Databases for Developers Exam Project

Authors
Rúni Vedel Niclasen - cph-rn118
Camilla Jenny Valerius Staunstrup - cph-cs340

Link to Source Code Repository

June 2nd 2021 - Software Development - Copenhagen Business Academy
Characer count: 18173

Table of contents

Introduction

For this project we wanted to create a user-driven online forum. As software developer students we spend a lot of time on forums like reddit[1], stackoverflow[2] and hackernews[3] and have wondered how they are built.

Databases

We were assigned to use three databases, so in order to make a semi-realistic use case in combination with a user-driven forum, we had an overall idea of dividing the responsibility as follows:

- A database for caching of the most requested data. The data will be retrievable by keys and does not require relationships.
- A database for most of the user info and especially the user messages/chats. The data will be fast growing and interconnected.
- A database for the forums, posts and comments. The data will be uniform and is easily structured.

We considered several databases for this project and settled on the following three:

- **Redis**
  - For our project, the key strength of Redis is the in-memory data store and its fast lookups. We can utilize this for caching the data that is queried most often and ease the strain on the other databases.
- **Postgresql**
  - We have wanted to dive deeper in our understanding of relational databases in this project. The post and comments combined with the user interaction is a good way to explore more features, such as functions, procedures and triggers.
- **Neo4J**
  - Our initial plan with Neo4J was to use its advanced features such as fraud detection and general efficiency of the database with growing amounts of connected data between users in the form of chat messages and user relationships [4].

Other considerations were HBase and MongoDB. We decided against MongoDB primarily because it hasn't been included in our curriculum for this course. HBase was decided against since the use case in our project would have been the same as Postgres. Due to the size and time constraints of the project we decided that it would be more beneficial to dive deeper in the postgres functionalities rather than adding a third, new database.

System overview

A full overview of our system can be seen in the following diagram. This has been modified slightly during the development and this is the final form.

Full application diagram. A greater resolution image can be seen in our repository [5].

As part of the development of the system we wrote down our product requirements. These can be found in Appendix B.

Product

The backend application is developed in Java using Spring as the framework for the REST API. It uses Maven to organize its packages for the database connectors, Bcrypt (password hashing) and Gson (JSON parsing). All databases were deployed on localhost in Docker containers.

We have strived to implement best practices for our solution, modifying where necessary and discussing best ways to overcome issues.

Postgresql

The Postgres database was implemented as in the application diagram[5:1]. The diagram represents the final version as can also be seen in the following ER diagram produced by the DBeaver client application.

ER diagram showcasing all tables and relationships in Postgres.

When designing our schema we aimed to avoid partial or transitive dependencies, as well as ensuring single valued columns and only storing columns with a common domain in the same table[6].

Responsibility

For this project we decided to put as much responsibility on Postgres as we could. This was done in order to be able to dive deeper into some of the functionalities in a relational database that we would have otherwise let the Java backend handle. This has resulted in a `PostgresAccessor` class in Java that only calls functions and procedures. Below you will find an example of how we call the functions and procedures:

```
public void upvotePost(String postID) {
    Connection conn = getConnection();
    PreparedStatement stmt;
    try {
        stmt = conn.prepareStatement("CALL public.increment_post_karma(?)");
        stmt.setString(1, postID);
        stmt.execute();
    } catch (SQLException ex) {
        ex.printStackTrace();
    }
}
```

Example on a simple procedure call in PostgresAccessor.java. Functions are called by `SELECT * FROM functionName();`

In regards to the stored procedures we are purposefully using a `PreparedStatement` instead of a `CallableStatement`. This is done to avoid the workaround to be able to use `CallableStatement`, as this might be changed in the future[7] [8].

Stored Procedures

We are using the Postgres Stored Procedures for all functionality where we need to set data.
As an example we have a procedure for upvoting a `Comment`:

```
CREATE OR replace PROCEDURE increment_comment_karma(c_id VARCHAR)
LANGUAGE plpgsql
security definer
AS $
    BEGIN
        UPDATE public.postcomment
        SET comment_karma=comment_karma + 1
        WHERE comment_id = c_id;
    END
$;
```

We have implemented the procedures as simply as possible. This results in many specific procedures, instead of a few lengthy ones that are responsible for updating many different tables. As an example we have four procedures to handle upvoting and downvoting `Comments` and `Posts`, one for each use case, instead of one or two larger procedures that would have a more broad responsibility. This way of implementing the procedures will in our experience be less prone to misuse and misunderstanding.

Stored Functions

For all retrieval of data we are using Stored Functions. We have fairly simple functions that just `SELECT` based on an input and a `WHERE` clause, and more complex functions like the following, where we are joining multiple tables and using `GROUP BY` to eliminate redundancy.

```
CREATE OR REPLACE FUNCTION get_FPitem(subID VARCHAR)
  RETURNS TABLE (post_title VARCHAR,
                post_id VARCHAR,
                post_url_identifier VARCHAR,
                post_timestamp TIMESTAMP,
                user_id VARCHAR,
                subreddit_name VARCHAR,
                post_karma INT,
                comments BIGINT) AS
$func$
BEGIN
   RETURN QUERY

        SELECT p.post_title, p.post_id, p.post_url_identifier, p.post_timestamp,
            p.user_id, s.subreddit_name, p.post_karma, COUNT(c.comment_id)
            AS comments
        FROM subreddit s
        LEFT JOIN post p ON s.subreddit_id = p.subreddit_id
        LEFT JOIN postcomment c ON p.post_id = c.post_id
        WHERE s.subreddit_id = subID
        GROUP BY p.post_id, s.subreddit_name;
END
$func$
LANGUAGE plpgsql;
```

Indexing and Queryplan

We looked into query plans in order to get a better understanding of what we should create our indexes on, as well as looking at our stored functions and procedures, specifically the `WHERE` clauses, as these are the fields being queried the most[9].
We generated a plan before and after creating an index based on `WHERE` clauses and could see an improvement in the overall execution time. We planned on creating another set of query plans after the creation of large amounts of synthetic data, but time constraints haven't made it possible for us to do that yet.

Triggers

We created triggers in order to easily keep a users "karma" (sum of upvotes and downvotes on all Posts and Comments) updated. The `TRIGGER` is activated on each `UPDATE` on the "karma" field for the given table and then calls the trigger function it is associated with. In our case the trigger function updates the sum of the users overall karma for either Posts or Comments.

```
CREATE OR REPLACE FUNCTION update_karma_sum_post() RETURNS TRIGGER AS
$BODY$
    BEGIN
        UPDATE reddit_user
        set sum_post_karma=sum_post_karma + (new.post_karma-old.post_karma)
        where user_id=new.user_id;
        RETURN new;
    END;
$BODY$
language plpgsql;

CREATE TRIGGER trigger_post_karma
    AFTER UPDATE OF post_karma ON post
    FOR EACH ROW
    EXECUTE PROCEDURE update_karma_sum_post();
```

Sorting

Simplified our Comments look like this in Postgres:

```
CREATE TABLE postcomment (
  comment_id VARCHAR (50) UNIQUE NOT NULL,
  parent_id VARCHAR (50),
  comment_timestamp TIMESTAMP NOT NULL,
  post_id VARCHAR (50) NOT NULL,
  -- Some fields and foreign keys removed for brevity
  PRIMARY KEY (comment_id),
  FOREIGN KEY (post_id)
    REFERENCES post (post_id) ON DELETE CASCADE,
  FOREIGN KEY (parent_id)
    REFERENCES postcomment (comment_id) ON DELETE CASCADE
);
```

Comments can have a parent (if the comment is at the top level the parent is `NULL`) and in order to achieve the functionality of sites like reddit[1:1] we need to be able to sort through all children based on both "timestamp" and "karma". Comments should be sorted based on "karma" and if any comments on the same level have equal "karma" they should be sorted by time.

Example of reddit comment thread with multiple levels of comments.

In order to achieve this we have implemented a solution[10] by Erwin Brandstetter[11], modified to our needs.

We use the statement `WITH RECURSIVE` to create a flat tree structure from the comments. In the statement there are two `SELECT` statements separated by `UNION ALL`. The first `SELECT` picks out all the top level comments using the `WHERE` clause and sorts them with the help of `row_number()` that assigns an integer value to the resultset based on the clause of `OVER` which in our case is `ORDER BY` "karma" descending and then "timestamp" ascending.
The next `SELECT` finds all lower level child comments in the same manner as described above. The `UNION ALL` combines both result sets into one.

```
CREATE OR REPLACE FUNCTION get_Comments_Sorted(postid VARCHAR)
  RETURNS TABLE (comment_id VARCHAR,
                parent_id VARCHAR,
                comment_timestamp TIMESTAMP,
                comment_content VARCHAR,
                comment_karma INT,
                post_id VARCHAR,
                user_id VARCHAR,
                pppath INT[],
                pppath_sort BIGINT[]) AS
$func$
BEGIN
   RETURN QUERY

        WITH RECURSIVE tree AS (
            SELECT pp.comment_id, pp.parent_id, pp.comment_timestamp, pp.comment_con
                pp.comment_karma, pp.post_id, pp.user_id, ARRAY[pp.comment_karma] AS
                ARRAY[row_number() OVER (ORDER BY pp.comment_karma::int DESC,
                    pp.comment_timestamp::timestamp ASC)] AS path_sort

            FROM   postcomment pp
            WHERE  pp.post_id = postid and pp.parent_id IS NULL

            UNION ALL
            SELECT c.comment_id, c.parent_id, c.comment_timestamp, c.comment_content
                c.comment_karma, c.post_id, c.user_id, t.path || c.comment_karma,
                t.path_sort || row_number() OVER (ORDER BY c.comment_karma::int DESC
                c.comment_timestamp::timestamp ASC)

            FROM   tree t
            JOIN   postcomment c ON t.comment_id = c.parent_id
            )
        SELECT *
        FROM   tree
        ORDER  BY path_sort;
END
$func$
LANGUAGE plpgsql;
```

With this sorting function all the java backend has to do is create a tree with the list of `Comments`. The `Comments` will be in order and just need to be assigned to their parents.

Neo4j

For Neo4j we wanted to explore its inner workings and how its relationships could create interesting queries.
A Docker Compose script was adopted from a classmate and rewritten to fit our system.
A combined DDL and DML script was created and named `neo4j.cypher`. These can be found in the `~/scripts/` directory.

The database was set up with three clusters and two replicas in *READ* mode.

Identifier	Address	Role	ROUTING
core1	localhost:7687	follower	READ, ROUTE
core2	localhost:7688	leader	WRITE, ROUTE
core3	localhost:7689	follower	READ, ROUTE
readreplica1	localhost:7690	read_replica	READ
readreplica2	localhost:7691	read_replica	READ

Generated using `:sysinfo` and `CALL dbms.routing.getRoutingTable({}, "neo4j")`

This setup follows Neo4j's *causal clustering* architecture, ensuring transactional safety, scalability and consistency.[12]
It is important to note that the roles and routing of the cores can change using the default setup of Neo4j, so even though *core2* is marked as **leader** in the above table, it may not always be.[13]

The database scheme was implemented as in the application diagram[5:2]. There were some slight changes throughout the development period, most notably the inclusion of the Chat node.

Database visualization generated using `CALL db.schema.visualization()`

These nodes and their relationships are all generated by `CREATE` or `MERGE` statements, except the `TTL` node, which is part of the Neo4j APOC library[14]. More specifically, it's the *Time to Live* procedure[15] tied to the `ttl` property on `Session`.

Furthermore, we created indexes and constraints on the relevant properties:

```
CREATE CONSTRAINT userName_UQ IF NOT EXISTS
ON (u:User)
ASSERT u.userName IS UNIQUE;

CREATE CONSTRAINT userEmail IF NOT EXISTS
ON (u:User)
ASSERT u.userEmail IS UNIQUE;


CREATE INDEX user_IDX_userName IF NOT EXISTS FOR (u:User) ON (u.userName);
CREATE INDEX user_IDX_userID IF NOT EXISTS FOR (u:User) ON (u.userID);
```

Timestamps were all implemented using Neo4js `localdatetime`.

For accessing the database we used Java driver supplied by Neo4j[16]. For a long time we used the `bolt://` URI scheme for connecting to the database we specified as leader, but due to the cluster changing leaders[13:1], we suddenly couldn't perform write transactions. This was solved by changing the URI scheme first to `bolt+routing://` and finally `neo4j://` which enabled server-side routing using Neo4js own routing tables[17] [18].

The driver was implemented in the `Neo4jAccessor` class, which then provides a method to start a `Session`, on which you run either a read or a write transaction:

```
416  public Chat GetOrCreateChat(String userNameOne, String userNameTwo) {
417  try (Session session = driver.session()) {
418      return session.writeTransaction(tx -> {
419          String query =
420          "MATCH\n" +
421          "(u1:User),\n" +
422          "(u2:User)\n" +
423          "WHERE u1.userName = $userNameOne " +
424          "AND u2.userName = $userNameTwo \n" +
425          "MERGE (u1)-[:PARTICIPATES_IN]->(ch:Chat)<-[:PARTICIPATES_IN]-(u2) \n"
426          "ON CREATE\n" +
427          "SET ch.timestamp = localdatetime(),\n" +
428          "ch.chatID = $chatID \n" +
429          "RETURN ch;";
430          ...
431          var res = tx.run(query, parameters(
432                  "userNameOne", userNameOne,
433                  "userNameTwo", userNameTwo,
434                  "chatID", CreateUUID.getID())).single().get("ch");
435          ...
436      });
437    }
438  }
```

This method either gets an existing chat between two users or creates a new one, using the intricate workings of Neo4j's `MERGE` and it's `ON CREATE` clause. This ensures no duplicate records with precautions[19].

Worth noting here, is that the driver does not support easy mapping between the returned value and the object we wish to represent. This is manually done in line 435 onward, omitted here for brevity.

Redis

The data we needed to cache was data that would be queried often and on reddit[1:2] the most queried data must be their items on a subreddit frontpage, like the one on the image below.

For this we created a class in java to represent this `FPitem` and we need to cache these items per user, per subreddit.
We came up with the following structure for Redis where we have a key-value pair with the `userID+subredditID` as the key and a generated `cacheID` as the value. This value is then used as the key for the next item in Redis, a range of `postID` as values. These `postID` are then each the key for a specific `hset` containing the actual data required to show the `FPitem` and to possible query Postgres for the given post.
The keys in the `hset` are named for easy mapping and the values are the actual data. This cache has an `expire` on all items so they don't stay in the database indefinitely.

The `DataController` queries the `RedisAccessor` for the cache and then checks if the cache has a given length. If it does the cache is returned to the endpoint, if it doesn't the `PostgresAccessor` is queried for the data, the data is then cached in Redis and also returned to the endpoint.

Additionally we also wanted to cache a user's subscriptions to subreddits as this data rarely changes and is shown on each reddit page. This was a much simpler cache since we only need to store a `hset` for each user with the `subredditName` as key and the `subredditID` as value. The key for the `hset` is the `userID+"sub"`.

Combination of databases

The `DataControllerImpl` class is the controller for the databases and all common logic resides here. As an example the cross-database queries are made from this class and the objects are combined here.

Reflection

If this had been a larger project with a more forgiving time constraint, we would have liked to explore more of the advanced functionalities in especially Neo4j, as well as maybe implementing HBase instead of Postgres.

Issues

We have completed a lot of the goals we had for this project, but there are still some to be done. The most pressing issues we haven't had time to address is:

- Large amounts of synthetic data. We haven't had the time for this yet.
- Functionality we would have liked to implement at this stage or in the future, for example:
  - Awarding "coins" to other users and expanding upon the Neo4J relationships in that regard.
  - Caching more data, for example the last ten messages per user.

Neo4j

For Neo4j we wanted to make mapping between the database and Java as easy as possible, especially for making it maintainable. We experimented with a version of Neo4j that is embedded into the Java application[20], unfortunately it was not feasible to implement within the time constraint. The next idea was to implement the Neo4j Java driver, which is used in the final version of the project, but to complement it with the Object-graph mapping (OGM) tool from Neo4j[21]. This tool, however, also proved troublesome, as one of its dependencies would not cooperate.

Postgres

It was a challenge to find a solution to the sorting of `Comments` in Postgres and we ended up spending a considerable amount of time trying out different approaches before we finally found the solution we described in the Sorting section.

REST

We had difficulties adding a REST framework to our project. We suspect it had something to do with the configuration of the framework and how we were exposing the endpoints, since we could deploy the application and see it in the Tomcat manager, but couldn't access the endpoints. The solution we ended up with was using an, to us, entirely new REST framework.

Appendix A: Setup and installation

Repository

https://github.com/Hold-Krykke-BA/DBD/tree/main/ExamSpringApplication

Java

The Java application provides interfaces, controllers and drivers to serve connections to the databases as well as providing a REST API for queries.

The application was developed in IntelliJ IDEA and may respond best to that.

1. Clone the repository
2. Open the `ExamSpringApplication` directory as a project and ensure folders are marked appropriately by your IDE.
3. Build the project using Maven
4. Run the REST API from `~/src/main/java/holdkrykke/rest/ExamSpringAppApplication.java`

The API responds to GET, PUT, POST and DELETE depending on the endpoint.

The accessors in `~/src/main/java/holdkrykke/dataLayer/dataAccessors` also have main methods with manual testing not utilizing the REST api.

Postgresql

To deploy Postgres on localhost simply pull this Docker image:
`docker run -p 5433:5432 -d -e POSTGRES_PASSWORD=softdbd -e POSTGRES_USER=softdbd -e POSTGRES_DB=soft2021 -v pgdata:/var/lib/postgresql/data postgres`

Neo4j

Setting up Neo4j is done using Docker Compose.

1. Open a terminal in `~/scripts/`
2. Run `docker-compose -p dbdexam up`
3. Insert test data using the *neo4j.cypher* script

Monitor your cores and replicas closely until setup is complete, as some complications may occur. This happens when the installation of Neo4j libraries and plugins collides with the Neo4j clustering. To fix an exited node, abort from the setup and start the unresolved cores up using `docker-compose -p dbdexam start`. The cores will connect to the existing cluster automatically. A command for monitoring logs can be found in the cypher script.

Redis

To deploy Redis on localhost simply pull this Docker image:
`docker run --name redistwo -v redis-data:/data -p 6379:6379 -d redis:alpine`.

Appendix B: Product requirements

We sketched out the basic requirements at the project start and have since added a few more. They are divided up in functional and non-functional in the following tables.

Functional requirements

Req. no.	Description	Status
1	See list of posts in given subreddit	Completed
2	Create a user account	Completed
3	See userinfo, karma & followed subreddits	Completed
4	Log in	Completed
5	Create a post in a subreddit	Completed
6	Comment on a post	Completed
7	Comment on a comment	Completed
8	See a specific post and all its comments	Completed
9	Upvote or downvote a post	Completed
10	Upvote or downvote a comment	Completed
11	Join a subreddit	Completed
12	Leave a subreddit	Completed
13	Message another specific user	Completed
14	See message-thread	Completed
15	Add another user to a friends list	
16	Block another user from messages	

Non-functional requirements

Req. no.	Description	Status
1.1	Caching of FPitems per user and subreddit	Completed
3.1	Caching user subscriptions	Completed
8.1	Sorting of comments should be on the database level	Completed
14.1	Ordering of messages should be on database level	Completed

---

1. reddit.com ↩ ↩ ↩
2. stackoverflow.com ↩
3. https://news.ycombinator.com/ ↩
4. https://neo4j.com/use-cases/social-network/ ↩
5. https://raw.githubusercontent.com/Hold-Krykke-BA/DBD/main/ExamSpringApplication/images/project_diagram.png ↩ ↩ ↩
6. https://docs.microsoft.com/en-us/office/troubleshoot/access/database-normalization-description ↩
7. https://www.postgresql.org/message-id/4285.1537201440%40sss.pgh.pa.us ↩
8. https://github.com/pgjdbc/pgjdbc/issues/1413#issuecomment-464851906 ↩
9. https://github.com/Hold-Krykke-BA/DBD/blob/main/ExamSpringApplication/scripts/Queryplan.json ↩
10. https://dba.stackexchange.com/a/171248 ↩
11. https://dba.stackexchange.com/users/3684/erwin-brandstetter ↩
12. https://neo4j.com/docs/operations-manual/current/clustering/introduction/ ↩
13. https://medium.com/neo4j/querying-neo4j-clusters-7d6fde75b5b4 ↩ ↩
14. https://neo4j.com/developer/neo4j-apoc/ ↩
15. https://neo4j.com/labs/apoc/4.1/graph-updates/ttl/ ↩
16. https://neo4j.com/developer/java/ ↩
17. https://neo4j.com/docs/operations-manual/current/clustering/internals/#causal-clustering-routing ↩
18. https://neo4j.com/docs/driver-manual/current/client-applications/#driver-connection-uris ↩
19. https://neo4j.com/developer/kb/understanding-how-merge-works/#_a_merge_without_bound_variables_can_create_duplicate_elements ↩
20. https://neo4j.com/docs/java-reference/current/java-embedded/ ↩
21. https://neo4j.com/docs/ogm-manual/current/reference/#reference ↩