IN4331 Web Data Management

Development Assignment

June 20, 2017

Authors:
Apourva Parthasarathy (4597583)
Bernie Jollans (4620984)
Jayachithra Kumar (4617312)
Vivek Subramanian (4601211)

1 Introduction

The goal of our project is to re-design an IMDB database using three different database management systems and to distribute it across three nodes. This has been accomplished using a relational database i.e PostgreSQL and two NoSQL database systems which are Cassandra and CouchDB.

The following sections aim to justify the reason for our choice of database management software that we used, why we have modeled our data the way we have, and how we implemented entire system to be capable of handling the Service calls using RESTful web services. The entire system has been designed based on the assumption that writes to the database are non-existent or minimal and read consistency is of paramount importance to this system. A section to separately address the write consistency based on a hypothetical scenario of having writes is also included.

2 Cassandra

2.1 Data Modeling

2.1.1 Basics

Cassandra is a NoSQL database system which is in principle based on Google's BigTable. It is highly scalable and intended for strongly distributed use cases which are in need of high performance.
On a high level, Cassandra's data structure can be thought of as one big table, where every row has the same columns and rows are accessed by their primary key. In order to make data modeling easier, *column families* are employed, which are groups of columns which are similar to an entity in traditional SQL systems. Every row belongs to a single column family and in practice only stores values in its columns.
On a lower level Cassandra's data structure is a nested sorted map, where each row key maps to a map where in turn column names map to values. Keeping this idea in mind it is easy to see that CQL (Cassandra Querying Language) does not support many operations. Cassandra's data model should be optimized for reading rows only by the primary key.
Cassandras distribution works by efficiently partitioning the database into separate rows. For this every row has a *partitioning key*, which is the first column of the columns that make up its primary key. If two rows have a

different partitioning key, they can be stored on different machines. If they however have the same partitioning key, they will be stored on the same machine.

2.1.2 Practical implementation

Since Cassandra is highly scalable, storage space becomes very cheap. On the other hand, since complex queries are not possible (or actually very slow), the tables need to be optimized for reading rows only by the primary key. Additionally, the amount of rows, that are read, with different partitioning keys should be minimized since for every new such read a new machine will have to be accessed.
In order to abide by these rules, we have created a separate column family for every query that we expect. We will have a closer look at these queries in the following.
Replication: Regarding the distribution of the database on multiple nodes, we decided to replicate the full database on all. This avoids a single point of failure and makes load balancing trivial. The only concerns that would be raised are those of consistency. Since our database is by design very low on writes and the writes are not critical, this is not an issue we plan to address. When it comes to Cassandra, this approach of replication has an additional effect on the data modeling. Since Cassandra moves rows to different machines based on their partition key, it is usually important to minimize reads of rows with different partition keys. In our case all rows are replicated on all nodes, therefore no matter how many different partition keys are accessed, only a single machine will have to be queried. This was an important factor in our design decision for SC4 and SC5.

2.1.3 Service Calls

Data Modeling for Reads only

SC1: Detailed movie information
Input: *movie id or title*; **Output:** *List of ⟨Full Movie info⟩*
We decided to create two tables, *tbl_movies_by_word* that has the title as the key and *tbl_movies_by_id* that has the movie id as the key. The reason we have two different tables is that, in Cassandra each table can have only one attribute that acts as the key. For instance, the movie title or the movie id,

but not both in the same table. Since Cassandra can only filter data based on the key, a user could search for detailed movie information on the website using either the movie id or the title. For the *tbl_movies_by_word* table, we have the movie id as the secondary key, as the title might not be unique. On the other hand, the *tbl_movies_by_id* does not require any secondary key.
Cassandra does not support partial key matching, even if the key is a string. In order to work around this we tested two approaches.

1. We create duplicates of a movie's row with a different key. As keys we give every possible combination of adjacent words in the title. For example the movie Alice in Wonderland" will have the following primary keys: "Alice", "Alice in", "Alice in Wonderland", "in", "in Wonderland", "Wonderland".

2. We create duplicates of a movie's row with a different key. This time we only give every single word in the title as a separate key. "Alice in Wonderland" will have the following primary keys: "Alice", "in", "Wonderland". After retrieving the possible titles for the longest word in a given query, we do an online string matching in our web applications code.

The two approaches differ in speed and usage of space. The first approach is faster, since every partial title lookup is still only a single key lookup. The second approach takes about half the space (3.6 million rows vs. 6.6 million rows) of the first approach, but is considerably slower for many queries.

SC2: Detailed actor information
Input: *actorID or firstname and/or lastname*; **Output:** *List of ⟨actor info⟩*
We created two tables for this *tbl_actors_by_id* that has the actorId as the key and *table_actors_by_name* which has the actor's name as the key. Since a user could search for detailed actor info either using the actor id and the first name and/or last name. In the case of *table_actors_by_name* table the first name, last name and both together act as separate keys for a duplicate row and hence we use the actor id as the secondary key. So when the user gives the actor id or the first name and/or last name, he gets the actors details together with a list of all movies that all actors with that name acted in.

SC3: Short actor statistics
Input: *actor id or first name and/or last name*; **Output:** *Full name of the*

actor, number of movies he/she played in
We used the same tables that we created for Detailed actor information i.e the *tbl_actors_by_id* table that has the actor id as key and *table_actors_by_name* that has the first name and last name of the actor as the keys, in addition to having the actor id as the secondary key. We did not have to create separate tables to handle this request, since the inputs were the same and this only required a slightly different query to output the number of movies he/she acted in, instead of the detailed actor info.

SC4: Genre exploration
Input: *Genre, year and optional end year*; **Output:** *List of all movies*
We created a single table *table_genres_by_name_and_year* which only consists of the "genre", "year", "movie id" and "movie title". Using a compound key with the year as the primary and the genre as the secondary key, we list all the movies as output for any given Genre, for a given year/year range.

SC5: Genre statistics
Input: *year and optional end year*; **Output:** *List of genre labels, number of movies on each genre for that year*
The same table that we created for Genre exploration is used here, to provide genre information on the number of movies of a particular genre for a given year or year range. Since we have a fixed set of genres, we query all of them with the year given as secondary filter.

SC4 and SC5 discussion
In the current approach for SC5 we have to execute about 30 queries, since there are about 30 genres in our database. This is a bad thing in theory, since we are accessing rows with 30 different partitioning keys. In the worst case we would be accessing 30 different machines for every execution of SC5. This could have been worked around by creating an extra table where the year acts as primary and the genre as secondary key, instead of the other way around. However in our case we replicate the database completely on all separate nodes. This means that even though there are 30 queries to different partitioning keys, only a single machine will be accessed in any case.

2.2 Write consistency

While choosing this data model for the required service calls, we have assumed that there are no writes to the database. Cassandra has a very good Availability and Partition tolerance. It has a tunable consistency, which lets the consistency levels to be configured to manage availability vs data accuracy. Since we only implement our database across three nodes, consistency would be achieved relatively faster than if we had implemented it over more nodes. One of the most important factors that needs to be taken into consideration is that, unlike in a relational database where the data is normalized, the data in Cassandra would be denormalized and hence each individual table would need to be updated in multiple instances, for the same data. We could try to reduce the number of tables we have, by trying to redesign the database, but there are instances where it cannot be done because, efficiency or query time might be more important than achieving write consistency. Hence a proper design decision is made, to achieve a proper trade-off between design and performance, by taking into consideration whether the write or read consistency is more important.

3 CouchDB

3.1 Why CouchDB?

For the data management system that we are building, tables are assumed to be static with pre-defined set of values that hardly change, and the system is aimed to perform only read operations. Hence a document database should be chosen in such a way that system Availability and Partition Tolerance are given higher priority than Consistency. Figure 1, taken from "CouchDB - A definitive guide" shows the placement of CouchDB with respect to the CAP theorem. By providing Eventual Consistency, where the system incrementally copies document changes between nodes, CouchDB makes it really simple to build applications that sacrifice immediate consistency for huge performance improvements. This together with the benefits of JSON document model and easy-to-use RESTful HTTP API makes CouchDB an apt choice for this application.

3.2 Data Modeling

3.2.1 JSON Document Format

Documents are CouchDB's central data structure. Contents of a database are stored in form of documents rather than tables. On a high-level, documents are self-contained units of data and each document provides a level of abstraction over the data units. This in turn gives some structure and logically groups primitive data types like integers, strings etc.. Internally data is stored in each document in a JSON format where objects are key/value lists.
That being said, modeling the structure of each document in CouchDB is purely application specific and there is no general rule or specific format to structure the data or split the documents.

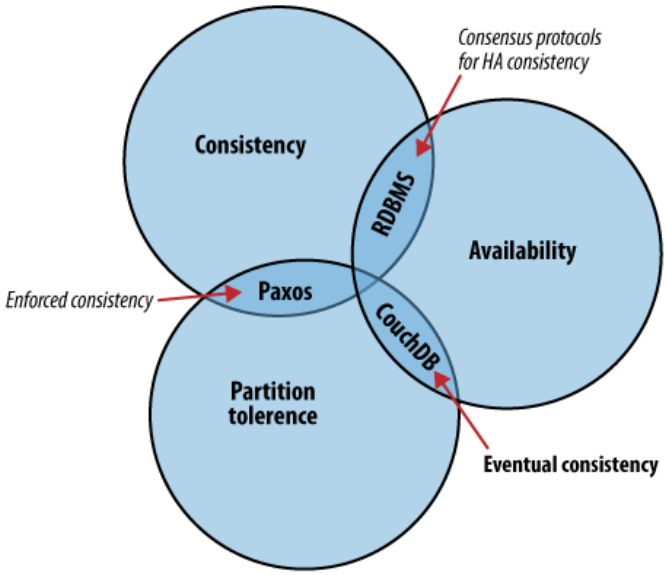

Figure 1: The CAP theorem

3.2.2 Design Decisions

One of the major design challenges that we faced while structuring the data is to decide the extent of denormalization for the relational data. Since duplication of data is not a concern in CouchDB, we had several ways to model data. The first and most obvious way to go about it was to retain the normalization of the relational database thereby providing no place for duplication. However, for this to work out, we should either store the contents of each table in each document or contents of each row in each document. The first case would be inefficient as we would not be able to utilize the benefits of indexing and we would have to read and write the whole document each time a query is made. In the second case, we would be able to make use of indexing but since each row is transformed into a document, the size of the database would be too large and hence execution of queries would take minutes.
Therefore, to overcome these setbacks we need to consider an alternate doc-

ument design which would provide a trade-off between the number of duplicates and execution time. One striking feature about CouchDB is that, unlike RDBMS, in CouchDB a single application could access multiple databases. Keeping that in mind, and assuming that the purpose of our system is defined by five Service Calls (SC1-SC5), we decided to create a database for each SC. In such a setting, each database would only contain documents that are structured to serve its intended SC, thereby reducing the size of each database considerably. Using this method we were able to utilize the benefits of indexing, reduce the load on database and reduce the look up time for data. Furthermore, by including some aggregate fields, we were able to limit the number of databases to three for all the five SCs.

Replication: When it comes to distribution of database, we decided to replicate all the three databases in all the three nodes so that we could avoid having a single point of failure. CouchDB replication synchronizes the copies of the same database allowing users to have low latency access to data no matter where they are. If the local node is down, one of the other two nodes are randomly chosen and queried. A round-robin method is then followed to query all the databases until the information is obtained. Another benefit of replicating data is that load balancing is trivial with replication. We can simply provision as many instances of database and web server as required to optimally handle the load.

Details about how we design the documents for each SC is explained in the next section.

3.2.3 Service Calls

SC1: Detailed Movie Information

Input: *movieid or title*; **Output:** *List of ⟨Full Movie info⟩*

For SC1 we created a single database named "**movies**" with each document holding detailed information about a particular movie (title, movieid, year, type, series, genres, keywords, actors). Here "title" and "type" are **strings**, "movieid" and "year" are **numbers**, "series", "genres", "keywords" and "actors" are stored as **JSON arrays**. For each actor, his/her actual name and the role he/she played were represented in the format "firstname lastname (role)". The actors are sorted according to their billing position in the list. As CouchDB allows multiple indexes, "movieid" and "title" were used as indexes. Apart from these values each JSON document contains a default "_id" which is a UUID generated for each JSON document by default and a

"_rev" or revision that represents an opaque hast value over the document content. These two fields are included by default for all the documents in discussion and we chose to retain them as they are.

SC2: Detailed Actor Information
Input: *actorID or firstname and/or lastname*; **Output:** *List of ⟨actor info⟩*
For SC2, "actors" database was created with each document containing detailed actor information like actor id, first name (fname), last name (lname), middle name (mname), gender, list of movies and number of movies he/she played. Indexes were created on the inputs "actorid", "lname" and "fname".

SC3: Short actor statistics
Input: *actorID or firstname and/or lastname*; **Output:** *Full name of the actor, number of movies he/she played in*
We used the same "actors" database as before with the same set of indexes. Since we have also stored the number of movies for each actor, the retrieval is straightforward.

SC4: Genre exploration
Input: *Genre, year and optional end year*; **Output:** *List of all movies*
For SC4 we created "genres" database. For this database we extracted all the distinct years and all the distinct genres from the relational database and for each combination of year and genre we created a list of movies from the relational database. Hence each document contains "year","genre", "movies" (which is a JSON array) and "number_of_movies" (which is basically a count of movies array). For this table the combination of "year" and "genre" attribute acts as a primary key. Since this table contains very few entries query execution was faster even without any indexes.

SC5: Genre statistics
Input: *year and optional end year*; **Output:** *List of genre labels, number of movies on each genre for that year*
The same "genres" database is used for SC5. The only difference is in the query and the output fields.

3.3 Querying

There are two ways with which we can view data from CouchDb. The first method is to create views with Map-Reduce functions in JavaScript. Creating views are efficient however they are hard to understand and are very time consuming. Hence for our application since we don't require complex queries, we resorted to a second easier option - Mango queries. Mango query was adopted from Cloudant query, which is a declarative style syntax for creating and querying Cloudant indexes. Mango provides a single HTTP API endpoint that accepts JSON bodies via HTTP POST. These bodies provide a set of instructions that will be handled with the results being returned to the client in the same order they were specified. Mango supports four normal CRUD actions - 'insert', 'find', 'update', 'delete' - and one meta action to create indices on the DB.

3.4 Write consistency

CouchDb was chosen under the assumption that there are no writes. However, even if there are writes, CouchDB will still be able to function efficiently by guaranteeing Eventual Consistency. Since we take only three nodes into consideration, consistency will be achieved relatively faster in this case. One main issue that should be considered for updates is that, due to denormalization, we might have to update multiple instances of the same data. This applies for inserting new documents as well, where we might have to insert into more than one database. We could consider reducing the amount of copies by re-designing the documents and normalizing the data. But this might come at the cost of reduced efficiency and increased query time. Hence a proper trade-off needs to be achieved between design and performance, before including writes.

4 Postgres

4.1 Replication for Load Balancing and Availability

Currently, the services only involve read operations which makes load balancing trivial. We can simply provision as many instances of the database and web server as required to handle the load. To ensure availability, each

web service will automatically failover to another server if the original one is down.

Consider the scenario where write operations are to be performed. We can assume that adding or updating movie or actor information occurs quite rarely- in the order of a few thousand movies per year. We can hence chose a fairly simple policy provided by Postgres called Trigger-Based Master-Standby Replication. Here, the write query is sent to the master server which then asynchronously sends the query to stand-by servers.

5 RESTful Web APIs for data access

The data in Postgres, Cassandra and CouchDB databses are exposed through RESTful Web APIs. The services are implemented and hosted on ASP.NET Core which provides a framework for building web applications. The operations performed by each service can be broadly described as

- Parse HTTP web request to identify the parameters requested by the user.
- Query the database and fetch the required data.
- Perform further filtering on fetched data if required.
- Serialize data to the format(JSON or XML) requested and return to client.

6 Conclusion

The report discusses how IMDB data is stored and accessed in NoSQL systems like Cassandra and CouchDB. The data access patterns in NoSQL systems differ widely compared to relational database like Postgres. This report provides a detailed discussion on the advantages and disadvantages of the three databases in the context of the service calls that define the required functionality.