UNIVERSITÉ LIBRE DE BRUXELLES ULB

INFO-H415 - Advanced Databases
NoSQL databases and Cassandra

Lev Denisov	000454497
Todi Thanasi	000455013

December 17, 2017

Contents

List of Figures

List of Tables

Introduction

This work is a review of Cassandra database and its comparison with Microsoft SQL Server. The structure of this report is the following: in section 1 we make a minimal overview of Cassandra basic principles required for understanding the differences from traditional relational databases in design and usage. In section 2 we describe our experiment design and used software and hardware. In section 3 we discuss the experiment results and compare the performance of Cassandra and Microsoft SQL Server.

1 Cassandra overview

1.1 NoSQL vs traditional RDBMS. ACID and CAP theorem

In traditional relational databases the important properties can be summarized in one abbreviation: ACID, which means Atomicity, Consistency, Isolation and Durability. Atomicity means that every transaction either completes or fails as a whole, no partial failures. Consistency means that after completion of a transaction, the data in the database is in valid state. Isolation means that all the transaction running at the same time do not affect each other and their result is the same as if they were running one after another. And finally, Durability means that after transaction has completed, the data will remain in its state no matter of the future failures, such as power outage or server failure.

All these properties are very desirable for any system but maintaining them comes at its cost. In case of one server the overhead is not that big, and maintaining ACID is relatively easy. The problem is that today many systems such as popular online stores or logging systems need to support a huge amount of read and/or write requests, making it impossible to run such system on a single machine because the resources of one machine are not very scalable. Another case when it might be desirable to use multiple machines is to make a system more resilient to unforeseen events, such as natural disasters or to reduce the time the users must wait for the response, by placing servers closer to them. In both these cases the system works on multiple computers, sometimes distributed between the continents. In these conditions it becomes very hard to maintain ACID, because every transaction must propagate to all the servers before next request can be made, incurring considerable delays and consuming resources. Eric Brewer proposed CAP theorem that states that in a distributed system, of three properties Consistency, Availability and Partition tolerance, only two can be guaranteed at the same time. In the context of CAP theorem, Consistency means that no matter what machine serves the request, the response should contain up-to-date data or an error. Availability means guarantee of getting the response, which may be the result of the query or an error. And Partition tolerance means that the system should be able to continue work even if one or a few machines in the cluster stopped responding or some messages were lost.

In accordance with the requirements, different systems may choose to strongly maintain different CAP properties. The architectural choice in Cassandra was to maintain high Availability and Partition tolerance at the cost of Consistency. In Cassandra, changes may take some time to propagate to all nodes, which is called Eventual Consistency. During the time changes propagate, some nodes may give old data as a response. To give some flexibility, Cassandra allows to specify the number of nodes with replicas that should agree on the response for the given request, which means that Cassandra supports variable consistency, depending on the importance of the consistency for each request.

1.2 Overview of replication strategies

To allow high availability and partition tolerance even in event of some nodes failing, the data must be replicated, which means that there are more that one copy of each data row, preferably on the nodes that are unlikely to fail at the same time. There is a parameter Replication Factor that defines how many copies of the data will be stored on the nodes of the cluster. More copies mean more nodes that cluster can lose without losing the data, but of course it comes at a cost of disk space. There are two replication strategies in Cassandra: SimpleStrategy and NetworkTopologyStrategy. SimpleStrategy is good when the whole cluster is in the same datacenter. The replicas are distributed between nodes in a ring without the consideration of the locality of the node. More advanced strategy is NetworkTopologyStrategy which is good for multi-datacenters setups. It makes sure that there are replicas of data in each of the datacenters, allowing to lose the whole datacenter without losing data. Also, this strategy allows to specify replication factor for each datacenter separately, which means that in case of some nodes failing in one datacenter, it can still serve requests without asking another datacenter.

1.3 Partitioning in Cassandra

To support horizontal scalability, the data must be distributed between all the nodes of the cluster, so each node can handle requests independently. Depending on the architecture of DB engine, there may be one or multiple special nodes called masters, that coordinate the distribution process between slave nodes. Failure of master node can cause failure of the whole cluster. To provide high Availability, all Cassandra nodes have equal responsibilities, which means that there is no single point of failure. Each node gets a part of data based on the primary key. The distribution of data between nodes is important, because if most writes or reads concentrate on one or a few nodes, it causes a hotspot, meaning slower responses from loaded nodes while most of the resources is idle. To distribute the load, different partition strategies may be used. The naïve approach is to use the value of the key itself to do the partitioning. This approach is implemented in Cassandra ByteOrderedPartitioner strategy. This strategy can be used for the ordered data and data assigned to the node according to the bytes of the key. This strategy allows range scans, since there is a clear dependence between the key and the node. However, this approach has a downside. In real applications, keys are often distributed unevenly, for example last names or timestamps. This means that there will be hotspots. Another, more preferred approach is to use a hash function to get a more even distribution of keys. This approach is implemented in Murmur3Partitioner strategy. This strategy does not allow effective range queries, since there is no clear dependency between the value of the key and its hash. This strategy is preferred in most applications as it gives better load distribution, while range queries are facilitated by secondary indexes.

1.4 Overview of Cassandra data model

Since Cassandra is not a traditional relational database, it has its own ways to model the data.

On the top level, there are Column Families which are in a way similar to tables in RDBMS: they contain rows and group closely related data. The main difference from tables is that Column Families do not describe the format of data they hold. Each Column Family has comparator, that defines the order in which columns will be returned to the user.

On the level below the Column Families, contained in them are Rows. Rows are containers for columns; they have an ID, that is unique and is used as a primary key as well as the partitioning key. Rows within the same Column Family are not required to have the same set of columns. In Cassandra, there are two distinct patterns of using rows. First is similar to traditional RDBMS approach when each Row contains relatively small amount of known in advance keys. Such rows are called Skinny rows. Another pattern is called wide row. Wide rows contain a large number of dynamic columns; the number of columns can vary from thousands to millions. As the partitioning happens on the row level, each row has to fit on the disc of one node.

Columns are the actual containers for the data. Each column has a name, value and a timestamp. Name is used by the Column Family comparator to define the order in which columns are stored and retrieved. Since it may be impractical to load the whole row into memory, the order of columns is important, as it allows Cassandra to retrieve only relevant slices of rows. There are multiple built-in comparators that define different orders. Custom comparators are supported as well. Columns values can be simple types as integers or strings or collections such as lists, maps and sets.

1.5 Short overview of CQL commands

Cassandra has its own language CQL which supports similar to SQL set of commands. The examples of usage are taken from the official documentation for Cassandra [1].

1.5.1 Data definition commands.

CREATE KEYSPACE is used to create a new keyspace. ALTER KEYSPACE is used to modify an existing keyspace. DROP KEYSPACE is used to delete an existing keyspace. Example of usage is given in the Listing 1.

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

ALTER KEYSPACE Excelsior
WITH replication = {'class': 'SimpleStrategy', '
   replication_factor': 4};

DROP KEYSPACE Excelsior;
```

Listing 1: Create/Alter/Drop Keyspace in Cassandra

CREATE TABLE is used to create a table (Column Family). ALTER TABLE is used to modify an existing table. DROP TABLE is used to delete a table. The example is given in the Listing 2.

```
CREATE TABLE monkeySpecies (
    species text PRIMARY KEY,
    common_name text,
    population varint,
    average_size int
) WITH comment='Important␣biological␣records'
   AND read_repair_chance = 1.0;

ALTER TABLE monkeySpecies ADD average_weight varchar;

DROP TABLE monkeySpecies
```

Listing 2: Create/Alter/Drop Table in Cassandra

1.5.2 Secondary indexes.

CREATE INDEX is used to create a secondary index on a table. DROP INDEX is used to delete a secondary index. The example is given in the Listing 3.

```
CREATE INDEX favsIndex ON users (keys(favs));

DROP INDEX favsIndex
```

Listing 3: Create/Drop Index in Cassandra

1.5.3 Data manipulation commands.

SELECT is used for querying data. Only primary key or fields with secondary indexes on them can be used inside WHERE clause. The example is given in the Listing 4.

```
SELECT name, occupation FROM users WHERE userid IN (199, 200,
    207);

SELECT time, value
FROM events
WHERE event_type = 'myEvent'
  AND time > '2011-02-03'
  AND time <= '2012-01-01'
  LIMIT 100

SELECT COUNT (*) AS user_count FROM users;
```

Listing 4: Select records in Cassandra

INSERT is used to insert data for a row. UPDATE is used to modify the row. DELETE is used to delete rows or parts of rows. The example is given in the Listing 5.

```
INSERT INTO Movies (movie, director, main_actor, year)
                VALUES ('Serenity', 'Joss␣Whedon', 'Nathan␣
                   Fillion', 2005)

UPDATE Movies USING TTL 400
    SET director   = 'Joss␣Whedon',
        main_actor = 'Nathan␣Fillion',
        year       = 2005
    WHERE movie = 'Serenity';

DELETE FROM Movies USING TIMESTAMP 1240003134
    WHERE movie = 'Serenity';
```

Listing 5: Insert/Update/Delete records in Cassandra

BATCH is used to group multiple INSERT, UPDATE or DELETE commands to execute them as a single statement to save network round-trips.The example is given in the Listing 6.

```
BEGIN BATCH
   INSERT INTO users (userid, password, name) VALUES ('user2'
      , 'ch@ng', 'second');
   UPDATE users SET password = 'ps22dhds' WHERE userid = '
      user3';
   INSERT INTO users (userid, password) VALUES ('user4', '
      ch@ng');
   DELETE name FROM users WHERE userid = 'user1';
APPLY BATCH;
```

Listing 6: Batch command in Cassandra

1.6 Data modeling in Cassandra

Data modeling in Cassandra is different from data modeling in traditional RDBMS. The reason for that is in contrast with relational database Cassandra cannot perform just any ad-hoc query since it does not have joins and can only filter by indexed fields. Thus, database architect should plan in advance how to store the data in a way that the required queries are possible and efficient. The internal representation of data in Cassandra is often described as "Map of sorted maps". The outer map is a column family that has rows with their primary keys as keys of the hash. The inner map is row with columns, that are sorted by their name. This allows efficient range queries on columns if their names are constructed from the values that they hold. In fact, it is a common pattern to have "valueless columns" when all the data needed is placed in column name. Since Cassandra does not have joins, it is very common practice to denormalize data, for example to have special column families for the joined data. It is often the case that for almost each query type there is a separate table in Cassandra database. There is a special case of data modeling often used in Cassandra: time series. There are a few features that timeseries data has:

1. Time series data is represented as pair (timestamp, value)
2. Often, timeseries are written at a high rate
3. Timeseries should be ordered by time
4. Queries on timeseries usually include ranges by time

Timeseries data can be efficiently modeled in Cassandra wide rows. Each event in timeseries is represented by one column in a row for the particular metric. Each column name includes timestamp of the event to exploit column sorting in Cassandra. One of the problems that this model has is that rows can be overflown by the number of events. Another problem is that writing in one row can cause a hot spot, reducing performance. Both problems are solved by additional partitioning, i.e. additional information in primary key. The problem with overflowing data is solved by creating a new row for each time interval (for example hour, day or month) depending on the data rate. In this case each row has only limited number of data points and does not grow infinitely. Also, it allows to discard old data, leaving only aggregates. The problem with hot spots is solved by dividing data into buckets by some natural or artificial property. One of the examples of this is dividing events stream on warnings, errors and info messages, each of which is stored in a separate row. Another approach is to create an artificial attribute that would divide data into N buckets (can be sequential number mod N).

1.7 Cassandra vs MSSQL

Companies all around the world use software applications of different complexities and volume of data. We are used to classify the data amount in 3 levels: small data, medium data and big data. Small data is referred to the cases where your data fit easily in one machine and you don't need to share those. This level is not present when we speak about enterprise softwares because the volume is much higher to fit in one machine. Medium data is the set of data that most of developers normally work. It can still fit in 1 machine and probably you are using some kind of legacy RDBMS like MSSQL and you can support hundreds of concurrent users. You can have ACID guaranty and on the same time if you need to scale up that is done only vertically. Nowadays, we speak more and more about big data and how much it is present in every company due to the increased size of data that is being collected and afterwards processed. The big data volume doesn't fit in a single machine, we need generally to retain them for a very long period of time and we need to be able to scale up horizontally to support the dynamic increase of this volume. Based on this situation we need to analyze and find out which database system fits better to our needs, so we raise the following question:

Does MSSQL (RDBMS) or Cassandra(NoSQL) work better for BIG DATA?

We will give answer to this question by taking in consideration some features that are present while speaking about systems that need to deal with Big Data.

1. **High Availability**

   MSSQL is using Master/Slave approach for clustering and has single point of failure. In addition, in case the Master is down we have to activate the slave and for this there are two methods: Manual or Automatic. Both of them need time to be activated and during this period of time (in the best case some seconds) the service is down. The automatic approach that is achieved by using controllers has again point of failure, the controller itself. Who is going to control that this automatic controller is not failing? Imagine when we have driver issues, power failure, change of database settings or even planned OS updates which in big clusters are happening often, in all these cases we will face unavailability which will probably lead to unhappy customers.

   Cassandra was designed to provide high availability and it has no single point of failure because there is no master/slave concept. All the nodes are equal between them and all offer read/write access. This makes possible to scale horizontally which is much times cheaper compared with the vertical scale of MSSQL.

2. **ACID is not true during replication**

   MSSQL and other relational databases are famous for their ACID properties. Lets think about the following situation: The client application is writing into the Master and this server fails. We have to reach the slave server in order to manipulate the data until the master is back and on this moment can happen that which is called replication lag, it means the data are not the same and we lost Consistency. Said that we are not under ACID theory anymore.

   Cassandra is offering a different theory which fits better to big data, it is using AID approach with tunable Consistency that permits you to decide when to tolerate or strict the consistency of your application.

3. **Data normalized or denormalized**

   MSSQL or even more generally in relational databases world we are used to apply third normal form while building our tables. We take a practical example where you need to provide access on the same time to million of users what you will probably face is a very long response time due to the joins between the tables. We try to optimize by using different indexes but after some time we have only one solution to speed up the process, denormalize that table to answer the log-in queries. This is the general approach that we use in relational databases when the table is becoming very big and we need to access it frequently.

   We use denormalized tables since the beginning with Cassandra, so we are just doing the same thing that system which uses MSSQL will do after some time when the data amount grows.

4. **Data sharding**

   We are speaking about huge volumes of data where placing everything in the same machine after a period of time is not possible anymore. This is why we need to do sharding of our data. Now, image that you have MSSQL installed and you need to split the data in four different databases in order to balance the load and you need to run a query which takes all users from a specific state. It means you need to run 4 queries in MSSQL to have the result. Than again we decide to denormalize in order to solve this and make 2 copies of users, 1 by user ID and 1 by State. Suppose another situation in the same example, we need to double the number of nodes it means we need to build a tool to manually split the data from 4 to 8. This is a very big pain, difficult and error proof. Lets don't speak about managing the schema which even more difficult to manage. If you need to change the schema your next step is to propagate this change in all cluster and data centers.

   Cassandra is a distributed database and it offering and managing all the sharding part and needed changes in the best possible way by itself. The developers don't need to think or worry about it because Cassandra will deal with all the complexity of that process.[2]

2 Experiment design

To see the real performance of Cassandra in comparison with Microsoft SQL Server we designed a database and ran number of queries on it. The database represents an online store with a very large number of customers and very large number of requests coming to it. We have modeled it in Cassandra and Microsoft SQL Server to make a performance comparison.

2.1 Database design

We designed our Cassandra database on the data from well-known Northwind database [3]; SQL version uses Northwind directly. The core of the database on which we test our requests is represented with two entities: orders and products in them. This entities are modeled differently in Cassandra and SQL database. The relevant parts of schema for the SQL database are shown in the Listing 7.

```
CREATE TABLE orders (
  id                INT NOT NULL,
  employee_id       INT ,
  customer_id       INT ,
  order_date        DATETIME ,
  shipped_date      DATETIME ,
  ship_name         VARCHAR(50) ,
  ship_address1     VARCHAR(150) ,
  ship_address2     VARCHAR(150) ,
  ship_city         VARCHAR(50) ,
  ship_state        VARCHAR(50) ,
  ship_postal_code VARCHAR(50) ,
  ship_country      VARCHAR(50) ,
  shipping_fee      DECIMAL(19,4) NULL DEFAULT '0.0000',
  payment_type      VARCHAR(50) ,
  paid_date         DATETIME ,
  order_status      VARCHAR(25),
  PRIMARY KEY (id)
);

CREATE TABLE order_details (
  order_id              INT NOT NULL,
  product_id            INT ,
  quantity              DECIMAL(18,4) NOT NULL DEFAULT '0.0000'
     ,
  unit_price            DECIMAL(19,4) NULL DEFAULT '0.0000',
  discount              DOUBLE NOT NULL DEFAULT '0',
  order_detail_status VARCHAR(25),
  date_allocated        DATETIME ,
  PRIMARY KEY (order_id, product_id)
);
```

Listing 7: SQL Tables schema (relevant parts)

Database design in Cassandra starts from identification of queries that the database should efficiently support. We decided that it is important to have fast read access to orders by their ID but also we need to support showing all the orders in some period of time or belonging to particular customers with all the products belonging to the order. The two basic tables for orders and order details are very similar to SQL version, their schema is shown in the Listing 8.

```
CREATE TABLE orders_by_id(
        order_id int PRIMARY KEY,
        customer_id int,
        employee_id int,
        order_date timestamp,
        required_date timestamp,
        shipped_date timestamp,
        ship_via text,
        freight decimal,
        ship_name text,
        ship_address text,
        ship_city text,
        ship_region text,
        ship_postal_code text,
        ship_country text,
) WITH comment= 'Orders␣by␣id'
   AND read_repair_chance = 0.3;

CREATE TABLE order_details(
        order_details_id int,
        product_id int,
        Unit_price decimal,
        Quantity decimal,
        Discount decimal,
        PRIMARY KEY ((order_details_id), product_id)
) WITH comment= 'Order␣Details'
   AND read_repair_chance = 0.3;
```

Listing 8: Cassandra tables schema: base tables

This database already supports efficient querying by order ID but lacks support for the other types of queries we identified as important. To support them we need to create two additional tables shown in the Listing 9. We can see that the tables are denormalized, and *orders_by_customer* contains data for order details. Also, the tables have different partitioning and clustering keys.

```
CREATE TABLE orders_by_time(
        order_id int ,
        customer_id int,
        employee_id int,
        order_date timestamp,
        Order_month timestamp,
        required_date timestamp,
        shipped_date timestamp,
        ship_via text,
        freight decimal,
        ship_name text,
        ship_address text,
        ship_city text,
        ship_region text,
        ship_postal_code text,
        ship_country text,
PRIMARY KEY ((order_month), order_date, order_id)
) WITH comment= 'Orders␣By␣Time'
   AND read_repair_chance = 0.3;

CREATE TABLE orders_by_customer(
        order_id int ,
        customer_id int,
        employee_id int,
        Order_detail map<int, frozen <tuple<int, text,
           decimal, decimal, decimal>>>,
        -- map<Oder_detail_id , tuple<Product_id ,
           Product_name , Unit_price, Quantity, Discount>>
        order_date timestamp,
        required_date timestamp,
        shipped_date timestamp,
        ship_via text,
        freight decimal,
        ship_name text,
        ship_address text,
        ship_city text,
        ship_region text,
        ship_postal_code text,
        ship_country text,
PRIMARY KEY ((customer_id), order_date, order_id)
) WITH comment= 'Orders␣By␣Customer'
   AND read_repair_chance = 0.3;
```

Listing 9: Cassandra tables schema: auxiliary tables

Disclaimer: even though this experiment database design tries to follow real world application design, it is in no way exhaustive. In this example we do not discuss all the possible cases that need to be covered in the real application, such as ensuring consistency between denormalized tables, handling the desired level of read and write consistency, etc.

2.2 Queries

In the experiment we decided to test basic queries such as Insert, Select, Update and Delete in Cassandra and SQL Server. For each desired result we have chosen an optimal for the particular database and schema query, so the queries are not always equivalent but their execution produces the same result. We tested performance of every query with 1000000 rows and for some queries we additionally measured the performance on 10000 and 100000 rows to see the performance dynamics. Every query was performed 6 times with the first round being a warmup and not included in the calculations. The results of the last 5 rounds were averaged. The end result is represented in seconds.

There are two main types of queries modeled in the benchmark. The first type is separate queries on single rows performed in a loop to model large number of clients querying the database. The second type is a single request covering very large number of rows modeled with range or IN. The queries are measured taking into account denormalization, so the request that in SQL updates only one table in Cassandra may update three tables. In SQL all the tables have appropriate indexes. For Cassandra queries we use consistency level QUORUM.

2.3 Database configuration

Used versions:

- Cassandra 3.11.0, CQL spec 3.4.4
- Microsoft SQL Server Enterprise (64-bit) version 13.0.4206.0.

The Cassandra cluster is configured to have 3 node cluster with replication factor of 2 and simple replication strategy; the configuration is given in the Listing 10. The SQL Server cluster is a failover cluster with 3 nodes and without sharding. Configuring sharding on SQL Server cluster may result in higher performance but it was not considered in this research.

```
CREATE KEYSPACE Retail
WITH replication = {'class': 'SimpleStrategy', '
   replication_factor': 2};
```

Listing 10: Cassandra cluster configuration

2.4 Hardware configuration

The experiment was conducted on Cassandra and SQL clusters located in Google Cloud. Cassandra and SQL clusters were configured to have 3 nodes. Each node has the configuration given in the Table 1. The machine issuing requests against the databases is located in the same virtual network as the databases so we consider the network latency to be negligible.

Processor	Intel Xeon Sky Lake, 4 vCPUs 2.0 GHz
RAM	15 Gb
Hard Drive type	HDD

Table 1: Hardware configuration (each node)

3 Experiment results

3.1 Insert

```
insert into order_details (order_details_id, product_id,
   unit_price, quantity, discount)
values (X, X, X, X, X)
```

Listing 11: Inserts in simple table Cassandra

```
insert into order_details (order_details_id, product_id,
   unit_price, quantity, discount)
values (X, X, X, X, X)
```

Listing 12: Inserts in simple table SQL

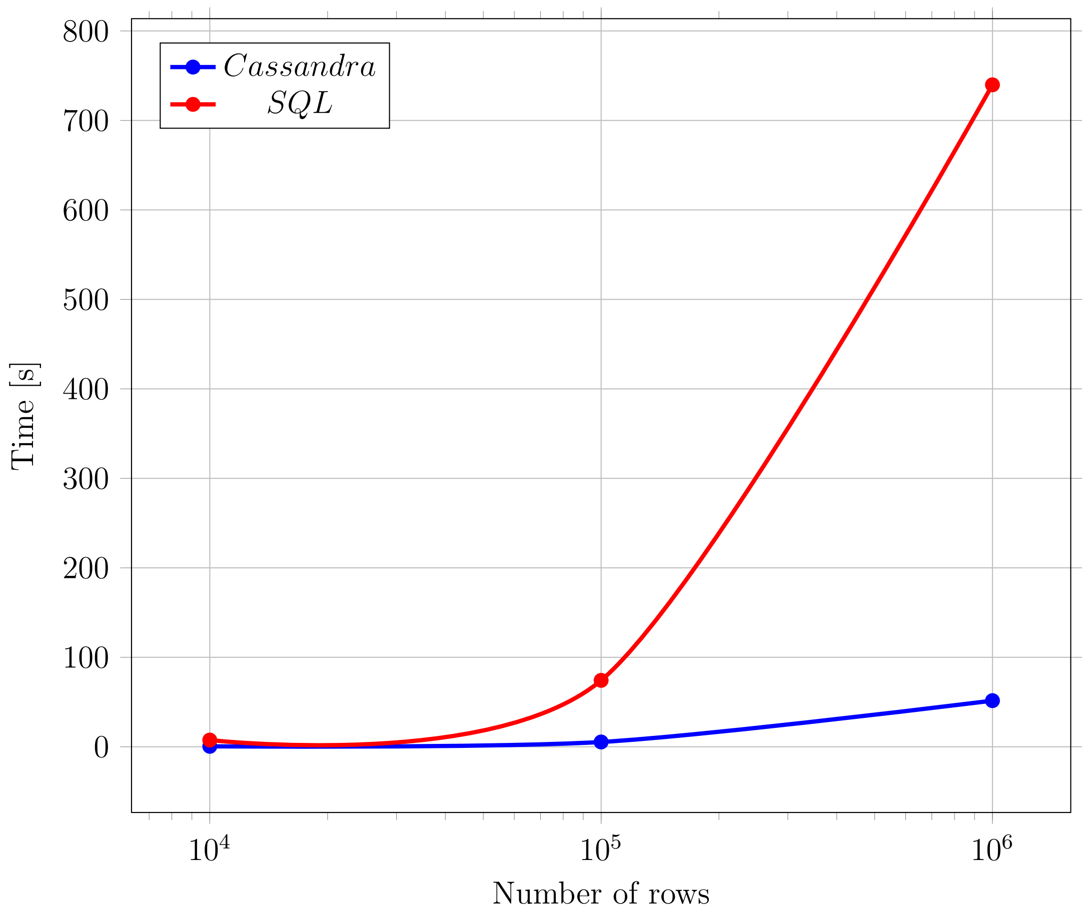

Figure 1: Time taken for single inserts in a loop

This experiment shows the performance when inserting records one by one in a loop. Simple inserts have the same syntax in Cassandra and SQL (listings 11, 12) but their performance is drastically different. We can see from the figure 1 that on 1 million records Cassandra is more than 10x faster.

```
insert into orders_by_id () values (X, X, X, X, X)
insert into orders_by_time () values (X, X, X, X, X)
insert into orders_by_customer () values (X, X, X, X, X)
```

Listing 13: Inserts into multiple tables in Cassandra

```
insert into order_details (order_details_id, product_id,
   unit_price, quantity, discount)
values (X, X, X, X, X)
insert into orders () values (X, X, X, X, X)
```

Listing 14: Inserts into multiple tables in SQL

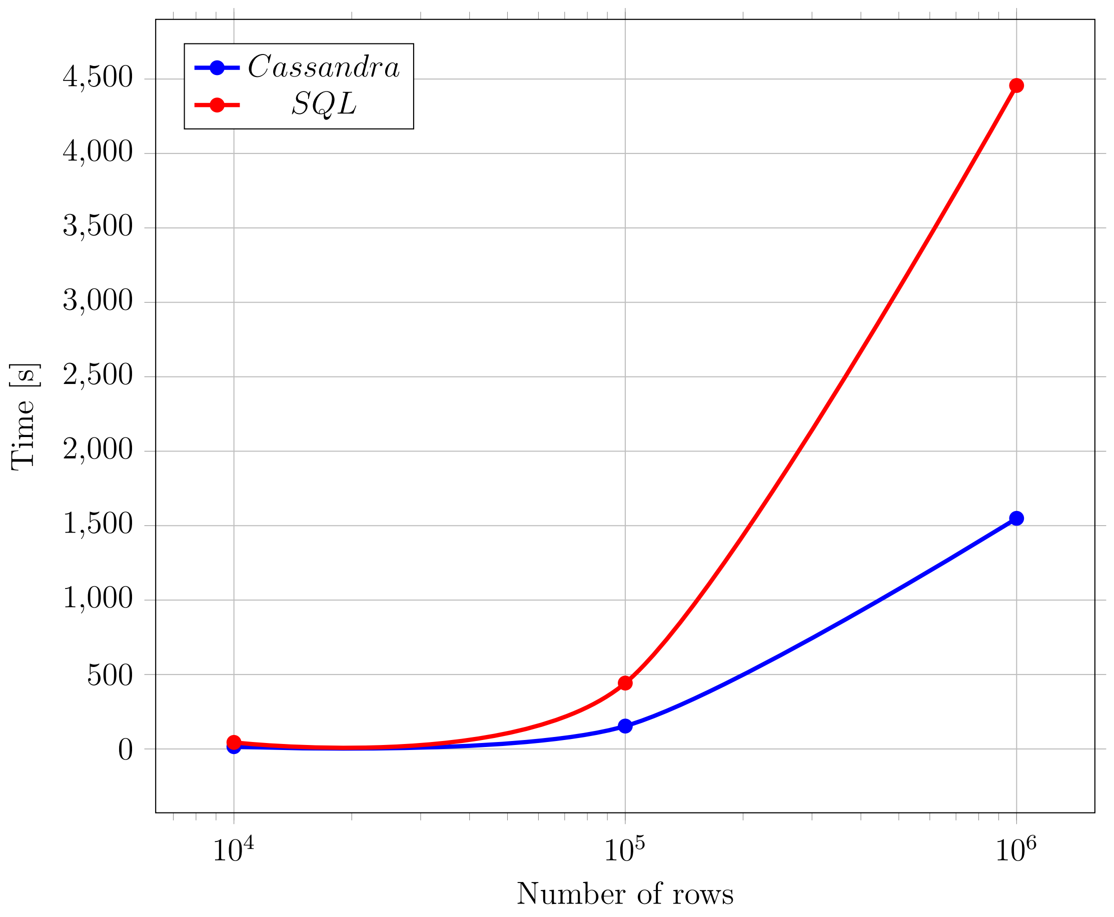

Figure 2: Time taken for inserts into multiple tables in a loop

This experiment shows the performance when inserting records one by one into multiple tables. Since our Cassandra database has denormalized schema with data duplication, every record has to be inserted into multiple tables when in SQL we insert every entity into one respective table. From listings 13, 14 evident that we insert into 3 tables in Cassandra and 2 tables in SQL. Figure 2 shows that in this case Cassandra is 2x faster.

3.2 Select

```
SELECT * FROM order_details
WHERE order_details_id = X
```

Listing 15: Select in loop in Cassandra

```
SELECT * FROM orders
WHERE order_details_id = X
```

Listing 16: Select in loop in SQL

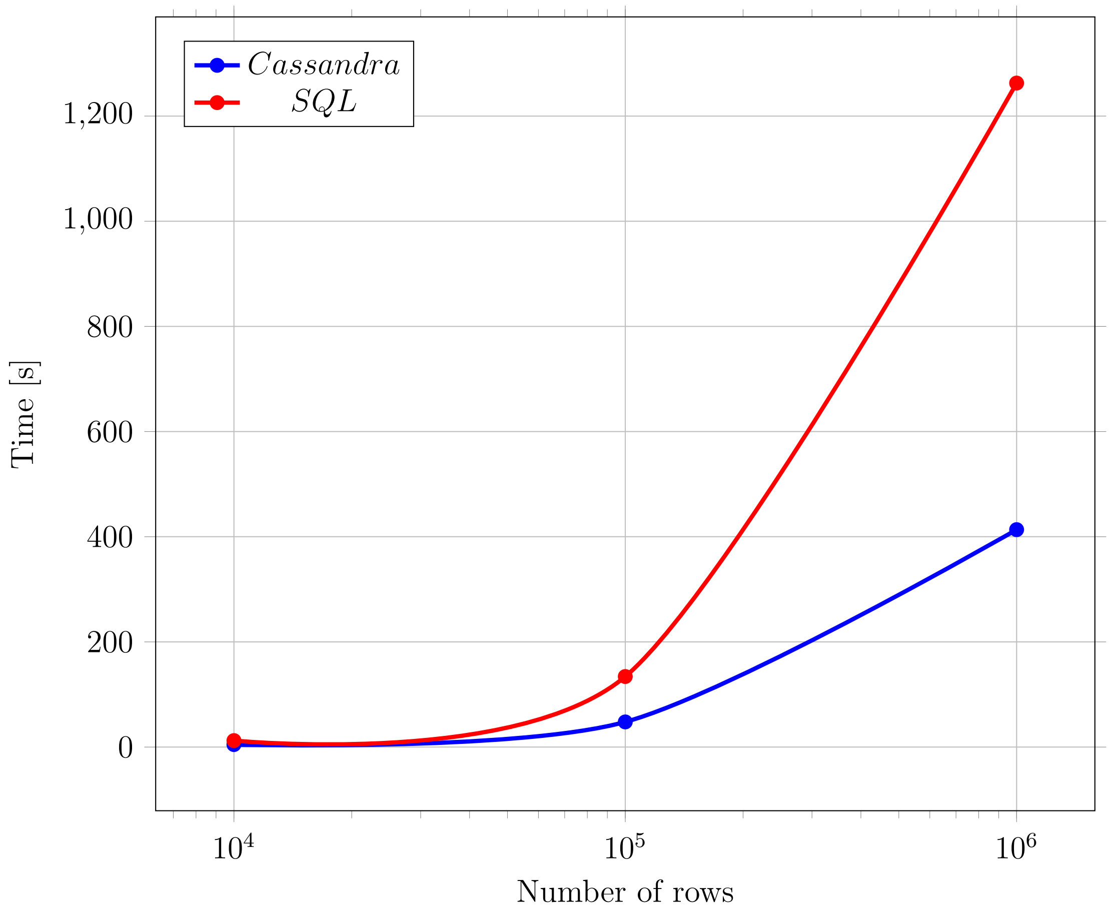

Figure 3: Time taken for point selects in a loop

This experiment shows performance on point selects performed multiple times. Listings 15 and 16 show that the syntax is equivalent. From figure 3 it is clear that on 1 million queries Cassandra is 3x faster.

```
SELECT * FROM orders_by_time
WHERE order_month = '1983-01-01'
```

Listing 17: Select by partition key (1 year) in Cassandra

```
SELECT * FROM orders
WHERE order_date >= '1983-01-01'
AND order_date < '1984-01-01'
```

Listing 18: Select by range of 1 year in SQL

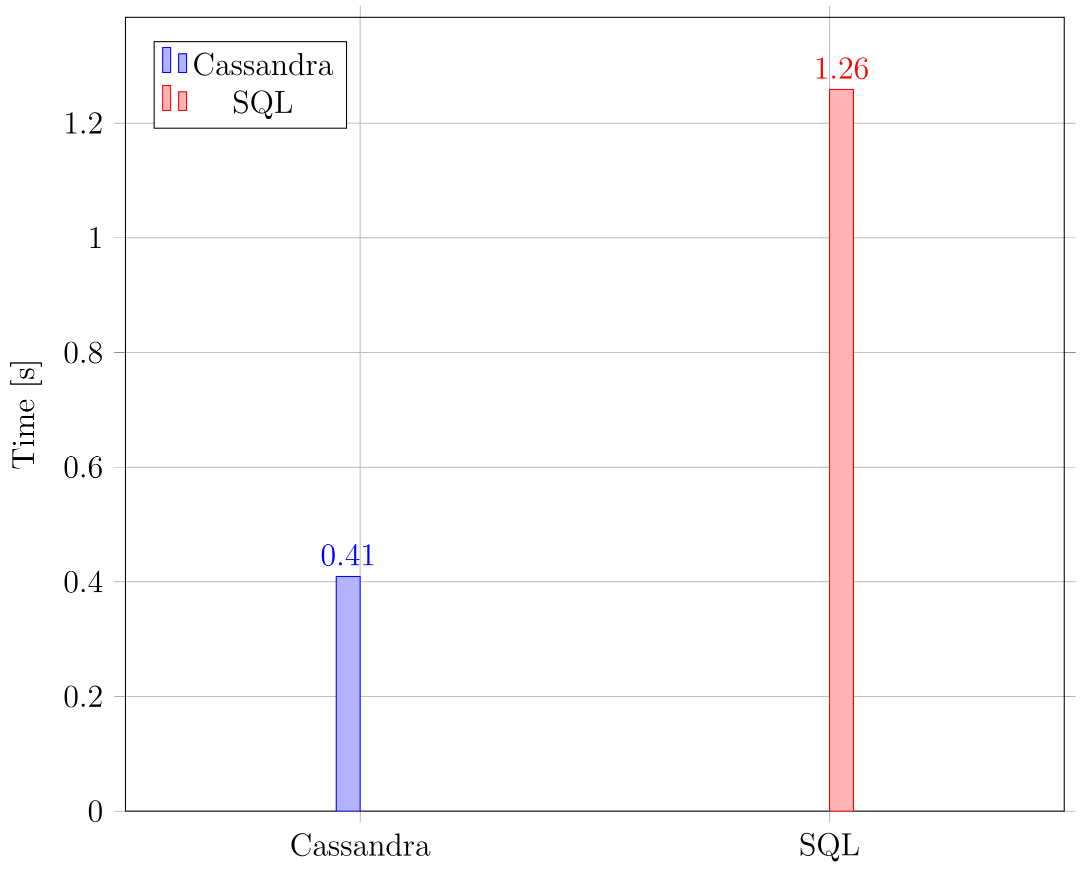

Figure 4: Time taken for select by range of 1 year

This experiment shows the selection of all rows for 1 year. We have different queries (listings 17 and 18) for Cassandra and SQL as we decided to partition this table by year. Figure 4 shows that Cassandra is 3x faster in this case. This case shows that with optimized schema Cassandra can show good performance even on non typical for it scenarios.

```
SELECT * FROM orders_by_time
WHERE order_month >= '1971-01-01'
and order_month <= '2017-12-30'
ALLOW FILTERING
```

Listing 19: Select by range of multiple years in Cassandra

```
SELECT * FROM orders
WHERE order_date >= '1983-01-01'
AND order_date < '2018-01-01'
```

Listing 20: Select by range of multiple years in SQL

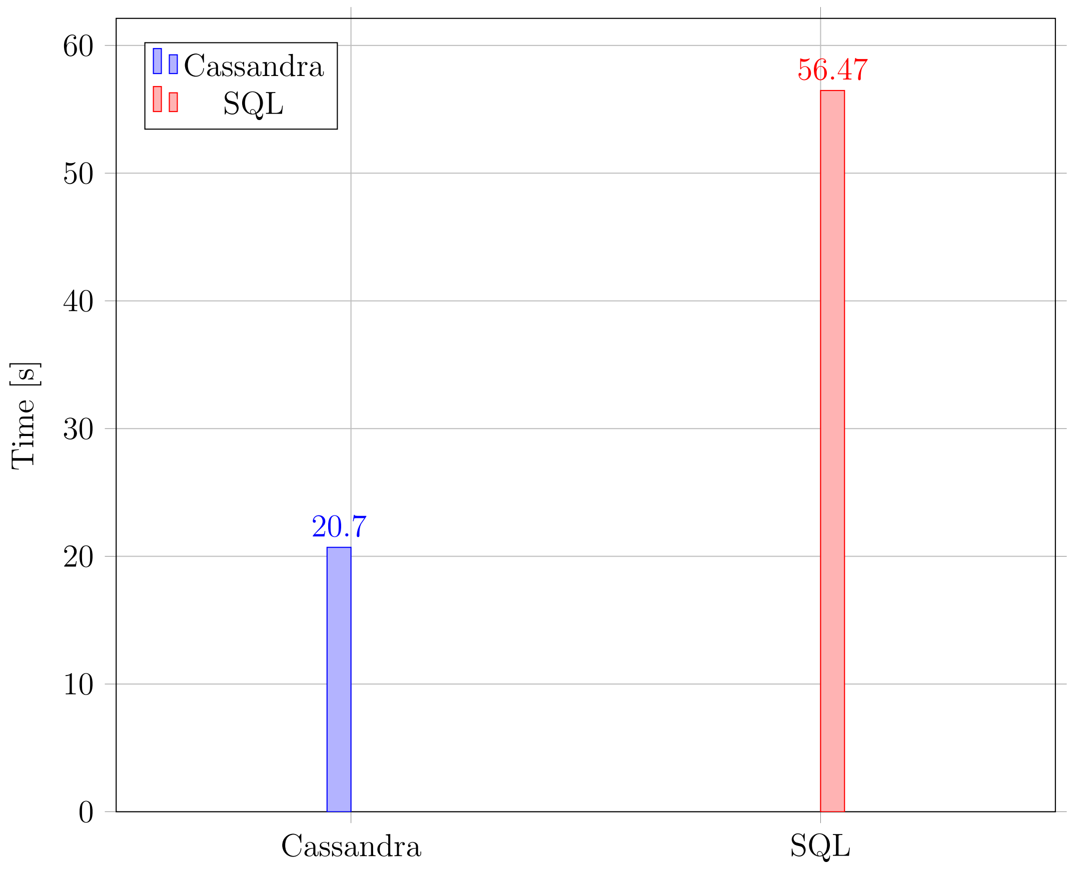

Figure 5: Time taken for select by range of multiple years

This experiment shows selection of data for multiple years. Listings 19 and 20 that the queries are different since we exploit the fact that we partition by year in Cassandra. Figure 5 shows that Cassandra is almost 3x faster.

```
SELECT * FROM orders_by_customer
```

Listing 21: Select from denormalized table in Cassandra

```
SELECT * FROM
[orders] o join [order details] od
on o.orderid  = od.orderid
```

Listing 22: Select from joined tables in SQL

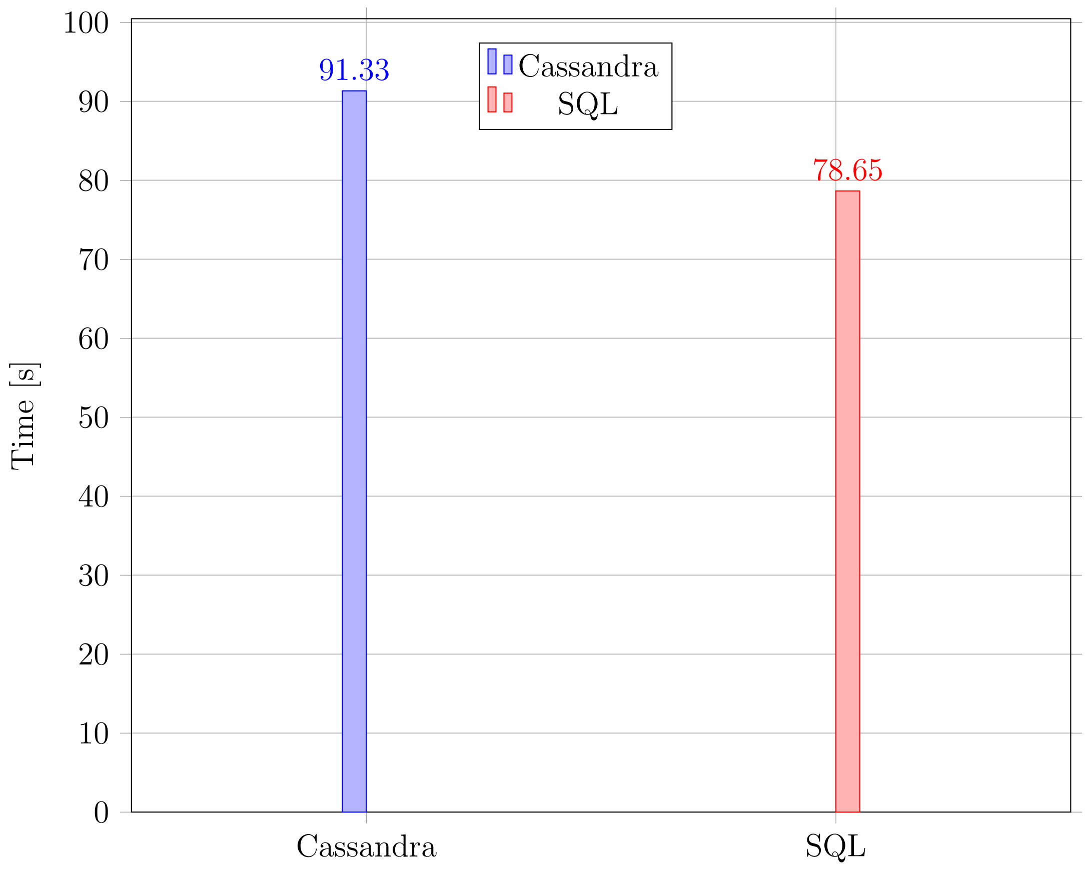

Figure 6: Time of SQL join versus selection from a denormalized table in Cassandra

This experiment shows getting all the orders with their order details. We have order details embedded in orders in Cassandra so we do not have to do join, while in SQL it is necessary; listings 21 and 22 show the difference between the queries. From figure 6 is is clear that Cassandra is slower than SQL even though it does not have to perform the join to produce the result. This case shows one of the weak spots of Cassandra which is unspecific queries hitting all the nodes with large number of results.

3.3 Delete

```
DELETE FROM order_details
WHERE order_details_id = X
```

Listing 23: Delete in loop in Cassandra

```
DELETE FROM orders
WHERE order_details_id = X
```

Listing 24: Delete in loop in SQL

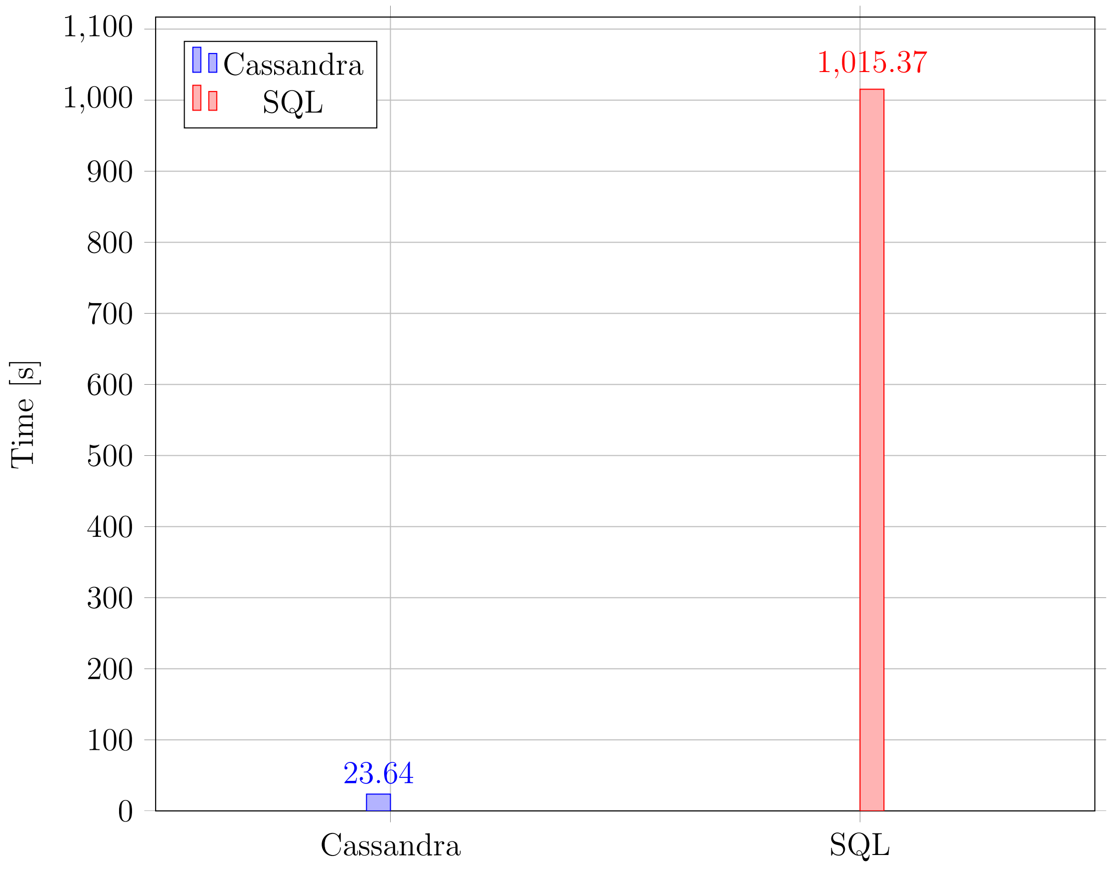

Figure 7: Time taken for point deletes in a loop

This experiment shows point delete queries in a loop on many records. Listings 23 and 24 show that the queries are identical. Figure 7 that Cassandra is 50x faster on 1 million records. Since can easily serve many requests at the same time and it is very efficient on point queries with primary key it is one of the best cases for Cassandra.

```
DELETE FROM order_details
WHERE order_details_id IN (X1, X2, X3, ...)
```

Listing 25: Delete with IN query in Cassandra

```
DELETE FROM [order details]
WHERE OrderID >= 1 AND OrderID <=200000
```

Listing 26: Delete with range query in SQL

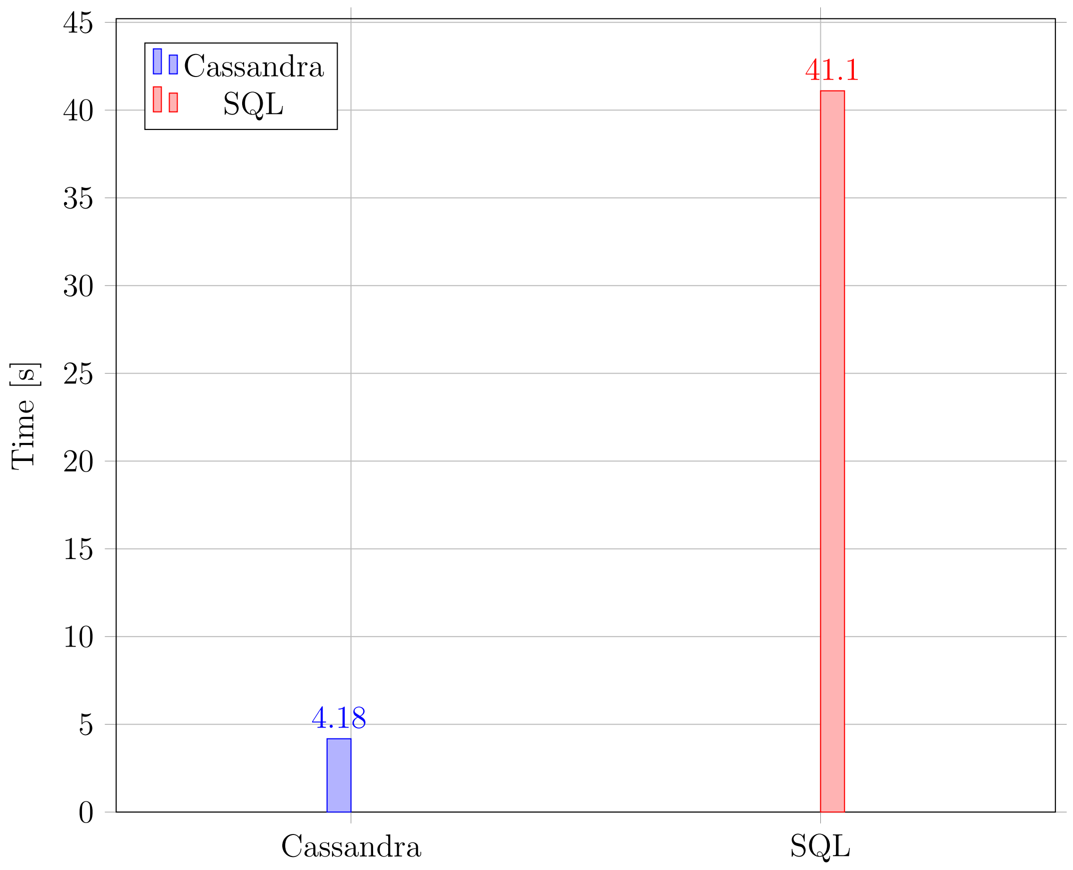

Figure 8: Time taken for mass delete

This experiment shows mass delete. Listings 25 and 26 are not equivalent queries since there is no efficient way to perform such query in Cassandra. Since there is a hidden cost in obtaining IDs for Cassandra which is not present in SQL, we consider this benchmark **non representative** and include it for the sake of completeness.

3.4 Update

```
UPDATE order_details SET quantity = 1
WHERE order_details_id = X
```

Listing 27: Update in loop in Cassandra

```
UPDATE [order details] SET Quantity=1
WHERE OrderID = X
```

Listing 28: Update in loop in SQL

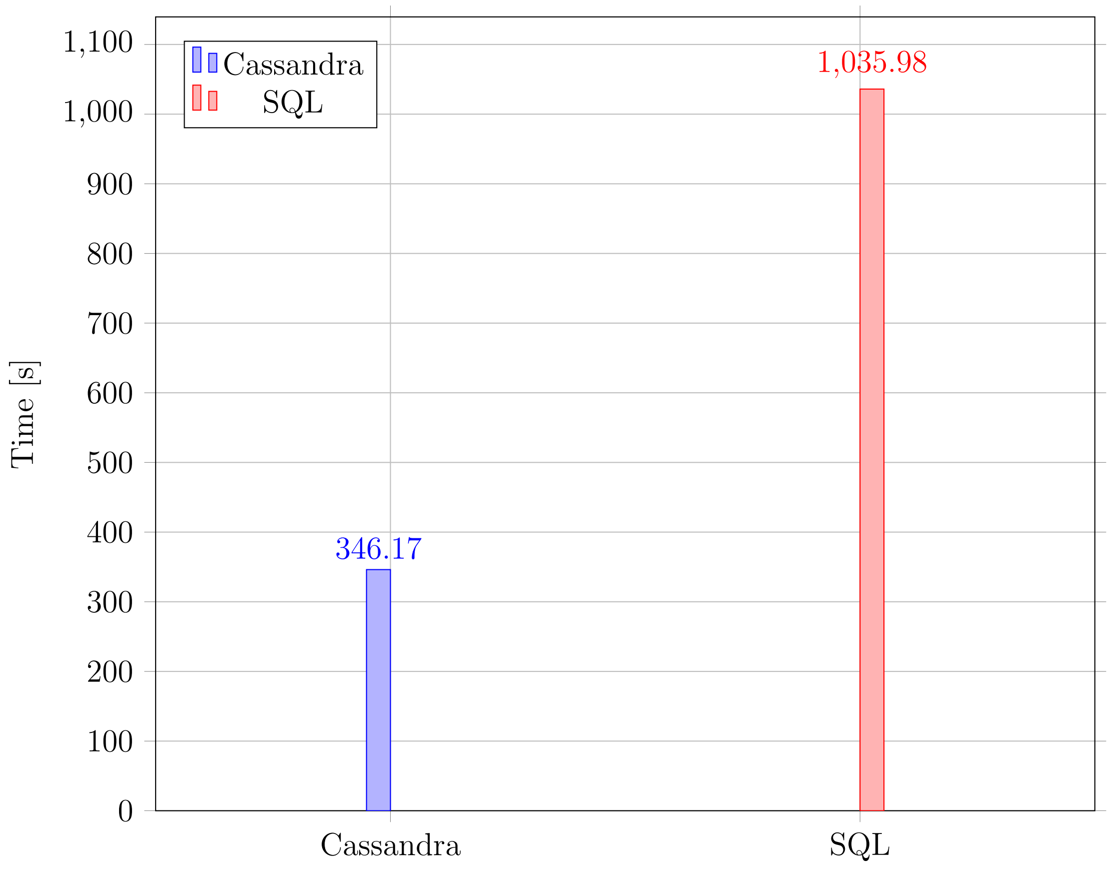

Figure 9: Time taken for point updates in a loop

This experiment shows point updates in a loop for multiple records. Listings 27 and 28 show that queries are equivalent. Figure 9 that Cassandra is 3x faster.

```
UPDATE oders_by_id SET ship_via = 5
WHERE order_id IN (X1, X2, X3, ...)
```

Listing 29: Update with IN query in Cassandra

```
UPDATE orders SET shipvia=5
WHERE OrderID >= 1 AND OrderID <=1000000
```

Listing 30: Update with range query in SQL

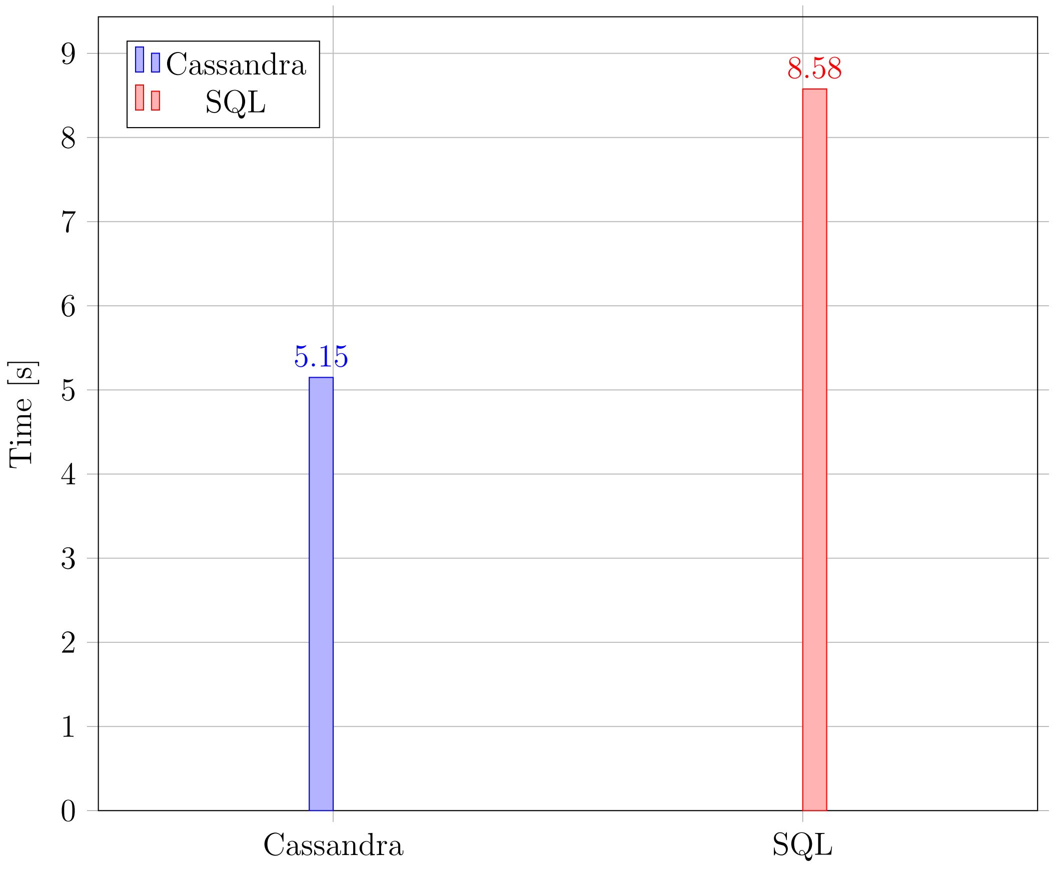

Figure 10: Time taken for mass update

This experiment shows mass update. Listings 29 and 30 are not equivalent queries since there is no efficient way to perform such query in Cassandra. Since there is a hidden cost in obtaining IDs for Cassandra which is not present in SQL, we consider this benchmark **non representative** and include it for the sake of completeness.

Conclusion

The experiment has shown that Cassandra has very good performance in comparison to MS SQL Server. However, this result may only be interpreted as Cassandra being a good choice for a set of applications that can fit its characteristic and limitations. One of the important disadvantages is the absence of joins and limited possibility for ad-hoc queries. The fact that denormalization in most cases is necessary in Cassandra also makes the development and maintenance harder. Another factor that may affect the database choice is the eventual consistency, which may be not acceptable for some applications. SQL Server and other relational databases have many settings by tweaking which one can gain better performance without using a NoSQL solution.

Otherwise, Cassandra is a very good choice for large-scale write-heavy applications. It allows easy and configurable horizontal scalability and high availability that are crucial for many projects in the modern world. Many of the disadvantages of Cassandra can be alleviated by using Spark and Map-Reduce computations.

References

[1] The cassandra query language (cql). [Online]. Available: http://cassandra.apache.org/doc/latest/cql/index.html

[2] D. Academy. (2017) Introduction to apache cassandra. [Online]. Available: https://academy.datastax.com/resources/ds101-introduction-cassandra

[3] Northwind database. [Online]. Available: https://docs.microsoft.com/en-us/dotnet/framework/data/adonet/sql/linq/downloading-sample-databases#downloading-the-northwind-database