Databases for developers - Eksamens projekt

- Claus Kramath, cph-ck83@cphbusiness.dk
- Morten Feldt, cph-mf227@cphbusiness.dk
- Mads Wulff, cph-mn492@cphbusiness.dk
- Jörg Oertel, cph-jo130@cphbusiness.dk

Forord

Gennem semestret har vi beskæftiget os med en lang række databaseteknologier, velkendte som nye. Især bruddet med opfattelsen af, at databaser altid følger det relationelle paradigme, har været interessant og vi vil, i dette projekt, få førstehåndserfaring med denne nye tilgang, da projektet omfatter forskellige databaseteknologier.

Repositories (GitHub)

tysker/DBD_Exam (github.com)

Indledning

Vores eksamensprojekt består af en polyglot database applikation, som skal kunne håndtere datastrømme og persistering på forskellige måder. Hver databaseteknologi har sine fordele, og vi vil gennem rapporten beskrive dels applikationen og dens forretningskontekst, samt begrundelser for valg af databaseteknologier i applikationens forskellige områder.

Forretningskontekst

Der skal udvikles en applikation der, på baggrund af aktiers kursudvikling og udvalgte artikler omhandlende disse aktier, skal kunne forudsige en fremtidig kursudvikling med en rimelig sikkerhed, således at investorer med større sandsynlighed kan skabe profit ved handel med værdipapirer.
Applikationens data lagres gennem en backend i forskellige databaser; serveren.

Virkemåde

Applikationen benyttes af en række brugere, som kan knytte en række værdipapirer til sin profil. Til hvert værdipapir kan knyttes en række søgeord, som brugeren finder relevant i forhold til papiret, f.eks. firmaer, teknologier, råstoffer, personer oma. Systemet vil hente

artikler baseret på disse søgeord, analysere budskabet i disse og lade analysens resultat indgå i kursforudsigelsen.

Når en bruger er logget ind, er det meningen at denne skal kunne finde sine fulgte værdipapirer og disses søgeord. Samtidig er det tanken, at systemet vil vise brugeren en graf over den hidtidige og fremtidige kursudvikling.
Når en bruger tilknytter søgeord til sit værdipapir, vil systemet foreslå andre relevante søgeord, baseret på andre brugeres søgeord for det samme værdipapir.
Videre skal det være muligt for brugeren at angive, fra hvilke kilder, artiklerne skal findes.

Krav

Af virkemåden, den episke user story, fremgår en række krav, funktionelle som ikke funktionelle. Det skal her nævnes, at kravfrembringelse er en tilbagevendende aktivitet i mange udviklingsmetoder, hvorfor følgende liste ikke skal ses som udtømmende. Videre har vi ikke kategoriseret alle krav efter eks. FURPS+ eller lign., men blot inddelt dem løseligt.

Funktionelle krav

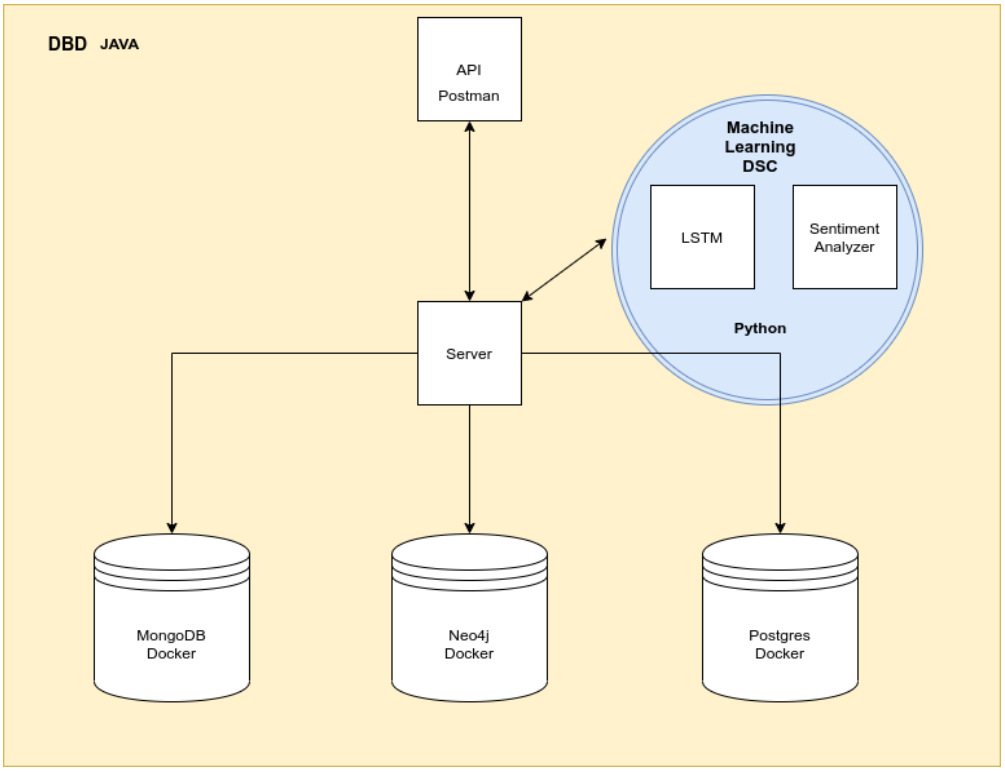
- CRUD af flg. entiteter: Bruger, Aktie, Søgeord, Artikel.
- Forslag til yderligere søgeord ved tilføjelse af søgeord i artikel.
- Tilknytning af aktie til bruger.
- Tilknytning af søgeord til brugers aktie.
- Visning af egen akties tilknyttede søgeord.

Ikke funktionelle krav

- Oprettelse af views til relevante forespørgsler.
- Oprettelse af triggers til sikring af eksempelvis logning.
- Oprettelse af stored procedures til sikring af opdatering af alle relevante tabeller ved CRUD operationer.
- Oprettelse af roller til forhindring af utilsigtet adgang til funktionalitet i databasen.
- Atomare ændringer på tværs af databaser.
- Korte svartider aht. aktiehandlens natur.

Afgrænsning

Som det fremgår af virkemåden, omfatter systemet, udover databaser til persistering, også elementer af machine learning og artificial intelligence. Disse elementer er ikke en del af dette database projekt, hvorfor de ikke diskuteres yderligere.

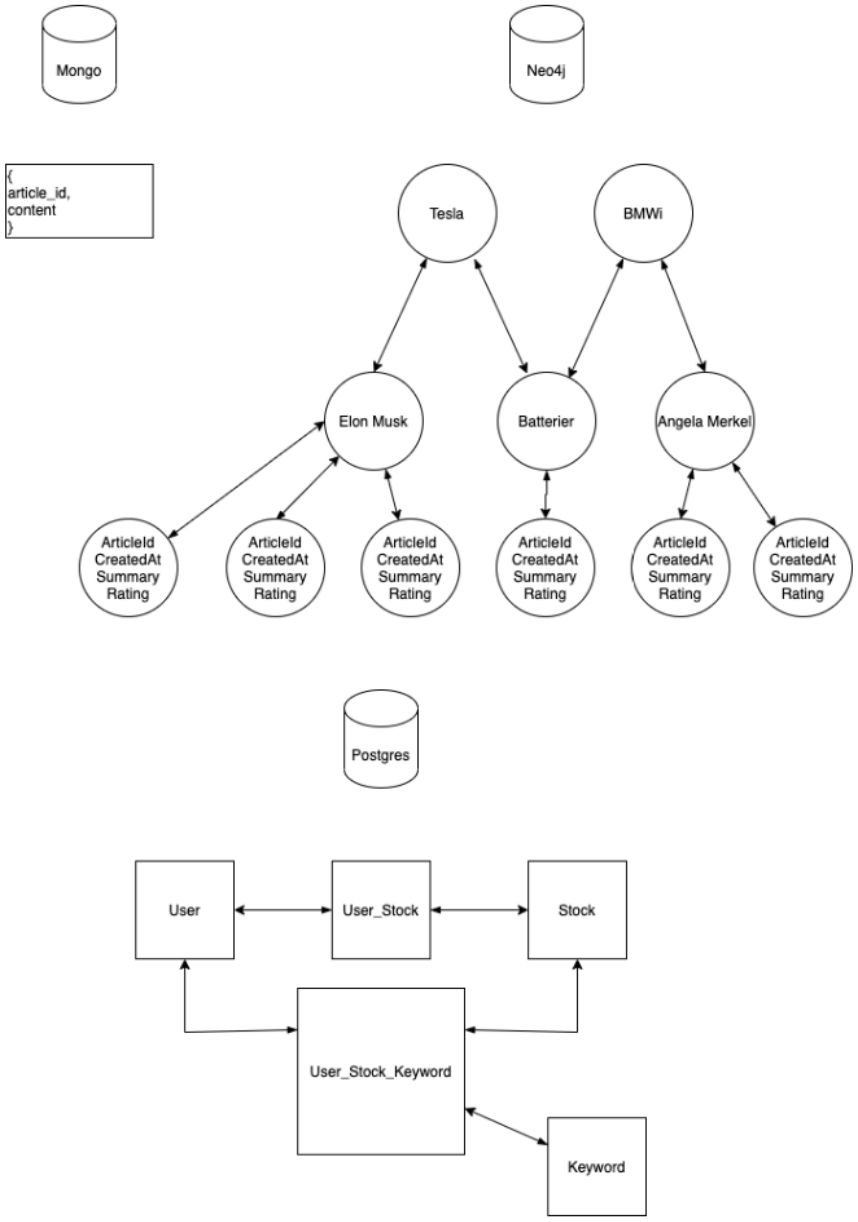


Indhentning af artikler fra forskellige kilder vha. eksempelvis scrapere, er ikke implementeret, da vi mener, at det flytter fokus fra det centrale emne; databaser. Der er dog taget højde for denne funktionalitet i afsnittet 'Datagrundlag og valg af teknologi' hvor vi omtaler mongodb. Ligeledes har vi, af tidsmæssige årsager, ikke implementeret databaser i clusters, men nævner i samme afsnit, i omtalen af neo4j, nogle overvejelser man kunne gøre sig, for at bringe dette aspekt ind i systemet.

Datagrundlag og valg af teknologi

For at tilvejebringe kandidater til tabeller i databasen, har vi gennem studiet benyttet lingvistisk analyse som, relativt enkelt, identificerer kandidatklasser og brugsmønstre. Blandt disse er der nogle, som man ønsker at gemme information om, over tid. Disse er ofte blevet til tabeller i en relationel database.
Af vores episke user story, nævnt i afsnittet 'Virkemåde', har vi således udledt flg. kandidater til vores databaser:
Bruger, søgeord, aktie, artikel.

Da handel med værdipapirer kan foregå i meget hurtige tidsintervaller, er systemets svartider af stor betydning. Denne betragtning er også medvirkende til valg af database teknologier. Ligeledes forestiller vi os, at mængden af data, alene på artikelsiden, hurtigt vil give udfordringer, hvis ikke en skalerbar databaseteknologi vælges her.



Ovenfor er afbilledet en tidlig, konceptuel fremstilling af vores database struktur.

PostgreSQL - relationel database teknologi

Vi har valgt at lagre brugere, søgeord og aktier i denne database. Valget faldt på en relationel database teknologi af flere årsager:

- Strukturen på disse data er kendt på forhånd og stabil.
- Det er ikke her, vi vil se de hurtigt ankommende og voksende datamængder allerførst, således skønnes behovet for skalering og høj hastighed ikke stort her. Nogle argumenterer endda for, at en langsommelig indlogning kan være med til at modvirke forsøg på hacking af brugerkonti, se f.eks. bcrypt[1].
- Der er scenarier, vi kan sikre, ved at udnytte nogle af den relationelle databases funktioner - f.eks. at et søgeord, ved tilføjelse, både skal oprettes i egen tabel og i join-tabellen.

[1] https://auth0.com/blog/hashing-in-action-understanding-bcrypt/

Stored procedures og functions

I forbindelse med en forespørgsel, kan man have behov for at bruge nogle ekstra variabler. Man kan også have en kompleks forespørgsel, som f.eks. indsætter data i flere tabeller. Her kan en stored procedure eller function være en løsning, da den kan tage variabler med ind og derudover også returnere, som en metode i et programmeringssprog.

Nedenfor ses en af vores functions; apply_keyword_stock. Her sikrer vi, at når der kommer et søgeord (keyword), så tjekkes der automatisk for om det eksisterer i forvejen. Såfremt det eksisterer, hentes dets id, ellers oprettes det i keyword-tabellen, hvorefter id'et hentes.
Til slut tilknyttes id'et en aktie og en bruger vha. proceduren add_user_keyword_stock.
Derudover har vi også brug for at finde en aktie ud fra et navn, og her har vi samme fremgangsmåde som med søgeordet.

```
CREATE OR REPLACE FUNCTION apply_keyword_stock(_userId int, _keywordName varchar, _stockName varchar)
RETURNS boolean AS $$
DECLARE
    did_insert boolean := false;
    keywordId integer;
    userId integer :=_userId;
    stockId integer;
BEGIN
    SELECT id INTO keywordId
    FROM keywords k
    WHERE k.keyword = _keywordName
    LIMIT 1;

    IF keywordId IS NULL THEN
        INSERT INTO keywords (keyword)
        VALUES (_keywordName)
        RETURNING id INTO keywordId;

        did_insert := true;
    END IF;

SELECT id INTO stockId
    FROM stocks s
    WHERE s.stockname = _stockName
    LIMIT 1;

    IF stockId IS NULL THEN
        INSERT INTO stocks (stockname)
        VALUES (_stockName)
        RETURNING id INTO stockId;

        did_insert := true;
    END IF;

    CALL add_user_keyword_stock(userId, keywordId, stockId);

    RETURN did_insert;
END;
$$ LANGUAGE plpgsql;
```

Views

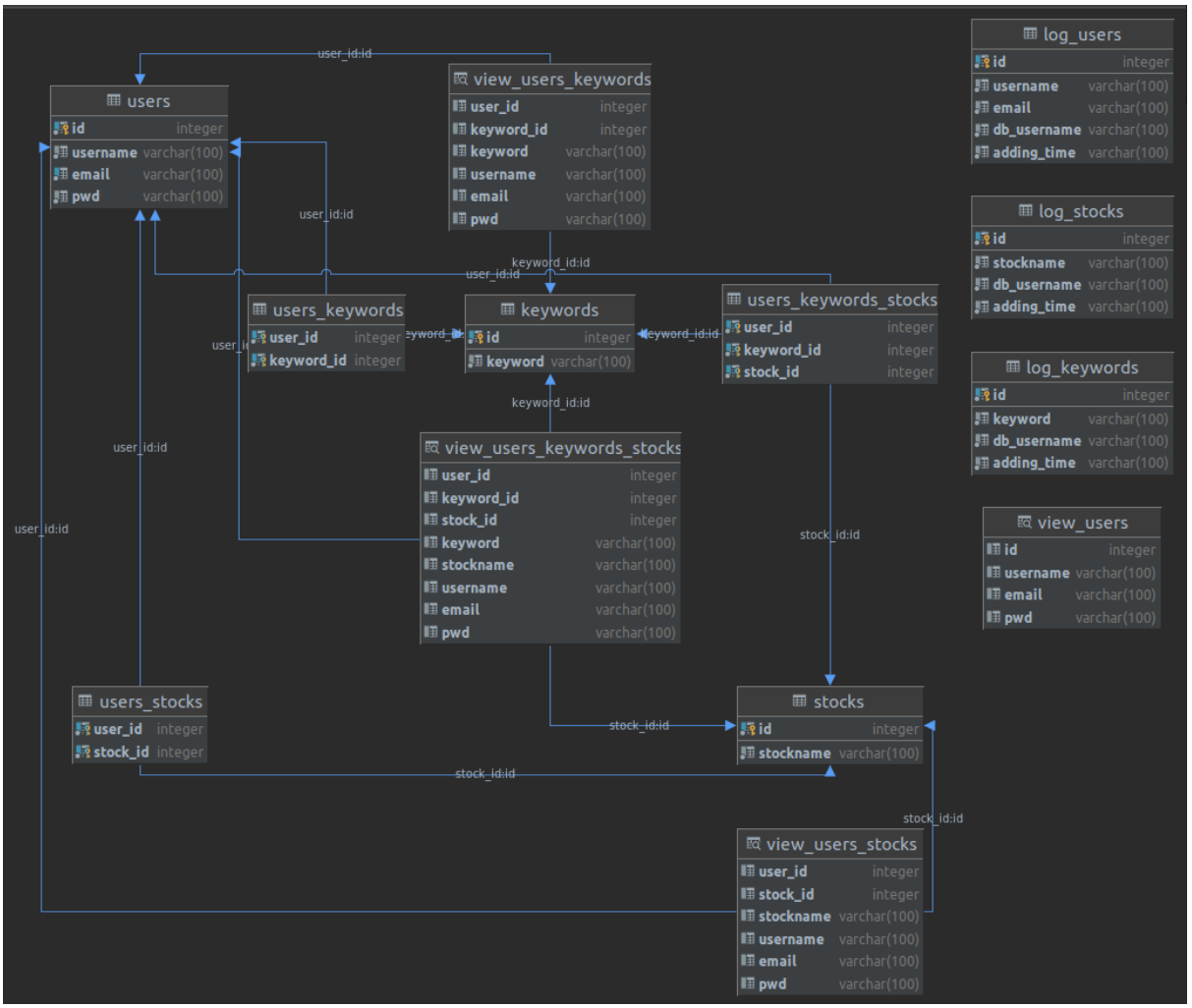
Et view kan være med til at gøre en kompleks SQL-query nemmere at køre, da man i stedet for selve den 'rå' SQL-query kun skal køre en SELECT-statement på selve view'et.
Vi kunne i forbindelse med vores database opsætning se, at vi ville komme til at have nogle komplekse SQL-queries, og derfor valgte vi at implementere views. Nedenfor kan ses et eksempel på en af vores views, og her kan man se forskellen i at lave SELECT på view'et fremfor en flere linier lang SQL-query.

```
CREATE OR REPLACE VIEW view_users_keywords_stocks AS
SELECT user_id, keyword_id, stock_id, keyword, stockname, username, email, pwd
FROM users_keywords_stocks
join keywords on users_keywords_stocks.keyword_id = keywords.id
join stocks on users_keywords_stocks.stock_id = stocks.id
join users on users_keywords_stocks.user_id = users.id;
```

Triggers

Såfremt man har noget, som man ønsker sker f.eks. ved en INSERT-statement i en tabel, kan man bruge en trigger. Vi brugte triggers til at lave en form for logging, sådan at hver gang der f.eks. blev oprettet en bruger, så blev bruger oplysningerne også automatisk gemt i en tabel sammen med oplysninger om hvilke database-bruger samt hvornår.

ER-diagram



Ovenstående ER-diagram viser sammenhængen og relationerne mellem tabellerne i vores relationelle database. Vi har 3 tabeller til data:

- 'users' til brugere
- 'keywords' til søgeord
- 'stocks' til aktier

Herudover har vi tre join tabeller:

- 'users_keywords' som kobler en bruger sammen med et søgeord.
- 'users_stocks' som kobler en bruger sammen med en aktie.
- 'users_keywords_stocks' som samler informationer om bruger, søgeord og aktie. Denne tabel er lavet med det formål, at kunne vise hvilke søgeord, en given bruger har for en given aktie.

Herudover har vi tre log-tabeller, som bruges i vores triggers når nye værdier indsættes.

MongoDB - document store

I modsætning til den relationelle database, må vi forventeligt have en hurtig og skalerbar database til håndtering af artikler, idet disse kan komme fra mange kilder, samtidigt og være af variabel størrelse.
Da vi således ikke kan være sikre på, hvilke data vi skal lagre ifm. artikler, skønnes det at være bedre, med en form for 'document store', hvor artikler kan lagres i sin rå form. Heraf kan læses, at vi benytter os af ELT; extract-load-transform, således at vi blot kan scrape og gemme data, for senere at behandle det.

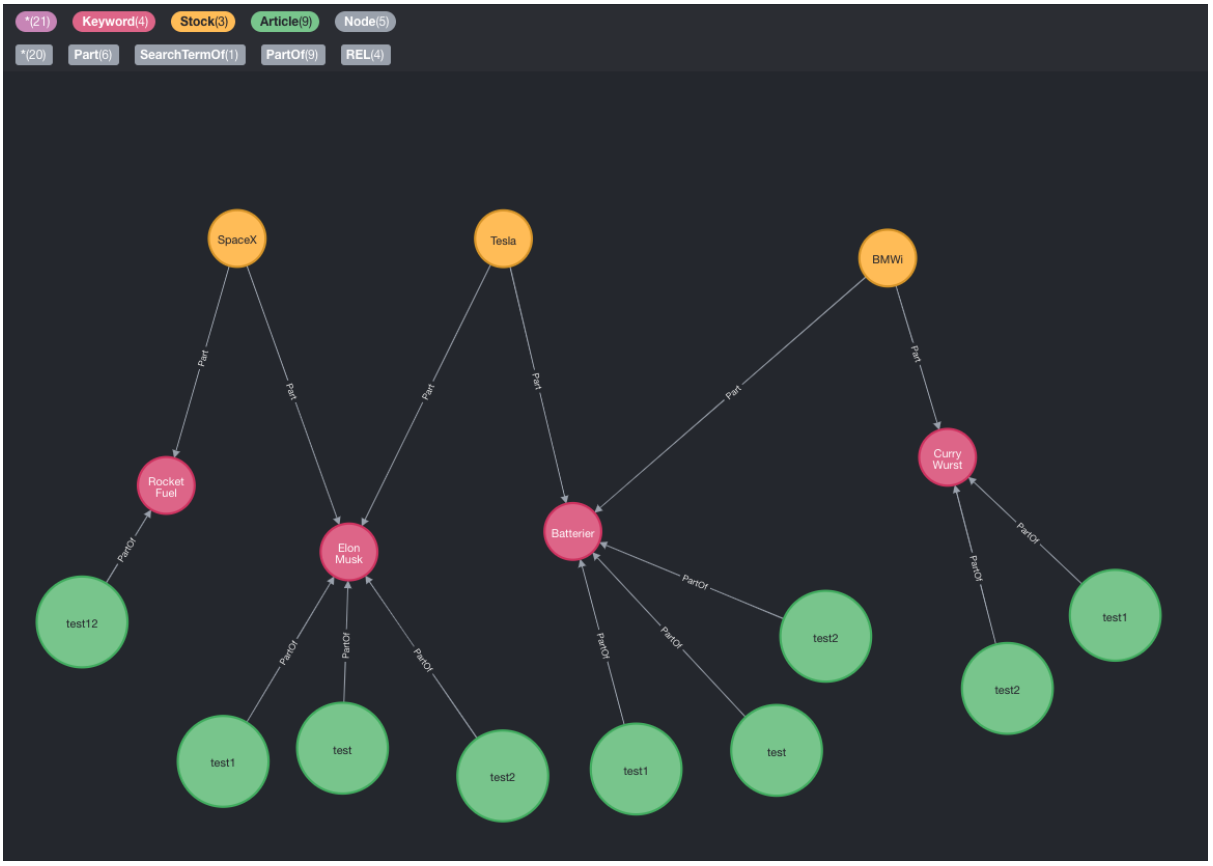
```
/**
 * Opretter artikel i MongoDB.
 * Article.id vil være opdateret med id fra MongoDB hvis persistering lykkes.
 *
 * @param article Article object
 * @return true hvis article persisteres ellers false.
 */
private boolean createMongoArticle(Article article) {
    if (article.getId() == null) {
        Document doc = new Document("content", article.getContent());
        articles.insertOne(doc);
        String id = doc.get("_id").toString();
        if (id != null) {
            article.setId(id);
            return true;
        }
    }
    return false;
}
```

Neo4j - graph database

I forbindelse med at en bruger tilføjer søgeord til en aktie, skal systemet komme med relevante forslag til yderligere søgeord, baseret på andre brugeres tilføjelser til samme aktie.
Til denne funktionalitet har vi valgt neo4j, da en graph database er skabt til at kunne traversere sine noder meget hurtigere end eksempelvis at foretage et opslag på 3 tabeller i den relationelle database.
Så selvom vi ville kunne frembringe de nødvendige data ved at foretage sql-forespørgslen herunder, vil vi kunne frembringe de samme data med en cypher forespørgsel hurtigere[2].

I løbet af udviklingen har vi testet traversering af data vha. gds-plugin i neo4j med flg. udgangspunkt. De orange noder er aktier, pink noder er søgeord tilknyttet aktier. De grønne noder vil være artikeldata, dette gør vi ikke brug af for gennemløb af søgeord.

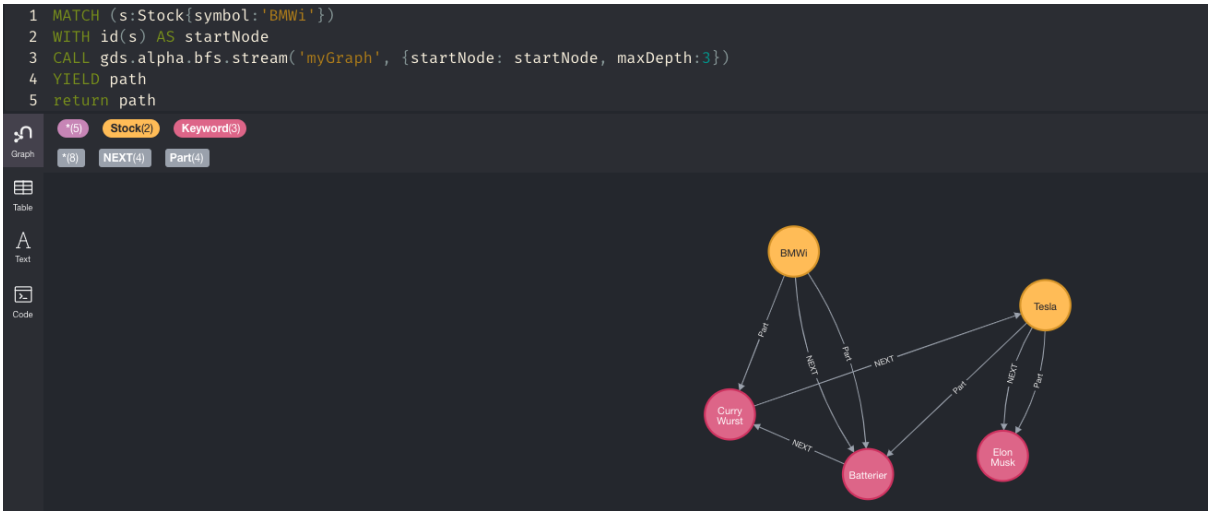
[2] https://neo4j.com/news/how-much-faster-is-a-graph-database-really/



For at kunne gennemløbe ovenstående struktur med udgangspunkt i en aktie, gennem et søgeord, hvortil relationen, af typen ‘Part’, er ‘directed’, må vi lave en graph, der baseres på både Stock- og Keyword noder og som er ‘undirected’.

```
CALL gds.graph.create('myGraph', '*', {Part: {orientation:'UNDIRECTED'}});
```

Med denne graph er vi nu istand til at gennemløbe noderne fra et givent udgangspunkt, f.eks. aktien ‘BMWi’ gennem relationer af typen ‘Part’ til alle andre typer noder. Billedet herunder, viser den resulterende graf, når vi traverserer med en dybde på 3.



Vi kan, på samme måde, trække tekster fra noder med property’en ‘text’ og således opnå en liste af relevante søgeordsforslag, se herunder:

```
MATCH (s:Stock{symbol:'BMWi'})
WITH id(s) AS startNode
CALL gds.alpha.bfs.stream('myGraph', {startNode: startNode, maxDepth:3})
YIELD path
UNWIND [ n in nodes(path) | n.text ] AS texts
RETURN texts
ORDER BY texts
```

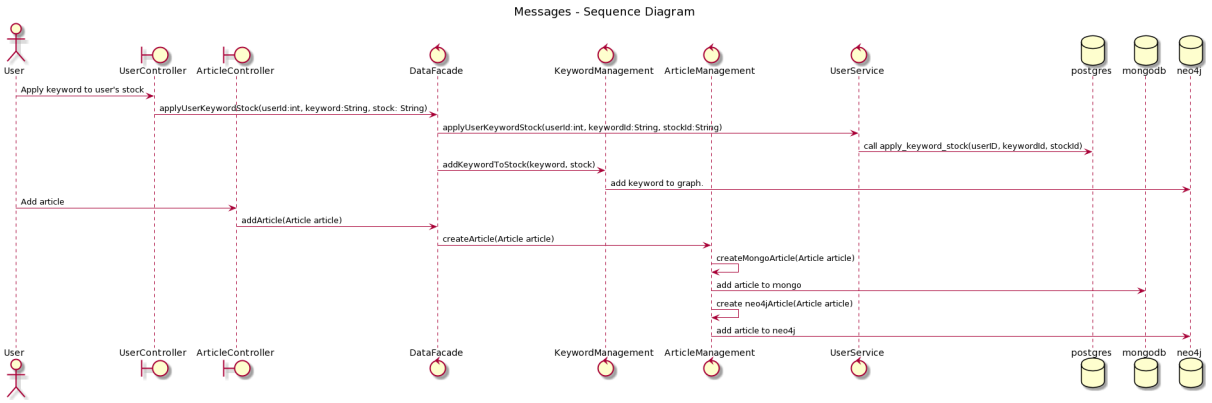
Table | Text | Code

	texts
1	"Batterier"
2	"Curry Wurst"
3	"Elon Musk"

Neo4J giver også mulighed for horisontal skalering i clusters, så vi ville kunne dirigere trafik mod lokale servere hvis det skulle vise sig nødvendigt vha. konfiguration af server policies. Her kunne man også forestille sig et setup, hvor eksempelvis danske aktier følges via skandinaviske servere.

Server teknologi

Serveren, som tilbyder REST endpoints, håndterer de videre funktionskald mod databaserne ved hjælp af en datafacade, services og DTO i tråd med lagdelt arkitektur. Herunder ses en forsimplet udgave, hvor 2 use cases, apply keyword og add article, er beskrevet.



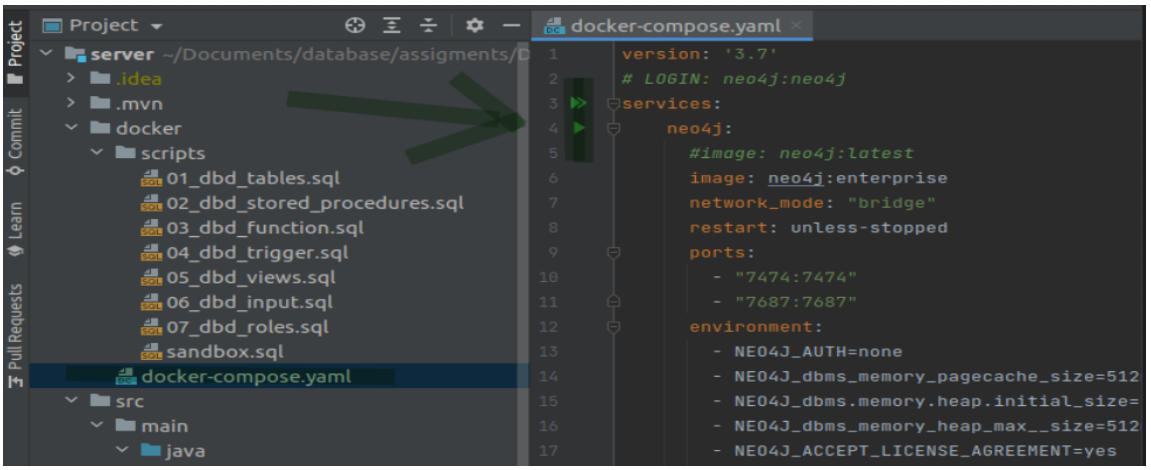
Metode for samarbejde

Da udviklingen har fordelt sig på 2 grupper, har det vist sig, at der har været forskellige tilgange til udviklingen. Dette kan især ses på måden, koden er testet på, samt fejlhåndtering runtime. Vi havde gerne styret projektet lidt mere formelt, med brug af f.eks. scrumboard på taiga.io eller lignende. I stedet deltes gruppen i 2 og opgaven ligeledes.
Vi har bestræbt os på at afholde 3 ugentlige stand-up møder. Også her erfarede vi, at tiden slet ikke var til så formel styring. Forløbet i sin helhed fik således en mere ad-hoc lignende karakter.

Installation og instruktioner

For alle tre databaser, valgte vi at oprette en docker-compose fil for nem implementering. Vores docker-compose-fil kan køres enten i en bash-terminal eller i Intellij[3].

Ind i Intellij:



Ind i bash terminalen: *docker-compose up [-d]*

For at stoppe containeren: *docker-compose down*
Da projektet som omtalt tidligere har været delt mellem gruppen, har vi haft to sideløbende projekter. For at kunne køre serveren er det derfor nødvendigt at inkludere DBPolyglot.jar i server modulet. Jar filen kan findes i projektets rod og tilføjes via Intellij.

Konklusion

Det har, næsten som forudsagt, vist sig at være en prøvelse at få transaktioner, på tværs af databaser, til at fungere. I den del af applikationen, hvor neo4j og mongoDB er involveret, har vi helt udeladt tværgående transaktioner.
Vi kan udlede heraf, at den transaktionsstyring, vi tidligere har fået forærende ved kun at arbejde med en relationel databaseinstans, har dannet skole for vores tænkning. Vi har således haft lidt udfordringer med at forestille os, og videre at implementere, distribueret transaktionsstyring.
Vi synes det havde været interessant at udrulle et database cluster. Under den del af udviklingen, der foregik test driven, måtte vi desværre undlade at benytte et neo4j cluster, fordi en enkelt neo4j testcontainer i Docker alene tog 2-3 minutter at starte.

Vi må slutte, at systemet ikke er produktionsmodent, men det har introduceret os for en række interessante problemstillinger, og -løsninger, som vi ikke er stødt på tidligere, som eksempelvis distribuerede transaktioner og skalering.

[3] for yderligere instruktion, se readme.md på https://github.com/tysker/DBD_Exam

Det videre arbejde

Da applikationen benytter sig af flere databaser, dukker en særlig problemstilling op når data på tværs af disse skal indsættes eller opdateres som en atomar handling; transaktionsstyring.
Vi har i gruppen debatteret emnet og ser umiddelbart 3 muligheder i det videre arbejde:

1. Vi ændrer databasestrukturen således, at behovet for flere databaser elimineres.
2. Vi ændrer databasestrukturen således, at der ikke er afhængigheder mellem dem.
3. Vi accepterer præmissen om 'eventual consistency' og ser således bort fra kravet om atomare ændringer, evt. ved at uddelegere visse opdateringer til message brokers. Dette er en form for distribueret transaktionsstyring, som kunne være interessant at undersøge nærmere.

Derudover er et naturligt videre skridt at implementere et login-system til brugerne. I den relationelle database er de forberedende skridt taget for tildeling af rettigheder til brugere. Videre kunne man forestille sig grupperinger af brugere for enklere tildeling af rettigheder til større mængder.