Big Data Assignment – Part 2

Contents

[NoSQL & SQL 2](#_Toc25257808)

[Consistency in Databases 2](#_Toc25257809)

[CAP Theorem 2](#_Toc25257810)

[Task 1 3](#_Toc25257811)

[Cassandra 3](#_Toc25257812)

[Neo4j 5](#_Toc25257813)

[Riak 6](#_Toc25257814)

[Task 2 – Cassandra In Depth 7](#_Toc25257815)

[Importing data to Cassandra 8](#_Toc25257816)

[Running Queries 9](#_Toc25257817)

[Query 1: What is the max amount of video game sales? 9](#_Toc25257818)

[Query 2: What are the average sales for all video games in Japan? 9](#_Toc25257819)

[Adding, Removing and Updating data 10](#_Toc25257820)

[Sources: 11](#_Toc25257821)

# NoSQL & SQL

NoSQL databases are databases that came from the concept of SQL/relational databases. The name NoSQL may be a bit misleading, as it actually stands for Not Only Search Query Language. What that means is that it is a non relational database, but it may support SQL-like query languages. By "non relational database" it means that the database does not follow a schema. This typically means that NoSQL databases allow for more flexibility in their data than SQL. Therefore, unstructured data is often used for NoSQL databases.

NoSQL data is also distributed and has horizontal scaling. Because it is a distributed database managment tool, it can handle big data storage. Horizontal scaling (as opposed to vertical scaling) is increasing or decreasing the resource sizes by adding or removing new components, rather than upgrading existing ones. This is also called scaling in or scaling out. A perk of horizontal scaling is that it does not require downtime. For example, if the resources need upgrading, it can be done one by one and thus have the system up and running with only a maximum of 1 resource down at a time. In many instances’ downtime could be a big problem. Big internet shops - such as Amazon make millions, and possibly billions per second. If there is so much as a second of downtime, that would be a massive loss of potential money.

Several different types of NoSQL database systems exist, and each have their own pros and cons. Choosing the right database system for data is important for optimizing the use of the data.

## Consistency in Databases

Consistency in a database context means that only valid data will be written to the database. Writing to a database doesn't mean the transaction is correct - only that it didn't break the database rules.

The type of application determines the consistency priority. For example, a social media post being posted instantly is not as important as an aeroplane getting weather updates instantly.

## CAP Theorem

The CAP Theorem is a concept that a distributed database can only have a maximum of 2 - out of 3 desired features.

**C (Consistency)** - The read queries will always get the correct/latest data for all clients

**A (Availability)** - Database requests always get a response

**P (Partition Tolerance)** - The system will continue working even if any number of messages sent between nodes is lost. This can be e.g. a network failure between two datacenters, where nodes in each datacenter form a partition.

With distributed databases, the partition tolerance (P) in CAP Theorem is used. This leaves the option for 1 more feature in CAP Theorem, either Consistency (C) or Availability (A).

# Task 1

## Cassandra

Cassandra is a column-oriented database. It provides high availability with no single point of failure. That is why Cassandra is often referred to as an AP system (Availability and Partition Tolerant System).

Some of its features are:

* Distributed
  + All the nodes in the cluster have the same roles. Each node contains different data and there is no master. Every node can service any request.
* Supports replication
  + Replication 'settings' are changeable.
* Fault Tolerant
  + The data is automatically replicated to other nodes for fault tolerance. Replication can occur for across different data centers. The nodes that fail can therefore be replaced without downtime.
* Fast linear-scale performance
* Flexible data storage
* Fast writes
* CQL (Cassandra Query Language)

Column-oriented databases uses concepts such as key-space, column family, super column and columns. Key-spaces can be viewed as the equivalent of schema in RDMS. It contains column families and defines how replication shall be handled.

There are two replication strategies in Cassandra: Simple Strategy and Network Topology Strategy. Simple Strategy is used only for a single datacenter and one rack. The Simple Strategy puts a replica on a node determined by the partitioner. If more replicas are to be made, then they are places clockwise in the ring. Otherwise, Network Topology Strategy is used. The strategy defines how many replicas should be made in each datacenter. Network Topology Strategy uses the same method as Simple Strategy (by placing replicas clockwise in the ring) until it stumbles upon a link to another datacenter.

Column family can be viewed as the equivalent to RDMS tables. They consist of multiple rows, where the rows can contain different amounts of columns. The rows can also have different datatypes and column names. Each row must have a unique/primary key. The column is a tuple that contains name, value and timestamp. A super column is a collection of columns that are used often.

The database has a commit log, mem-table and SStable. Every write operation is written to the commit log. It functions as a crash recovery mechanism by going back to previous versions in the commit log. The Mem-table is written to after the commit log. It is a memory cache to store the in-memory copy of the data. Each node has a memtable for each CQL table. The memtable accumulates writes and provides read for data which are not yet stored to disk. The SSTable is the final destination of data. They are the actual files on disk.

Column-oriented databases are good database managment tools if the data has these elements:

* Can tolerate some short-term inconsistency in replicas
* Has dynamic fields
* Has potential for large volumes of data
* Applications that are geographically distributed over multiple data centers.
* Need to operate aggregate function quickly
* Specific read query operations only need very few columns

The dataset i chose was <https://www.kaggle.com/gregorut/videogamesales>. It is a dataset that contains information on video game sales. The reason I'm using Cassandra for this dataset is because Cassandra is a good database managment tool for performing aggregate queries, something that is a positive for analyzing sales information.

Some example queries:

* What is the max amount of video game sales?
* What are the average sales for all video games in Japan?

*\*\* These queries are expanded upon in task 2 of this rapport.*

I would argue that Cassandra is the best database managment system to handle the dataset because it's column-oriented model is best for handling aggregate functions. When for example performing the second example query, the only "column" required is the JP\_Sales column. With Cassandra being column-oriented and not row-oriented, it does not need to read all the other unneccessary columns. It does this by saving all the columns as seperate files, and the way the rows across different files are 'linked' by its offset placement in the file. Therefore, Cassandra only needs to read 1 file, the JP\_Sales file. This makes Cassandra's performance significantly quicker (potentially 100x and 1000x times) than row-oriented database models and perfect for this dataset given the desired data-usage (example queries).

The dataset scrapes for the data using python. The python file is available on their github: <https://github.com/GregorUT/vgchartzScrape/blob/master/vgchartzfull.py>.

Given the queries I have chosen as examples, I would argue that column-oriented database models are the best solutions used in a day-to-day setting, long term or as historical analysis of the dataset. Cassandras effectiveness with aggregate functions are useful across all the different use case settings.

## Neo4j

Neo4j is a graph database. A graph database is a database that uses vertices and edges to represent data. The vertices meaning nodes, and edges meaning relationship. In graph databases, the emphasis is layed on the relationships/edges. How data is connected is considered equally - if not more important than the actual entities themselves. Some examples of this could be if in a social media database, the connections (relationships such as friends with, blocked by, friend of friend of friend) between people (entities) is the important data. Graph databases make queries that would be "joins" in RDMS considerably easier and less complex.

Queries on a graph database does not take longer time the more complex the database becomes. This is because the query does not have to traverse the entire database in order to execute the query. It need work with the data where they are located. Therefore, the amount of time it takes to execute a query only increases when the actual relevant data to the query changes/becomes more complex.

The "schema" in graph databases are extremely flexible and allow for changing and altering of existing vertices and edges. Therefore, when developing a graph database, it is not crucial to plan the structure beforehand, it can easily handle being reworked.

The Dataset I chose was <https://www.kaggle.com/stefanoleone992/filmtv-movies-dataset>. The dataset has information about films (title, year, genre, duration, country, avg vote, votes) and - most interestingly - actors and director. The actor and director "relationships" in neo4j will allow for complex 'join' queries. For this reason, I have chosen Neo4j as the databasemodel, because it puts an emphasis on relations. I would define relationships such as:

|  |  |  |
| --- | --- | --- |
| Actor | acted\_in | Film |
| Actor && Director | worked\_with | Actor && Director |
| Director | directed | Film |

Some example queries:

* What is the shortest connection between two directors?

This is a type of query that really helps graph databases shine and stand out from other database models. In the query language Cypher, there is a shortestpath algorithm that finds the shortest path based on Djikstras algorithm. The syntax of this is as follows (example entities used):

MATCH (d1:Director {name: "Denny Directorsen"}), (d2:Director {name: "Examply Namey"}),

path = shortestpath((d1) - [r] - (d2))

RETURN path

* "Friend of friend" concept, given a specific director, how many directors have worked with the same actor the given director has?

This is a query that emphasises relations, in RDMS this would require many joins. In neo4j the query would look something like this:

MATCH (d1:Director {name: 'Denny Directorsen' })-[:worked\_with]-(a:Actor)-[:worked\_with]-(d2:Director)

RETURN d1, d2

* "Friend of friend of friend" concept, count how many people have worked with people that has worked with people a specific actor has worked with.

MATCH (a1:Actor {name: 'Acty Actorsen' })-[:worked\_with]-()-[:worked\_with]-()-[:worked\_with]-(n)

RETURN count(distinct(n))

As we can see these complex relation-based queries are quite simple and easy in graph databases. If the intention/usage of the dataset is to perform queries such as the example queries I provided, then it is easy to see why neo4j is the better of the four database managment systems. Complex queries based on relationships are easy to implement and avoid having to use joins.

The dataset is imported from the website [www.filmtv.it](http://www.filmtv.it). That site looks like it updates their data frequently, however the dataset on kaggle doesn't keep updating when the site does.

**TODO:: historical analysis**

As stated previously, graph databases handles scaling well. Queries take longer time to perform only when the specific data related to the query becomes more complex and lengthier. That means that it's perfectly fine to store the data used day-to-day and in long-term in the same database, as they won't affect each others’ performances if their data is not entwined.

## Riak

Riak is a key-value database managment system. Key-value databases are NoSQL databases that use a key-value method to store data. This is the simplest way to store data. Key-value database model is very unstructured, and concepts such as foreign keys are not used which means there are no relationships. The key must be unique/primary key. Both the keys and values can be literally anything because they use datatype blob. This means that even image and audio can be stored, aswell as code such as HTML and php.

With the simplicity of the key-value model, as said before it does not permit join queries. It also is only capable of performing queries using the key and cannot query based on the value. This is why you can say the key-value database model is a hash table. In general, key-value database models do not even have a query language. They typically only provide 4 functions:

* Insert(key, value)
* Delete(key)
* Update(key, value)
* Lookup(key)

When choosing a dataset for Riak I wanted to choose a dataset that contained images as blobs to help illustrate a strength with key-value database managment tools. Unfortunately, I could not find a dataset that had this, so instead I opted for <https://www.kaggle.com/gjbroughton/christmas-recipes>.

The christmas recipe dataset is a good option for key-value database model because when working with recipes, one would most likely want to retrieve the whole recipe. My thought process here is to have the dataset be able to return an entire recipe’s information based on the recipe’s name. If the recipe’s name shall be the key content, that means there can’t – or at least shouldn’t be two or more recipe’s sharing the same name.

My example query would then be: Get the entire recipe using its name:

In Riak, the get requests would look something like this using python:

*bucket = client.bucket\_type(‘recipes’).bucket('recipe')*

*obj = bucket.get('recipe-name',)*

*print obj.data*

Bucket here is the ‘keyspace’ in riak. Several buckets can share a ‘bucket type’ which means they would all share the same configuration details. A get() function performed on the bucket requires a key inorder to return the corresponding value.

The dataset is scraped from <https://www.bbcgoodfood.com/>. The update frequency for the dataset is not specified, it seems like it does not update at all.

**TODO:: day-to-day, long-term and historical**

## MongoDB

MongoDB is a document based database magament system. The documents would contain a data structure with name-value pairs. The keys are of type string and the values can be any basic data type aswell as structures such as arrays or objects. In other words, the documents must be JSON-like documents.

Document databases give the flexibility of NoSQL databases, but also provides a more complex data structure than key-value databases.

# Task 2 – Cassandra In Depth

**\*\* The task was completed using the terminal. Despite the task description saying to use Java, Python or Scala the lecturer said using the terminal was ok.**

## Importing data to Cassandra

Firstly, I need to get cassandra up and running:

*sudo service cassandra start*

Then I create the keyspace that shall contain :

*create keyspace videogamesales with replication = {'class':'SimpleStrategy', 'replication\_factor':1};*

Next, I want to create the table/column family for the data. I must navigate to the keyspace that shall contain the table first:

*use videgamesales;*

The create table creates and defines the structure of the table:

*Create table vgsales (pk int, name text, platform text, year int, genre text, publisher text, NA\_Sales float, EU\_Sales float, JP\_Sales float, Other\_Sales float, Global\_Sales float, PRIMARY KEY(pk));*

The copy command:

*Copy vgsales (pk, name, platform, year, genre, publisher, NA\_Sales, EU\_Sales, JP\_Sales, Other\_Sales, Global\_Sales) from '/home/student/Desktop/datafile/shared\_folder\_bigdata/vgsales.csv' with delimiter=',' and header=true;*

This last command (importing the data from csv file into the cassandra column family) resulted in some parsing errors due to the csv file having 'N/A' as value in a column that should be int. The rows that failed were simply skipped during import.

Machine generated alternative text:
Starting copy of 
Faited to import 
'Faited to import 
'Faited to import 
'Faited to import 
videogamesales.vgsales With columns [pk, name, platform, year, genre, 
publisher , 
with base 
with base 
with base 
with base 
na 
lo: 
lo: 
lo: 
lo: 
sales, 
'N/A', 
'N/A', 
'N/A', 
'N/A', 
eu sales, 
given up 
given up 
given up 
given up 
jp_sales, other_sales, 
without retries 
without retries 
without retries 
without retries 
global _ sales] . 
Faited to parse N/A : invalid titerat for 
82 rows: ParseError - 
Faited to parse N/A : invalid titerat for 
69 rows: ParseError - 
Faited to parse N/A : invalid titerat for 
90 rows: ParseError - 
Faited to parse N/A : invalid titerat for 
30 rows: ParseError - 
int() 
int() 
int() 
int() 
'Faited to process 271 rows; fatted rows written to import _ videogamesates_vgsates.err 
processed: 16598 rows; Rate: 
10249 rows/s; Avg. rate: 
9844 rows/s 
16598 rows imported from 1 files in 1.686 seconds (O skipped) . 

A simple select query on the column family will tell us if the import worked as expected (expect to see some data):

Machine generated alternative text:
pk 
4317 
3372 | 
14340 
1584 | 
7034 | 
13909 
9892 | 
16096 | 
9640 
9067 
14059 | 
4830 | 
2731 | 
10113 | 
5056 | 
eu sales 
07 | 
o. 
0.14 
select * from vgsales Itmtt 
15; 
01 
01 
0.42 
0.03 
01 
0.23 
0.33 
0.03 
0.03 
0.17 
0.02 
0.13 
01 
01 
0.23 
0.02 
01 
0.01 
0.02 
01 
0.12 
0.01 
01 
0.01 
0.01 
0.05 
0.13 
0.01 
0.02 
0.09 
0.01 
01 
0.03 
0.37 
0.21 
0.01 
0.47 
0.07 
01 
01 
0.11 
0.01 
0.21 
0.02 
genre 
Adventure I 
Adventure I 
Adventure I 
Shooter I 
Sports I 
Shooter I 
Adventure I 
Action I 
Action I 
Rote -Playing I 
Rote-P1aytng I 
Sports I 
Misc I 
Shooter I 
Sports 
I global _ sales 
I jp_sales 
na 
sales 
O 94 
21 | 
o. 
O .04 
O .04 
name 
Ultimate 
other 
0.45 | 
0.6 | 
1.25 
O .04 
0.14 
O .04 
0.4 | 
o. 75 | 
0.11 | 
0.38 | 
Barbie as The Island Princess I 
Xena: Warrior Princess 
Rampo 
SOCOM: U.S. Navy SEALS confrontation I 
Duck Hunting: Hunting & Retrieving Ducks I 
PO'ed I 
The Walking Dead: Season Two I 
Grand Theft Auto 111 | 
Transformer: Rise of the Dark Spark I 
Record of Agarest War 2 | 
Monster Kingdom: Jewel Summoner I 
NFL Quarterback Club 2000 | 
Rare Replay 
Heavy Fire: Afghanistan I 
NCAA Final Four 2000 | 
sales 
O .04 
platform 
Wii I 
PS I 
SAT I 
PS3 1 
Wii I 
PS I 
x360 1 
PC 
PS3 1 
PS3 1 
psp I 
N64 1 
xone I 
Wii I 
ps I 
publisher 
Activision I 
Electronic Arts I 
Sega 
Sony Computer Entertainment 
Detn8 Games I 
Time Warner Interactive 
O .04 
O .04 
Sony 
Telttate Games I 
Take -Two Interactive I 
Activision I 
Compile Heart I 
Computer Entertainment 
Acctaim Entertainment 
Microsoft Game Studios I 
Mastiff I 
989 sports 
year 
2007 
1998 
1995 
2008 
2007 
1996 
2014 
2002 
2014 
2010 
2006 
1999 
2015 
2011 
1999 

## Running Queries

### Query 1: What is the max amount of video game sales?

The query to answer this question would be like this:

*select max(global\_sales) from vgsales;*

However, for an order by clause to work the column that uses the order by must be indexed. He pk column in this table is the primary key and is automatically indexed. For the global\_sales column to be indexed it must be a secondary index.

*Create index if not exists gs\_i on videogamesales.vgsales(global\_sales);*

Machine generated alternative text:
select from vgsales; 
system. 
82.74 
(1 rows) 

### Query 2: What are the average sales for all video games in Japan?

The query to answer this question would be like this:

*select avg(jp\_sales) from vgsales;*

Machine generated alternative text:
select from vgsales; 
system. 
0.078661 
(1 rows) 

A Warning pops up: Aggregation query used without partition key.

What that means is that the aggregation on the entire table will work on small tables and small clusters but is not a scalable query. When using aggregates, they should be used within a partition.

## Adding, Removing and Updating data

When inserting new data, one can continue importing data from datafiles (as done in earlier) or insert a record manually. Say a new video game sales statistic has popped up that you wish to insert into the database, this can be done as such:

Machine generated alternative text:
e.5,'t 
ttle' 
pk 
insert into videogamesales . vgsales (pk, 
'CD Project Red' , 
2015); 
' Playstation' , 
0.5, 
select * from vgsales where pk=12333; 
eu 
sales, 
genre , 
global _ sales, 
jp_sales, 
na sales, 
publisher 
name 
other sales, 
I year 
platform, 
publisher , 
year) values(12333, 
e .5, 'Shooter', 
2.e, 
e.5, 
I global _ sales I jp_sales I na_sales I 
I eu_sales I genre 
0.5 | Shooter I 
21 
0.5 | 
0.5 | 
name 
titte 
I other_sales 
0.5 | 
platform 
Playstation I 
12333 | 
(1 rows) 
CD Project Red 
| 2015 

Firstly, a record is inserted using the 'insert into' statement. Then, to check that the data was successfully entered a select statement with pk=12333 shows that the statement was successful.

The way in which Cassandra inserts a new record is that it must write each column data to their column file. So, in the above example Cassandra writes to x12 files because there are x12 new column data. When writing this new data, because they are all a connected 'row' the data must be sure to have the same offset in the column files.

To update a record, we can use the record example we created above. Let's say we recieve an update to the sales data, newer numbers have come in. All regional sales have gone from 0.5 to 1. We can change this with the update statement.

Machine generated alternative text:
sales = 1, 
[applied] 
True 
update 
videogamesales . vgsales 
set 
eu 
na 
sales — 
other 
sales 
—1, 
jp_sales = 1, 
global _ sales = 4 where pk=12333 if extsts; 

Here we use the update statement to update specific columns. The where clause defines which rows the update shall perform on. Cassandra tells us that the statement was applied and succeeded. We can still check manually if the data was updated:

Machine generated alternative text:
select * from vgsales where pk=12333; 
pk 
I global _ sales I jp_sales I na_ 
I eu_sales I genre 
sales 
name 
titte I 
other 
sales 
I publisher 
platform 
I year 
12333 | 
(1 rows) 
1 | Shooter I 
11 
Playstation I CD Project Red | 2015 

Cassandra performs an update much like how it performs an insert. But instead of creating new data in the column files, it overwrites the previous data. It uses the offset value to determine the correct data to overwrite.

In this dataset example there is no need to delete a record. I will be demonstrating how to delete a record in Cassandra anyways for the sake of answering the task. The record that will be deleted is the record previously created and updated.

Machine generated alternative text:
delete from videogamesales.vgsales where pk=12333 if extsts; 
[applied] 
True 

Machine generated alternative text:
select * from vgsales where pk=12333; 
pk I eu_sales I genre I global _ sales I jp_sales I 
na_sales I name I other_sales 
(e rows) 
platform I publisher 
year 

To delete data in Cassandra the statement 'delete' is used. One can choose to only delete specific columns or the entire row. In my example I delete the entire row, this is done by not specifying any columns to delete, the default is then to delete all of them. The where clause is used to define what rows the delete statement should affect. Again, Cassandra tells us that the statement was applied succesfully, and we can manually check that the record is gone from the table.

When Cassandra deletes an entire row, it must delete data from all the different column files. In order to make sure it deletes the rows correct columns it makes sure to delete the data with the same offset value.

# Sources:

Consistency & CAP Theorem

<https://en.wikipedia.org/wiki/Consistency_(database_systems)>

<https://hub.packtpub.com/the-cap-theorem-in-practice-the-consistency-vs-availability-trade-off-in-distributed-databases/>

<http://www.warski.org/blog/2011/07/trying-to-understand-cap/>

Cassandra

<https://en.wikipedia.org/wiki/Apache_Cassandra#Main_features>

<https://docs.datastax.com/en/cassandra/3.0/cassandra/architecture/archDataDistributeReplication.html>

<https://medium.com/@marikalam/study-guide-cassandra-data-consistency-496e5bf9cadb>

Neo4j

<https://en.wikipedia.org/wiki/Neo4j>

<https://markhneedham.com/blog/2015/05/19/neo4j-finding-all-shortest-paths/>

Riak

<https://aws.amazon.com/nosql/key-value/>

<https://en.wikipedia.org/wiki/Key-value_database>

<https://docs.riak.com/riak/kv/latest/developing/usage/reading-objects/index.html>

Kaggle datasets:

<https://www.kaggle.com/gregorut/videogamesales>

<https://www.kaggle.com/stefanoleone992/filmtv-movies-dataset>

<https://www.kaggle.com/gjbroughton/christmas-recipes>

**TODO:: MongoDB**

Task 2 - Cassandra In Depth

<https://medium.com/@erbalvindersingh/importing-data-into-cassandra-with-csv-a006ff5f2904>

[Apache Cassandra - Tutorial 12 - CQL - Importing and Exporting Data](https://www.youtube.com/watch?v=N71NwCKfyQ4)

<http://www.batey.info/cassandra-udas.html>

<https://docs.datastax.com/en/cql/3.3/cql/cql_reference/cqlDelete.html>