Databases Exam Project SOFT

Introduction

This report was written by Jesper Rusbjerg, and Nikolai Perlt. It is written as a supplement to our first semester exam project in databases, reflecting on the decisions we made throughout the development of this project, and comparing alternative methods of implementations. Our use case is that of a webshop developed to scale and should you wish to try the project yourself, you can follow the installation guide in Appendix 1.

User Stories

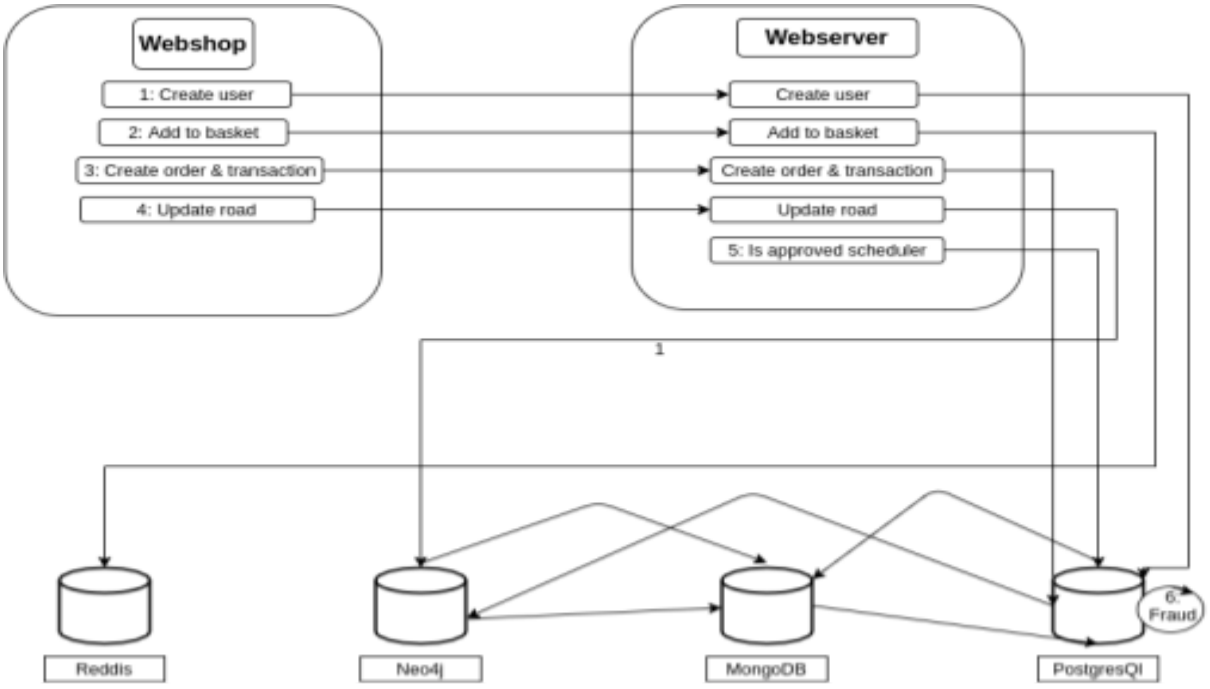
1: As a user, I want to be able to register myself on the site so that I can become a customer and purchase products from the site;
2: As a customer, I want to be able to see random products when I enter the webshop so that I can make an informed decision on what to buy;
3: As a customer, I want to be able to add products to a basket so that I can buy multiple things at once;
4: As a customer, I want to be able to turn my shopping basket into an order and be able to pay for my products with a credit card so that the webshop can deliver my desired products to me;
5: As a delivery man, I want to be able to see all the orders that have been paid for and not yet delivered so that I know which orders to prioritise;
6: As a delivery man, I want to set an order as ‘delivered’ so that we do not deliver it twice;
7: As a delivery man, I want to be able to update a route between two cities so that my colleagues can find the quickest route available;
8: As an owner, I want to be able to add unlimited amounts of products to my random webshop so that my customers can keep searching for the best deals;
9: As a fraud-detection employee, I want to be able to see a list of customers who may potentially commit fraud so that we can ensure our customers safety.

Overview

In this project we created a website that displays random products from different manufacturers. A user should be able to select one or more products, buy them with their credit card and select a city to have the products delivered to. After a purchase, the site stores this data and finds the quickest path to the selected city from our warehouse.

To create this site, we have used four types of databases: Redis (to keep track of a user’s baskets); mongoDB (to store the products that a user can select and the orders that are made by users); PostgresQL (to keep track of user’s credit card information and how much money is spent on each order, as well as detecting possible fraud incidents); and Noe4j (to calculate the shortest path to a city by storing information about all the cities and their roads to the other cities).
In order to give a better understanding of our system architecture, the next chapter will illustrate the technical flow and functionality of our applications.

Flow of program



Program flow diagram

- The above picture represents all the functionality that is available to the site. Operations such as ‘basic get requests’ are not included in the diagram.

- As you enter the webshop page, you will be met by two tables: one representing a shopping basket, and another representing a product catalogue. Each time you update the page, a new product line will be presented for you. The shopping basket remains intact while you refresh the page, as it is being kept in - or “added to” - the external Redis database (as shown in operation 2: “Add to basket”, in the above diagram). You can clear your shopping basket at any time, by pressing the “Clear” button.

- You must be a registered user in order to place an order in this webshop. You can register yourself by pressing the “Register user” button, where you can enter a phone number and a name, which will create your user profile in the PostgresQL database (as shown in operation 1: “Add user”).

- When you are happy with your shopping basket, you can press the “Buy” button. This button will trigger a modal where you must enter a legitimate user profile, combined with a valid credit card and inform us of which city you wish your products to be delivered to. This will then trigger Operation 3: “Create order & transaction”. First, the operation checks if the user is valid; if the user is valid we proceed to the Neo4j

database, where the shortest available path between our company city ("København") and your city of choice is calculated.

- When Neo4js calculation is complete, an order will be created in mongoDB with all of the relevant information: User info, order info and shortest path. We then retrieve the given OrderID from MongoDB and proceed to PostgresQL. PostgresQL will create a new transaction which contains: the credit card information, the total cost of the order and the OrderId received from mongoDB. At this point we consider the user interaction as complete; we now must wait to see if the bank of which the credit card belongs accepts the payment.

- Every 10 minutes, Operation 5: "Is approved scheduler" runs on our web server. Its purpose is to check if any new payments have been confirmed by a bank.

- The second page of our application is meant for our delivery team. On this page you are met by a table that represents orders that have not yet been delivered but have their payment confirmed, which is why the scheduler was an important implementation. If you press on the "Details" button of an order you will be presented with a table showing the shortest path available to your destination.

- As a delivery man, you may pick an order to deliver, but on your way to the destination, you get stuck in a big queue between "København" and "Rødovre". In order to stop your colleagues from entering the same queue you can on the delivery page enter two city names and a new estimate of the time that it takes to drive between the two. This will trigger Operation 4: "Update road"; this operation enters the Neo4j database and updates the given road. Thereafter it goes to mongoDB and updates all of the affected orders, possibly creating a new shorter path for your colleagues.

Strengths of databases

Redis

Redis - first released in 2009 - is an open source database created by Salvatore Sanfilippo . It uses a simple key value design, and can save types such as: string, list and set[1]. Although Redis cannot save objects, almost all objects can be converted to JSON and stored as a string. Redis stands out from other types of databases by primarily saving data in memory rather than on a disk. Redis does have the ability to save its data to disk, but with the intent to recover the data in case of computer shutdown, Redis will as soon as possible read the data into memory again.[2]
This makes reading and writing data much quicker, and makes Redis a prime candidate for use cases such as session store, caching, and other types of temporary data[3]. However, the

[1] https://redis.io/topics/data-types
[2] https://en.wikipedia.org/wiki/Redis
[3] https://aws.amazon.com/redis/

fact that Redis stores data in memory can lead to problems in larger datasets. If the dataset is larger then the servers total memory you either have to release some of the previous data, or reject the new data. One solution to this is partitioning. Partitioning allows us to have more than one instance of Redis running and therefore enables us to store a great deal more data [4].

MongoDB

MongoDB is the most commonly used noSQL databases[5]. It stores its data in a flexible document format, which mimics a JSON structure. MongoDB is open source and used by companies such as Google.[6]

The MongoDB supports master-slave replication[7], in order to ensure consistent data throughout the replica set. The master is the first replica to get updated. Afterwards the master then syncs its slave replicas up to date[8]. Having the data synchronized into multiple replicas in this way creates high data availability, as you are able to balance load between the replicas in the events of high traffic.

If your data is able to fit a denormalized data structure, MongoDB has a very fast query speed, because no joins are needed to retrieve your data. For this reason, it is very important to analyse your data structure needs before choosing MongoDB. If your system does not depend on data consistency then this can be a reason to pick MongoDB as your database[9].

MongoDB supports scaling horizontally and it comes out of the box in many cloud solutions. The main task as a developer is to provide a good division key for the cluster in order to scale each collection. When a collection grows too large, it is sharded into smaller collections, where each collection will contain its own replica sets. A shard manager will keep track of the sharded collections in order to later find a given document[10].

Neo4j

Neo4j is another open source database and is currently the world's most popular graph database[11], used by companies such as Microsoft, IMB and Airbnb[12]. What makes graph

---

[4] https://redis.io/topics/partitioning
[5] https://db-engines.com/en/ranking
[6] https://www.mongodb.com/press/mongodb-powers-modern-application-development-google-cloud-platform?c=082ddb81b2
[7] https://medium.com/@Jelastic/mongodb-replica-set-with-master-slave-replication-and-automated-failover-be3cb374452
[8] https://www.quora.com/How-does-master-slave-architecture-work
[9] http://alronz.github.io/Factors-Influencing-NoSQL-Adoption/site/MongoDB/Results/Strengths%20and%20Weaknesses/
[10] https://dzone.com/articles/divide-and-conquer-high-scalability-with-mongodb-t
[11] https://db-engines.com/en/ranking/graph+dbms (25-05-2020)

databases unique and highly usable is its strength in storing data and the connections created between these data. This model fits very well with the norm of object oriented programming and its relation from one object to another. Many algorithms also use nodes and relations, which is why Neo4j has a lot of ready to use implementations of some of these algorithms. Being able to run these algorithms on a database level has a lot of benefits.

1. We can save network traffic by not having to send data to our client server.
2. We don't have to load all our data into memory, which can be particularly beneficial when dealing with large data.
3. Since Neo4j specializes in graph operations, it can also perform some algorithms, such as shortest path, faster then it can be done in databases such as PostgresQL[13]. However as the paper also concludes it does this by loading more into the memory, which can be an issue in some systems.

PostgresQl

PostgresQL is the fourth most used database in the world[14] and is open sourced and maintained by the PostgreSQL Global Development Group. PostgresQL is a relational database, where some of the most valued principles are to keep consistent data. This data consistency is made possible through numerous principles within the database.

Data Consistency - Transactions

Being able to do everything or nothing is extremely important when you are in need of consistent data[15]. If you are in the middle of a transaction and the connection dies, you want the transaction to rollback to its previous state, and thereby keep the transaction consistent.

Atomic

Being atomic means completing a single or a series of operations at once, or you do none of it; either you do it all, or you do nothing. Each transaction must be atomic to ensure the database always stays consistent.

Consistency

Any data being saved cannot violate any constraint within the database. If a constraint is violated, the transaction must roll back.

Isolation

Each transaction must be isolated from all other transactions running in the database and different levels can be set to ensure different levels of isolations.[16] The more critical your data consistency is, the higher you want your isolation level to be. This can result in slower transactions as they will more frequently have to rollback and try again.

[12] https://neo4j.com/customers/?ref=home
[13] https://bib.irb.hr/datoteka/690975.1268-4089-1-PB.pdf
[14] https://db-engines.com/en/ranking (27-05-2020)
[15] https://www.postgresql.org/docs/8.3/tutorial-transactions.html
[16] https://tapoueh.org/blog/2018/07/postgresql-concurrency-isolation-and-locking/

Durable

Once a commit is made, it is there to stay; the database stays durable and intact even when restarting it.

Data Consistency - Referential integrity

The referential integrity is built on foreign-primary key relations. If Table A has a foreign key field constraint to primary key of Table B, we can be certain that a row with the given ID exists in table B. An example of this could be the connection between a user and a credit card: we want to be sure that a credit card only is created for an existing user[17].

Normalization

If you combine normalization and referential integrity, your database will become easy to keep updated because you only will need to update fields in one place. When a field gets updated it becomes automatically updated in other tables and will be connected through the referential integrity. The basic principles of normalization is that you would like your database to be at least in the 3nf (third normal form)[18].

Databases in relation to user stories

Basket (User story: 3)

In order to keep track of the product that a customer has selected we needed a shopping basket. We needed this process to be quick and snappy so the user did not get frustrated. For this reason we decided to use Redis because of its quick access time and its ability to save it in memory. We also needed to consider the ability to scale our chosen database, due to the fact that we save our user's baskets for up to 72 hours, so that even if they close down their browser they can still access their basket. Redis' ability to use partitions also contributed in it being our chosen database. Another way we could have solved this problem was by saving user's baskets on the client. However we still went with Redis, since this allowed us to have more direct control of our users' data. This could allow us to do analytic work on all the collected data in the future, without the fear of having the user delete their data from their browser.

Create order (User story: 4)

When creating an order we have to save a number of fields. It is not expected that we need to update the order details very often, and all the orders we are creating will fit the same structure. This speaks to the strength of MongoDB because, due to the lack of joins, we will be able to retrieve orders quickly. MongoDB is easy to scale when our order collection grows

[17] https://www.w3resource.com/PostgreSQL/foreign-key-constraint.php
[18] https://www.guru99.com/database-normalization.html

too large for a single container. When such a container scales, it is important that the availability stays high, which MongoDB supports through its replica sets in each sharded part of the container.

When creating an order we also need to store user information and credit card details. When dealing with sensitive data and financial applications, referential integrity is a strong tool as it allows the application to ensure the existence of a correspondent row in each table before creating a transaction. For this very reason we chose to store the financial side of our application in PostgresQL. It is vital that our data consistency is perfect, which also speaks to the strengths of PostgresQL, as normalization and transactions is a great tool for accurate updates.

Cities & Routes (User story: 7)

We wanted to be able to deliver our products in the fastest way possible, and one way to cut down on this metric was to lower the amount of time our drivers spend on the road. To accomplish this we created a graph consisting of nodes in the form of cities and their respective information, and made relations between these in the form of roads. These roads held data that could indicate how much time the road would take to traverse. When we had to progress this graph, the choice was Neo4j for numerous reasons. For example, one such reason was being able to save the graph in its native graph structure. This also allowed us to use some of Neo4j's other algorithms, such as Dijkstra's, which consequently saved a lot of development time and made our application easier to scale, since we didn't have to worry about memory management. This algorithm also helped us because our backend server is developed with Nodejs, which runs on a single thread. By handling Dijkstra's on our data layer we now have the option to use our servers' compute time on other important operations.

Unlimited products (User story: 8)

Each product is highly static and thereby not meant to be changed. All relevant data for a product is stored in a single document. For these reasons we will be doing a great deal of read and writes, and presumably not many updates. With this in mind, one of the core strengths of MongoDB is to quickly read a single document without having to perform any joins. When our product container grows towards infinity we will be able to horizontally scale the container. The key component to doing this scale is a shard key, which should uniformly distribute the collection into smaller shards. When we receive a request for certain products, the database can process the shard key, and thereby know which shard to look to, to find the requested product[19].

Database synchronisation

We have many database synchronizations within our project, whenever you take an action you affect multiple databases at once.

[19] https://dzone.com/articles/divide-and-conquer-high-scalability-with-mongodb-t

Postgres self-sync

Postgres offers a variety of functions to customise your database. Rules, stored procedures and triggers creates a high amount of creativity when designing your PostgresQL database. We are running a webshop, so it was important for us to ensure that we did not send products to fraud accounts. If you change your name multiple times, our administration would need to know about it for this very reason. To ensure this policy, we created a trigger on our accounts table: before every update of the name column we call a trigger that creates a new row in our “namelog” table. If your account appears three times or more in this table, the trigger will mark your account as “suspicious”, meaning you are ineligible to purchase further orders until our administration team have investigated your account.

From order to delivery

When an order gets created, we first create an order containing all of its information in our MongoDB, then a transaction row is created that contains mongoId, amount and user info in our transaction table within our PostgresQL database. After these tasks are completed, the bank must approve the money transaction before a delivery can be made. For this reason a synchronisation was made on a scheduler: every 10 seconds all transactions that have been approved from PostgresQL are collated, then afterwards we can view all orders from MongoDB that have been classified as: ’Delivery: false, Payment: false’. These two results can then be cross-referenced with each other. Any matching ID will mean that the order is now eligible to be delivered, as the bank has approved the payment. This feature was vital for our delivery team; to avoid an overflow of orders and only approved orders will appear on their admin page.

Security

Whenever you deploy any kind of system you have to consider security. This is especially true when you are working with databases. If an attacker gets access to our data, they could not only read the data - and thereby get access to personal/confidential information - they could delete or alter said data. Depending on the criticality of a system, this could have the utmost consequences. Our system handles credit card information and therefore we also have to deeply consider this topic. In order to protect our databases we have considered the following:

1. We have made multiple users with different access to different sections of each database, in order to narrow down the access an attacker could get if one user is compromised.
2. We have configured our firewall to only accept requests from a list of servers. This forces an attacker to go through our main server if they want access to our database giving us the ability to detect and mitigate said attack.

The future of the project

Before 2030, we expect our webshop to grow larger than Amazon. For this reason, it is very important that we find a way to be able to backup our order data and analyse it to find certain consumer patterns. For storing and analysing big amounts of data, HBase would be a great candidate to implement. Hbase is able to scale its rows to unlimited amounts of clusters[20]. You can scale out entire rows, or divide rows into column families and scale out the different families and then combine them again if you need to during a fetch request[21]. Furthermore having your data spread out onto many different clusters is efficient if you want to analyse and MapReduce to find patterns within your data. Each cluster can run a MapReduce on its data in parallel, then when all the clusters are done processing their data, you combine all of the clusters results into one, which is highly effective performance wise[22].

Conclusion

When deciding on a specific database, you have to keep in mind that all databases can solve the majority of problems. However some databases can solve certain problems more efficiently and elegantly than others. Our approach during this project was to analyse each user story and compare its needs to the different strengths of each database. Based on our analysis we then picked the database for each task that we felt would solve the task the most sufficiently. Throughout the whole process we kept in mind our need for scalability because our projection predicts a steep rise in consumers. This was a factor we also kept this in mind when choosing a database for the specific task at hand. Overall we are very happy with the functionality of our project and the integration between our client and databases. Our synchronisations of the different databases will most certainly benefit us as our consumer base rises.

[20] https://blog.eduonix.com/bigdata-and-hadoop/use-hbase-nosql-db/
[21] https://events.static.linuxfound.org/sites/events/files/slides/ApacheBigData2016.pdf
[22] https://subscription.packtpub.com/book/big_data_and_business_intelligence/9781783987245/1/ch01lvl1sec10/use-cases-of-hbase

Appendix

Appendix 1 Installation guide

In order to download and try the project, you first need to clone or download the source code at https://github.com/Perlten/databaseExam. In order to get it working you will need local instances of Redis, PostgresQL and Neo4j. MongoDB already has a cluster deployed in the cloud and therefore there is no further installation process tied with it. PostgresQL does not need pre populated data to get the system to work you do however need to create the different tables, trigger, checks, and stored procedures. This can be done with the sql script backup.sql in the folder postgressBackup/. In order to populate Neo4j you need to run the commands in the file citiesNeo4jRelations.txt found in the folder dataGenScripts/, in order to set up Neo4j configurations run the file neo4jConfig.txt in the same folder. This will create all the cities and their respective relations. In order to start the backend server that handles the management of the databases, run “node index.js”. You can also enter jeperlt-frontend/ en use the command “npm start” to get the backend running.