0	26648.5	3172.0	2978.8	115	6.286	2	0.062	6.348
81920.0	81920.0	11719.8	0.0	655360.0	284372.0	7569408.0	3965460.7	27232.0	26648.5	3172.0	2978.8	116	6.306	2	0.062	6.368
81920.0	81920.0	0.0	8251.4	655360.0	171672.3	7569408.0	3971992.0	27232.0	26648.5	3172.0	2978.8	117	6.328	2	0.062	6.389
81920.0	81920.0	0.0	8251.4	655360.0	346195.7	7569408.0	3971992.0	27232.0	26648.5	3172.0	2978.8	117	6.328	2	0.062	6.389
81920.0	81920.0	0.0	8251.4	655360.0	531013.2	7569408.0	3971992.0	27232.0	26648.5	3172.0	2978.8	117	6.328	2	0.062	6.389
81920.0	81920.0	66648.5	0.0	655360.0	48089.7	7569408.0	3978684.2	27232.0	26648.8	3172.0	2978.8	118	6.377	2	0.062	6.439
81920.0	81920.0	66648.5	0.0	655360.0	237666.2	7569408.0	3978684.2	27232.0	26648.8	3172.0	2978.8	118	6.377	2	0.062	6.439
81920.0	81920.0	66648.5	0.0	655360.0	398499.3	7569408.0	3978684.2	27232.0	26648.8	3172.0	2978.8	118	6.377	2	0.062	6.439

Figure 6.8: Monitoring the garbage collector in JVM during read of one hundred thousand rows

The heap used during execution has multiple parts, also known as generations [11]. All parts can be seen in Figure 6.9. The young generation will contain all the new objects with two survivor spaces and one eden space (this fills up first). The old generation stores all objects that have survived a long time (a threshold may have been set for the young generation for movement to the old generation). The last generation is called permanent. This generation contains metadata that the JVM requires describing the classes and methods [11].

As you can see in Figure 6.8, each time the eden space (EU) fills up, this space is garbage collected, and the surviving objects are moved to either S00 or S01. If S00 or S01 contains objects, the surviving objects from one of them are also moved (only one survivor space contains data at once). It is possible to see that the OU (old generation utilization) fills up during execution (never close to the capacity of the old generation). However, this does not affect Cassandra much compared to the execution times from the graphs in Chapter 6.4. From Visual GC, it can also be seen that the total GC time is 1.9 seconds (for one hundred thousand rows).

With this information, it is not likely that the garbage collector in Java makes the execution time high. It must, therefore, be something else that causes Cassandra to perform poorly when a join query is executed.

6.5.2 I/O usage

Another aspect that was investigated was the I/O usage for Cassandra. However this is not likely to be the problem because most of the data will be in memtables

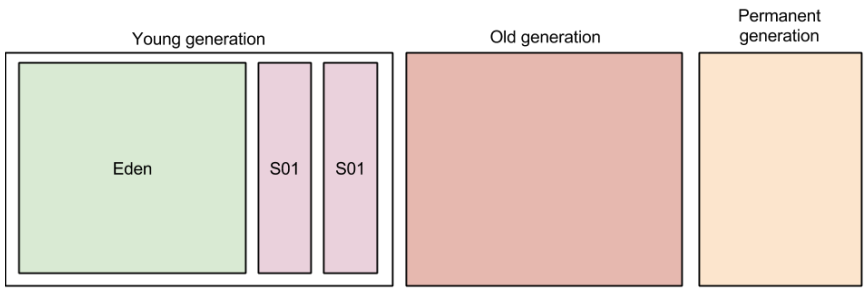


Figure 6.9: Garbage collector design in JVM

(explained in Chapter 2.7.1). Retrieval of data in memtables (if SSTables are avoided) does not require disk reads, hence no I/O against the disk. If reads from SSTables had to be done, the search time for all rows might have been higher since the SSTables are stored on disk while the memtables are stored in the memory. However, the SSTables may also be located in the memory if they are mapped to the memory, and disk I/O is avoided [26].

However, the I/O usage should be checked since it may be a bottleneck in some circumstances (for example heavy reads from SSTables). A Linux command called iostat is used to retrieve some statistics about the I/O for the devices and partitions on the machine. The iostat command was used to monitor the I/O during execution (it monitor all I/O operations on the machine and not only Cassandra). The command used on the Linux machine can be seen here:

```
iostat -dmx 1
```

With this command, some results can be seen in Figure 6.10. As you can see in this figure, there is no reads per second (r/s), and this shows that Cassandra uses the memtables that does not need to read from the disk. Hence, this does not explain the execution time for Cassandra. However, there are some writes per second (w/s). As mentioned earlier, this command monitors all I/O on the whole machine. This means that these writes most likely does not belong to Cassandra (or they may be writes to commit log which is cheap). This implies that there must be something else that causes the execution time to be high.

6.5.3 Profiling of method calls

To get an overview of the CPU usage for each method in Cassandra (those used during a join), profiling was done with the program VisualVM on the CPU. Some of the results from this profiling can be seen in Figure 6.11. Self time is the time used only in the method. Total time is all the time from all methods called from the first method. For example, the nested loop join method uses other methods and the total time will be all these methods execution time including the self time.

All methods belonging to the class JoinStatement are either implemented or changed in this master thesis. As explained in Chapter 5.3, many read commands

```
Device:         rrqm/s   wrqm/s     r/s     w/s    rMB/s    wMB/s avgrq-sz avgqu-sz   await r_await w_await  svctm  %util
sda               0,00     0,00    0,00    5,00     0,00     0,04    16,00     0,00    0,00    0,00    0,00   0,00   0,00

Device:         rrqm/s   wrqm/s     r/s     w/s    rMB/s    wMB/s avgrq-sz avgqu-sz   await r_await w_await  svctm  %util
sda               0,00     0,00    0,00    0,00     0,00     0,00     0,00     0,00    0,00    0,00    0,00   0,00   0,00

Device:         rrqm/s   wrqm/s     r/s     w/s    rMB/s    wMB/s avgrq-sz avgqu-sz   await r_await w_await  svctm  %util
sda               0,00     0,00    0,00    0,00     0,00     0,00     0,00     0,00    0,00    0,00    0,00   0,00   0,00

Device:         rrqm/s   wrqm/s     r/s     w/s    rMB/s    wMB/s avgrq-sz avgqu-sz   await r_await w_await  svctm  %util
sda               0,00     0,00    0,00    0,00     0,00     0,00     0,00     0,00    0,00    0,00    0,00   0,00   0,00

Device:         rrqm/s   wrqm/s     r/s     w/s    rMB/s    wMB/s avgrq-sz avgqu-sz   await r_await w_await  svctm  %util
sda               0,00     0,00    0,00    0,00     0,00     0,00     0,00     0,00    0,00    0,00    0,00   0,00   0,00

Device:         rrqm/s   wrqm/s     r/s     w/s    rMB/s    wMB/s avgrq-sz avgqu-sz   await r_await w_await  svctm  %util
sda               0,00     2,00    0,00    2,00     0,00     0,02    16,00     0,00    0,00    0,00    0,00   0,00   0,00

Device:         rrqm/s   wrqm/s     r/s     w/s    rMB/s    wMB/s avgrq-sz avgqu-sz   await r_await w_await  svctm  %util
sda               0,00     0,00    0,00    0,00     0,00     0,00     0,00     0,00    0,00    0,00    0,00   0,00   0,00

Device:         rrqm/s   wrqm/s     r/s     w/s    rMB/s    wMB/s avgrq-sz avgqu-sz   await r_await w_await  svctm  %util
sda               0,00     0,00    0,00    0,00     0,00     0,00     0,00     0,00    0,00    0,00    0,00   0,00   0,00

Device:         rrqm/s   wrqm/s     r/s     w/s    rMB/s    wMB/s avgrq-sz avgqu-sz   await r_await w_await  svctm  %util
sda               0,00     0,00    0,00    0,00     0,00     0,00     0,00     0,00    0,00    0,00    0,00   0,00   0,00

Device:         rrqm/s   wrqm/s     r/s     w/s    rMB/s    wMB/s avgrq-sz avgqu-sz   await r_await w_await  svctm  %util
sda               0,00     8,00    0,00   15,00     0,00     0,07    10,13     0,00    0,27    0,00    0,27   0,27   0,40
```

Figure 6.10: I/O statistics when reading one hundred thousand rows in Cassandra

Hot Spots - Method	Self Time [%]	Self Time	Total Time ▼
org.apache.cassandra.cql3.statements.JoinStatement.**execute** (org.apache.ca...		0.014 ms (0%)	490,225 ms
org.apache.cassandra.cql3.statements.JoinStatement.**execute** (org.apache.ca...		0.051 ms (0%)	490,225 ms
org.apache.cassandra.cql3.statements.JoinStatement.**executeJoin** (java.util.A...		0.487 ms (0%)	490,225 ms
org.apache.cassandra.cql3.statements.JoinStatement.**nested_loop_join** (org....		457 ms (0.4%)	490,224 ms
org.apache.cassandra.cql3.statements.JoinStatement.**executeJoinRows** (org....		34.9 ms (0%)	484,241 ms
org.apache.cassandra.service.StorageProxy.**getRangeSlice** (org.apache.cass...		2,239 ms (1.8%)	450,704 ms
org.apache.cassandra.service.ReadCallback.**get** ()		54.5 ms (0%)	51,157 ms
org.apache.cassandra.service.ReadCallback.**await** (long, java.util.concurrent.T...		71.9 ms (0.1%)	48,279 ms
org.apache.cassandra.utils.SimpleCondition.**await** (long, java.util.concurrent.T...		48,207 ms (39.5%)	48,207 ms
org.apache.cassandra.service.StorageProxy.**read** (java.util.List, org.apache.ca...		76.0 ms (0.1%)	33,502 ms
org.apache.cassandra.service.StorageProxy.**fetchRows** (java.util.List, org.apa...		127 ms (0.1%)	32,595 ms
org.apache.cassandra.service.AbstractReadExecutor.**get** ()		9.61 ms (0%)	28,633 ms
org.apache.cassandra.service.StorageProxy.**getLiveSortedEndpoints** (org.a...		1,032 ms (0.8%)	22,237 ms
org.apache.cassandra.service.StorageProxy.**getRestrictedRanges** (org.apac...		1,519 ms (1.2%)	21,402 ms
org.apache.cassandra.service.StorageService.**getLiveNaturalEndpoints** (or...		1,408 ms (1.2%)	19,844 ms
org.apache.cassandra.locator.AbstractReplicationStrategy.**getNaturalEndpoints**		888 ms (0.7%)	17,866 ms
org.apache.cassandra.locator.TokenMetadata.**firstToken** (java.util.ArrayList, o...		383 ms (0.3%)	11,134 ms

Figure 6.11: This figure shows CPU profiling for an N-Way-Join on one hundred thousand rows

are issued and used in other parts of Cassandra for retrieving the correct rows. All other methods that are seen in Figure 6.11 that does not belong to the class JoinStatement is not implemented in this master thesis.

When investigating the code, it is possible to see that there are two possible read paths. Either through the method getRangeSlice or read, where the first applies to all commands where it is a restriction in the joinon caluse on the secondary index or no restrictions in the joinon clause at all. The latter one applies to all commands where there exists a restriction in the joinon clause on a primary key.

Both methods end up in the same method await. This method can be seen in Figure 6.11. This method has one of the highest self times compared to the others.

The await method has a while loop where there are two conditions, one if the rows have been found (a boolean variable), and a timeout check. The while loop iterates when the boolean variable is false and the elapsed time has not exceeded the timeout value. However, the while loop aborts when the boolean variable is set to true. This happens when the signalAll method (in the same class as the await method) is called. Both the await and signalAll method are synchronized methods.

During each read for the inner rows, a Runnable object is created and managed by the ThreadPoolExecutor. This runnable will manage the read, and when it is finished, the signaAll method described above will be called.

The problem is that for each outer row in the nested loop, a read or getRangeSlice method call is issued. Each of these calls must scan the table for the correct rows to finish. The total time will, therefore, be significantly higher compared to MySQL. It can also be seen from the joins with a where relation, that Cassandra improves quite drastically. However, compared to MySQL, Cassandra performs poorly. This is because many rows have to be scanned when the size of a column family is large. The reason for the reduction of execution time is because the number of outer rows is reduced. As seen from the results for a join statement with a projection, the execution is almost unchanged from the standard join statement. This is because it is the same number of rows that is iterated in the nested loop including all the scans.

The await method described above has a big self time. This is because the program must wait on the result to be fetched from the memory or disk. When each read uses some nanoseconds or 1 millisecond to fetch some rows, combined with the number of reads that is necessary, the total time for completing a join will have a longer execution time than MySQL. This is because the underlying architecture in Cassandra was not implemented by the creators with a join opportunity in their thoughts. It is possible that a join could be more efficient if there were more support for this in the underlying architecture.

6.5.4 Query optimization

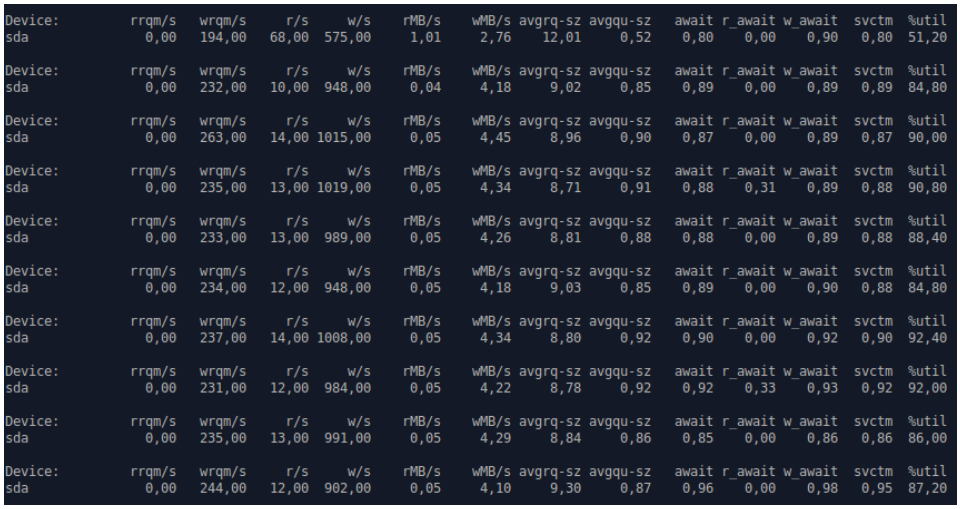
No special effort has been invested to create a query optimizer to avoid the worst plan. However, there are one optimization that has been implemented. If there is a where relation on a column in a column family that is in the join, this column family is the first one to be executed. It is likely that this where clause removes many rows from the result. It is, therefore, advantageously to execute this first to avoid unnecessary iterations in the nested loop. As Bratbergsengen [23] describes, it is beneficial to reduce the operand volume as early as possible.

However, the ordering of the join is random, in terms of how the user enters the restrictions in the joinon clause. If the user is lucky, an optimal or suboptimal ordering is chosen. However, this is not a good idea since the user does not know which ordering that creates a plan that is not the worst. As described in Chapter 2.1, it is more important to avoid the worst plan and creating a suboptimal plan. This does not happen in this implementation, because of the lack of statistics to create this plan. This may affect the result because the worst plan may iterate through more rows than other suboptimal plans.

6.5.5 Other observations

During this evaluation, the select statement and insert statement was also monitored and the results can be seen in Chapter 6.4.4 and 6.4.5, respectively.

As can be seen from the results in Chapter 6.4.5 (insertion), Cassandra is better than MySQL when it comes to insertions. One reason for this can be the use of memtables in Cassandra while MySQL stores the data on a disk. Since the heap size for Cassandra is set quite high, the memtables are not flushed to SSTables and disk I/O is therefore avoided. As you can see in Figure 6.13, the I/O for Cassandra is less than the I/O for MySQL, shown in Figure 6.12. However, the insert queries written for MySQL may not be the best suited for the insertion. A different approach, like executing batches of insertions, may have decreased the execution time for MySQL (because less transaction management is needed).



Device:	rrqm/s	wrqm/s	r/s	w/s	rMB/s	wMB/s	avgrq-sz	avgqu-sz	await	r_await	w_await	svctm	%util
sda	0,00	194,00	68,00	575,00	1,01	2,76	12,01	0,52	0,80	0,00	0,90	0,80	51,20
Device:	rrqm/s	wrqm/s	r/s	w/s	rMB/s	wMB/s	avgrq-sz	avgqu-sz	await	r_await	w_await	svctm	%util
sda	0,00	232,00	10,00	948,00	0,04	4,18	9,02	0,85	0,89	0,00	0,89	0,89	84,80
Device:	rrqm/s	wrqm/s	r/s	w/s	rMB/s	wMB/s	avgrq-sz	avgqu-sz	await	r_await	w_await	svctm	%util
sda	0,00	263,00	14,00	1015,00	0,05	4,45	8,96	0,90	0,87	0,00	0,89	0,87	90,00
Device:	rrqm/s	wrqm/s	r/s	w/s	rMB/s	wMB/s	avgrq-sz	avgqu-sz	await	r_await	w_await	svctm	%util
sda	0,00	235,00	13,00	1019,00	0,05	4,34	8,71	0,91	0,88	0,31	0,89	0,88	90,80
Device:	rrqm/s	wrqm/s	r/s	w/s	rMB/s	wMB/s	avgrq-sz	avgqu-sz	await	r_await	w_await	svctm	%util
sda	0,00	233,00	13,00	989,00	0,05	4,26	8,81	0,88	0,88	0,00	0,89	0,88	88,40
Device:	rrqm/s	wrqm/s	r/s	w/s	rMB/s	wMB/s	avgrq-sz	avgqu-sz	await	r_await	w_await	svctm	%util
sda	0,00	234,00	12,00	948,00	0,05	4,18	9,03	0,85	0,89	0,00	0,90	0,88	84,80
Device:	rrqm/s	wrqm/s	r/s	w/s	rMB/s	wMB/s	avgrq-sz	avgqu-sz	await	r_await	w_await	svctm	%util
sda	0,00	237,00	14,00	1008,00	0,05	4,34	8,80	0,92	0,90	0,00	0,92	0,90	92,40
Device:	rrqm/s	wrqm/s	r/s	w/s	rMB/s	wMB/s	avgrq-sz	avgqu-sz	await	r_await	w_await	svctm	%util
sda	0,00	231,00	12,00	984,00	0,05	4,22	8,78	0,92	0,92	0,33	0,93	0,92	92,00
Device:	rrqm/s	wrqm/s	r/s	w/s	rMB/s	wMB/s	avgrq-sz	avgqu-sz	await	r_await	w_await	svctm	%util
sda	0,00	235,00	13,00	991,00	0,05	4,29	8,84	0,86	0,85	0,00	0,86	0,86	86,00
Device:	rrqm/s	wrqm/s	r/s	w/s	rMB/s	wMB/s	avgrq-sz	avgqu-sz	await	r_await	w_await	svctm	%util
sda	0,00	244,00	12,00	902,00	0,05	4,10	9,30	0,87	0,96	0,00	0,98	0,95	87,20

Figure 6.12: I/O statistics when inserting one hundred thousand rows in MySQL

Another aspect that is important is the use of ACID properties (described in Chapter 2.5.2) in MySQL. This makes the insertion slower because MySQL must ensure that data inserted is compliant with the ACID properties. The RDBMS ACID transactions (with rollback and locking mechanisms) are not used in Cassandra. However, it offers atomic, isolated and durable transactions where it is eventual/tunable consistency. Since no consistency level has been set during the evaluation, the default consistency level is ONE [6]. This means that the data only has to be written to a memtable and commit log at one replica [20]. This is the consistency level that has the highest availability, but also the lowest consistency. This can explain some of the insertion times for Cassandra compared to MySQL. Another reason for why Cassandra is faster than MySQL is that the data inserted in Cassandra is written to memtables and commit log without having to flush to the disk all the time. Another point that can make Cassandra faster is that log-structured merge trees are used on the disk, which implies that all writes are done

```
Device:         rrqm/s   wrqm/s     r/s     w/s    rMB/s    wMB/s avgrq-sz avgqu-sz   await r_await w_await  svctm  %util
sda               0,00   334,00    9,00  135,00     0,18     1,91    29,61     0,16    1,11    0,00    1,19   1,03  14,80

Device:         rrqm/s   wrqm/s     r/s     w/s    rMB/s    wMB/s avgrq-sz avgqu-sz   await r_await w_await  svctm  %util
sda               0,00    24,00    5,00    3,00     0,47     0,16  162,00     0,01    1,00    0,80    1,33   1,00   0,80

Device:         rrqm/s   wrqm/s     r/s     w/s    rMB/s    wMB/s avgrq-sz avgqu-sz   await r_await w_await  svctm  %util
sda               0,00    13,00    3,00   36,00     0,38    16,71  897,23     0,67   17,23    0,00   18,67   1,13   4,40

Device:         rrqm/s   wrqm/s     r/s     w/s    rMB/s    wMB/s avgrq-sz avgqu-sz   await r_await w_await  svctm  %util
sda               0,00   206,00    4,00  118,00     0,50     1,27    29,64     0,00    0,03    1,00    0,00   0,03   0,40

Device:         rrqm/s   wrqm/s     r/s     w/s    rMB/s    wMB/s avgrq-sz avgqu-sz   await r_await w_await  svctm  %util
sda               0,00     0,00    3,00    0,00     0,38     0,00  256,00     0,00    1,33    1,33    0,00   1,33   0,40

Device:         rrqm/s   wrqm/s     r/s     w/s    rMB/s    wMB/s avgrq-sz avgqu-sz   await r_await w_await  svctm  %util
sda               0,00     0,00    4,00    2,00     0,50     0,01  173,33     0,00    0,00    0,00    0,00   0,00   0,00

Device:         rrqm/s   wrqm/s     r/s     w/s    rMB/s    wMB/s avgrq-sz avgqu-sz   await r_await w_await  svctm  %util
sda               0,00     0,00    4,00    0,00     0,50     0,00  256,00     0,00    1,00    1,00    0,00   1,00   0,40

Device:         rrqm/s   wrqm/s     r/s     w/s    rMB/s    wMB/s avgrq-sz avgqu-sz   await r_await w_await  svctm  %util
sda               0,00    15,00    3,00    2,00     0,38     0,07  180,80     0,01    2,40    2,67    2,00   2,40   1,20

Device:         rrqm/s   wrqm/s     r/s     w/s    rMB/s    wMB/s avgrq-sz avgqu-sz   await r_await w_await  svctm  %util
sda               0,00     0,00    4,00    0,00     0,50     0,00  256,00     0,00    1,00    1,00    0,00   1,00   0,40

Device:         rrqm/s   wrqm/s     r/s     w/s    rMB/s    wMB/s avgrq-sz avgqu-sz   await r_await w_await  svctm  %util
sda               0,00    10,00    3,00    2,00     0,28     0,05  136,00     0,00    0,00    0,00    0,00   0,80   0,40

Device:         rrqm/s   wrqm/s     r/s     w/s    rMB/s    wMB/s avgrq-sz avgqu-sz   await r_await w_await  svctm  %util
sda               0,00   414,00    6,00  181,00     0,25     2,41    29,05     0,19    1,05    0,00    1,08   0,96  18,00
```

Figure 6.13: I/O statistics when inserting one hundred thousand rows in Cassandra

sequentiality. For MySQL and InnoDB, a B+-tree is used. This may give more random disk seeks during a write, and this may cause the higher execution time for MySQL. However, since all rows are stored with auto-incremented primary keys, there should not be so many random disk accesses.

Another query that was evaluated was a simple select statement. Figure 6.6 shows that Cassandra and MySQL have the same execution time up to one hundred rows. At one thousand and ten thousand rows, MySQL performs better than Cassandra. However, at one hundred thousand rows and up to one million rows, Cassandra performs better than MySQL. It is a bit unclear why Cassandra is better than MySQL at one million rows. Running the iostat command does not show any reads or writes that can explain the higher execution time for MySQL. However, the difference is not significant, and a possible explanation is the use of memtables in Cassandra. However, this is not so important.

With the results from the join and select and insert statements, it is possible to see that the implementation in this thesis does something wrong. As discussed in Chapter 6.5.3, the main problem is the number of threads created where each scans a table. With a select without any where relations, it possible to just read the whole column family and return it. With the current join implementation, many scans are needed, and this makes the execution time slow.

Chapter 7

Conclusion

This chapter presents the different contributions to the research field of join and a conclusion that will try to answer the goals described in Chapter 1.2.

7.1 Contributions

This thesis shows that it is possible to implement the join operation in Cassandra. This may encourage other developers to implement support for the join operation in other NoSQL systems. However, this implementation is currently only supported on one node. This should not be seen as an obstacle, but as a possibility to implement support for join in a NoSQL where multiple nodes are deployed. It must be possible to execute the join on a single node before it can be implemented as a distributed solution

7.2 Conclusion

The first goal of this thesis was to implement support for the join operation on one node in Cassandra. This goal was successfully reached. The grammar for CQL was changed to support the expression of equijoins in a query. Besides this, an index nested loop join was implemented in the source code of Cassandra. The implemented solution for this thesis also supports multiple N-Way-Joins including queries that contain both join and column families outside any join. When a join query is prepared for execution, an appropriate join ordering is found. However, this join ordering is not optimized in any way, except column families used in the join and the where clause. These column families are executed early on in the process. The implemented code was separated in an own class. Comparison of a join result in both MySQL and Cassandra shows the same output data.

The implemented join operation was evaluated with different types of join queries. This evaluation shows that there are multiple problems that need to

be addressed before this can be a production ready functionality in Cassandra. If the join only contains the join restrictions or join restrictions with projection, the execution time is much higher for Cassandra than MySQL. However, the execution time is significantly reduced if there exist selections. It turned out that the main problem was and is that the underlying architecture does not work well with a join operation. The only reason for why join queries with selections have a better execution time is that the number of rows is stripped away early. This time is still high compared to MySQL. For each outer row in the index nested loop join, a thread is started, and this thread executes a scan on the inner column family. This method increases the execution time greatly.

The conclusion is that it is possible to execute a join in a NoSQL system, but the current implementation does not utilize the architecture of Cassandra in an efficient way. Further work is therefore needed to decrease the execution time. Besides this, the current and further implementation should support a join over multiple nodes.

Chapter 8

Further work

This implementation of join in Cassandra is only a proof of concept. As seen in Chapter 6, Cassandra performs slower than MySQL. It is, therefore, necessary to pinpoint some further work on this implementation. This chapter looks at some subjects that may be interesting to work with in the further. This work may include distributed join, better query optimization, and fault tolerance including the use of intermediate storage if the memory usage is big.

8.1 Support different join types

Only equijoin is supported in this prototype. As further work, an implementation that supports all the different join types described in Chapter 2.3.1 should be made. A user will expect that the join types are supported by Cassandra since most of the database systems using SQL supports them. Since data is distributed over multiple nodes at different locations, different strategies may be chosen.

8.2 Distributed join

Today, no effort has been put into making the implemented join operation work on multiple nodes. Since Cassandra is designed as a distributed data store, distributed joins must be supported if the join operation is going to be seen as a standard operation. Bratbergsengen [23] presents a partition-based algorithm where each node executes the relational algebra locally. When performing a join query in a distributed system, each node can then hash the attribute that is joined and send the row to the correct node (also referred to as meeting place). At the meeting node, the join can then be executed. This algorithm shows that it is possible to implement a join functionality in a distributed system.

8.3 Query optimization

A small optimization was implemented in this thesis. If there is a selection on a column family in a join, this column family is first in the join order. Then a lot of potential data is removed early in the joining process. This small optimization decreased the execution time significantly. However, no other query optimization techniques are implemented in the current version of the join operation in Cassandra. Query optimization (described in Chapter 2.1.1) should be implemented with the purpose of finding a optimal or suboptimal execution plan. With this plan, the cost of execution most likely is cheaper than choosing a bad plan. Today, the most ineffective join ordering may be used. However, a suboptimal or optimal ordering should be used instead, and this ordering can be found using optimization. Besides supporting optimization on one node, distributed optimization should be supported if there are multiple nodes involved. Hevner and Yao [34] presents an algorithm for query processing in distributed database systems. This shows that it is possible to implement query processing in a distributed database system, similar to Cassandra.

8.4 Efficiency

The efficiency of the current join implementation is bad and should be improved significantly before it can be used by real users. Today, an index nested loop join is used to perform the join. Instead of using this algorithm, a block nested loop join algorithm could have been used. Then a data structure (for example a hash table pointing to different binary trees) with a bloom filter could be used to store blocks from the outer row. The inner relation can then be scanned for each block from the outer relation and matched against this data structure. Unnecessary scans (which returns few rows) of the inner relation (as the current implementation does) could have been avoided. There is probably other methods that could be used to increase the efficiency of this implementation. The most beneficial solution would have been to redesign the architecture so that a join could be supported while maintaining good horizontal scalability.

8.5 Fault tolerance

There are many possible ways the current implementation of the join operation can crash. If this implementation is going to be used by real users, it is expected that errors can happen. However, today the system can crash if a query is written wrong and this should not occur. For example, if a column family is used in the join, but the user forgot to type it in the from clause, the Cassandra server will crash with a "TSocket read 0 bytes" message returned to the user. Instead of crashing, the Cassandra server should return an understandable error message explaining that the column family is missing in the from clause.

8.6 Temporary storage

In the current implementation, all data in the join is matched and stored in memory. However, if the amount of data is bigger than the memory size, a solution for storing temporary data on disk should be possible. The memory is a limited resource compared to the disk space available and avoiding memory leakage is important.

8.7 Support all key types in Cassandra

Only single primary keys and secondary indexes are supported in a join operation. However, a join operation in Cassandra should also support composite partition key and compound primary key (described in Chapter 2.7.2) that can be used in Cassandra.

Bibliography

[1] ANTLR parser generator. http://www.antlr3.org/. Accessed: 2014-9-28.

[2] Apache Ant. http://ant.apache.org/. Accessed: 2014-9-29.

[3] The apache cassandra project. http://cassandra.apache.org/. Accessed: 2014-11-26.

[4] Apache CouchDB. http://couchdb.apache.org/. Accessed: 2015-5-5.

[5] Cassandra Query Language. https://cassandra.apache.org/doc/cql3/CQL.html. Accessed: 2015-1-19.

[6] DataStax CQL 3.1.x documentation. http://www.datastax.com/documentation/cql/3.1/cql/cql_intro_c.html. Accessed: 2015-1-22.

[7] GitHub. https://github.com/. Accessed: 2014-9-1.

[8] HBase. http://hbase.apache.org/. Accessed: 2015-5-5.

[9] Hypertable. http://hypertable.org/. Accessed: 2015-5-5.

[10] Implementation code for join in Cassandra. https://github.com/chrpeter/Masteroppgave.

[11] Java garbage collection basics. http://www.oracle.com/webfolder/technetwork/tutorials/obe/java/gc01/index.html. Accessed: 2015-4-14.

[12] Java Virtual Machine Statistics Monitoring Tool. http://docs.oracle.com/javase/7/docs/technotes/tools/share/jstat.html. Accessed: 2015-4-14.

[13] MongoDB. https://www.mongodb.org/. Accessed: 2015-5-5.

[14] MySQL: The world's most popular open source database. https://www.mysql.com/. Accessed: 2015-5-4.

[15] Planning your data model. `http://www.datastax.com/docs/1.0/ddl/data_model_planning`. Accessed: 2015-2-27.

[16] Redis. `http://redis.io/`. Accessed: 2015-5-5.

[17] Riak – basho technologies. `http://basho.com/riak/`. Accessed: 2015-5-5.

[18] Source code for data generator and benchmarking. `https://github.com/chrpeter/Cassandra_and_mysql_tester`.

[19] Voldemort. `http://www.project-voldemort.com/voldemort/`. Accessed: 2015-5-5.

[20] *Apache Cassandra 2.0 Documentation*, 14 January 2015.

[21] Veronika Abramova and Jorge Bernardino. NoSQL databases: MongoDB vs Cassandra. In *Proceedings of the International C* Conference on Computer Science and Software Engineering*, pages 14–22. ACM, 10 July 2013.

[22] Apache. Apache/Cassandra. `https://github.com/apache/cassandra/tree/cassandra-2.0/`, 25 August 2014. Accessed: 2014-9-12.

[23] Kjell Bratbergsengen. *Storing and Management of Large Data Volumes.* 11 June 2014.

[24] Andre Calil and Ronaldo dos Santos Mello. SimpleSQL: A relational layer for SimpleDB. In *Advances in Databases and Information Systems*, Lecture Notes in Computer Science, pages 99–110. Springer Berlin Heidelberg, 1 January 2012.

[25] Rick Cattell. Scalable SQL and NoSQL data stores. *SIGMOD Rec.*, 39(4):12–27, May 2011.

[26] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Trans. Comput. Syst.*, 26(2):4:1–4:26, June 2008.

[27] David J. DeWitt, Randy H. Katz, Frank Olken, Leonard D. Shapiro, Michael R. Stonebraker, and David A. Wood. Implementation Techniques for Main Memory Database Systems. In *Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data*, SIGMOD '84, pages 1–8, New York, NY, USA, 1984. ACM.

[28] Jonathan Ellis. Facebook's Cassandra paper, annotated and compared to Apache Cassandra 2.0.

[29] Jonathan Ellis. Architecture Internals. `https://wiki.apache.org/cassandra/ArchitectureInternals`, 16 November 2013. Accessed: 2015-1-16.

[30] Ramez Elmasri and Shamkant B. Navathe. *Database Systems: Models, Languages, Design, and Application Programming.* Pearson, sixth edition, 2011.

[31] Elvis C. Foster and Shripad V. Godbole. SQL data manipulation statements. In *Database Systems*, pages 219–258. Apress, 2014.

[32] Seth Gilbert and Nancy Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *ACM SIGACT News*, 33(2):51–59, 1 June 2002.

[33] Jing Han, E Haihong, Guan Le, and Jian Du. Survey on NoSQL database. In *Pervasive Computing and Applications (ICPCA), 2011 6th International Conference on*, pages 363–366. ieeexplore.ieee.org, October 2011.

[34] Alan R. Hevner and S. Bing Yao. Query processing in distributed database system. *IEEE Trans. Software Eng.*, SE-5(3):177–187, May 1979.

[35] Michael Kruckenberg and Jay Pipes. Essential SQL. In *Pro MySQL*, pages 235–297. Apress, 2005.

[36] Avinash Lakshman and Prashant Malik. Cassandra: A decentralized structured storage system. *Oper. Syst. Rev.*, 44(2):35–40, April 2010.

[37] Ramon Lawrence. Integration and virtualization of relational SQL and NoSQL systems including MySQL and MongoDB. In *Computational Science and Computational Intelligence (CSCI), 2014 International Conference on*, volume 1, pages 285–290, March 2014.

[38] Neal Leavitt. Will NoSQL databases live up to their promise? *Computer*, 43(2):12–14, February 2010.

[39] Priti Mishra and Margaret H Eich. Join Processing in Relational databases. *ACM Comput. Surv.*, 24(1):63–113, March 1992.

[40] Vivek Mishra. Cassandra data modeling. In *Beginning Apache Cassandra Development*, pages 27–42. Apress, 2014.

[41] Karen Morton, Kerry Osborne, Roby Sands, Riyaj Shamsudeen, and Jared Still. *Pro Oracle SQL.* Expert's Voice in Oracle. Apress, 2013.

[42] Sasha Pachev. *Understanding MySQL Internals.* O'Reilly Media, Inc., April 2007.

[43] Terence Parr and Kathleen Fisher. LL(*): the foundation of the ANTLR parser generator. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, volume 46, pages 425–436. ACM, 4 June 2011.

[44] Dan Pritchett. BASE: An acid alternative. *Queueing Syst.*, 6(3):48–55, 1 May 2008.

[45] Tilmann Rabl and Hans-Arno Jacobsen. Materialized Views in Cassandra. In *Proceedings of 24th Annual International Conference on Computer Science and Software Engineering*, CASCON '14, pages 351–354, Riverton, NJ, USA, 2014. IBM Corp.

[46] Raghu Ramakrishnan and Johannes Gehrke. *Database Management Systems*. McGraw-Hill, third edition, 2003.

[47] Patricia G. Selinger, Morton M. Astrahan, Donald D. Chamberlin, Raymond A. Lorie, and Thomas G. Price. Access Path Selection in a Relational Database Management System. In *Proceedings of the 1979 ACM SIGMOD International Conference on Management of Data*, SIGMOD '79, pages 23–34, New York, NY, USA, 1979. ACM.

[48] Rick F. Van der Lans. Introduction to SQL. In *Introduction to SQL: Mastering the Relational Database Language, Fourth Edition/20th Anniversary Edition*, pages 3–28. Addison-Wesley Professional, September 2006.

[49] Xite Wang, Derong Shen, Tiezheng Nie, Yue Kou, and Ge Yu. The Equi-Join processing and optimization on ring architecture Key/Value database. In *Web Technologies and Applications*, Lecture Notes in Computer Science, pages 243–254. Springer Berlin Heidelberg, 1 January 2012.

[50] Zhou Wei, Guillaume Pierre, and Chi-Hung Chi. Scalable join queries in cloud data stores. In *Proceedings of the 2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (ccgrid 2012)*, pages 547–555. IEEE Computer Society, 13 May 2012.