Project Gutenberg

Which database engines are used

PostgresQL

Den første database, vi har valgt at bruge, i vores projekt er PostgreSQL. Postgres er en SQL database og ORDBMS, som har et kæmpe system bag sig for at optimere hvordan queries bliver kørt på den mest optimale måde.
Vi valgte denne database, da den er objektbaseret og følte det passede godt ind i forhold hvad problemstillingen var. PostgreSQL følger også ACID “bedst” i forhold til de databaser vi har snakket om, og dette er også et vigtigt punkt i forhold til at have en effektiv database.
Postgres bruger tables og rows til at opbygge sit system. Her vil tables blive beskrevet med navn, hvilke typer de skal indeholde og navne på de givne columns. Dette betyder du opretter en form for objekt og laver så et table hvor alle af de samme objekter kan blive lageret.
Postgres gør det også muligt at bruge indexering så vi kan undgå at få en look-up time på O(n) men derimod opnå O(log(n). Eksempelvis er den mest normale form for indexering i Postgres B-tree indexering. B-tree opdeler dataen op i et træ hvor hvert element kan have “x” childs.
Postgres har også mange forskellige egenskaber som vi har lært gennem studiet som måske kunne blive nødvendigt for at effektivisere vores database.

MongoDB

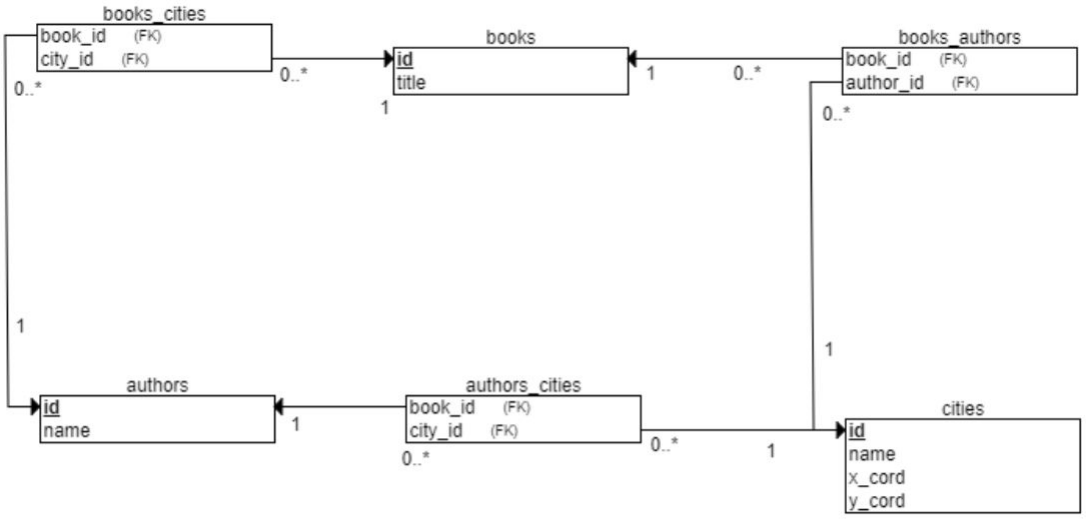
Vores anden database i dette projekt har vi valgt skulle være MongoDB. Vi har arbejdet med MongoDB gennem semesteret og har fået arbejdet med det og fået en god grundforståelse af hvad den kan bruges til. MongoDB variere også en del med Postgres så vi havde også muligheden for at teste denne problemstilling mellem to forskellige database og dermed få følelsen af hvad deres styrke og svagheder er i et reelt projekt.

MongoDB gemmer sit data som JSON og bruger ikke tables som man gør i SQL databaser. Man bruger noget som hedder “collections” som er det tables svarer til i en SQL database. collections indeholder så en masse “documents” som svarer til rows i en SQL database. En stor forskel der dog er i forhold til collections og tables er at tables skal have beskrevet hvad den indeholder og hvilke typer der kan ligge i det gældende table. Med collections kan du frit putte meget varierende documents ind uden dette skaber nogen som helst problemer.
MongoDB har også mulighed for indexering for at opnå hurtigere queries. Igen bruger vi indexering for at undgå at søge en hel collection igennem og dermed undgår vi at få en lookup time på O(n).

Solr
Solr var en “database” som vi stødte på under vores udkig efter den 2. database vi ville bruge. Vi søgte lidt rundt på nettet og ledte efter noget som var god til at håndtere store tekst søgninger.
Solr er bygget til at søge i store tekstfiler som både kan være PDF, word eller txt og meget andet. Eftersom en stor del af projektet indebar at søge i vores bøger følte vi at dette var noget vi ville prøve som den anden database i vores projekt.
Solr gemmer sin data og giver brugeren mulighed for at bruge noSQL til at søge og query efter data som skal bruges.
Solr giver også mulighed for at indexere og derved give hurtige query tider til brugerne.
Vi fik diskuteret med vores projektleder at dette ikke var en 100% rigtig database og blev nødt til at finde noget andet. Vi har dog stadig Solr med i projektet som en halv database.

How data is modeled in the database

Postgres
I postgres har vi bygget databasen op i flere tables for at undgå for meget redundant data og følge normale formene så optimalt vi kunne med stadig egen logisk sans.

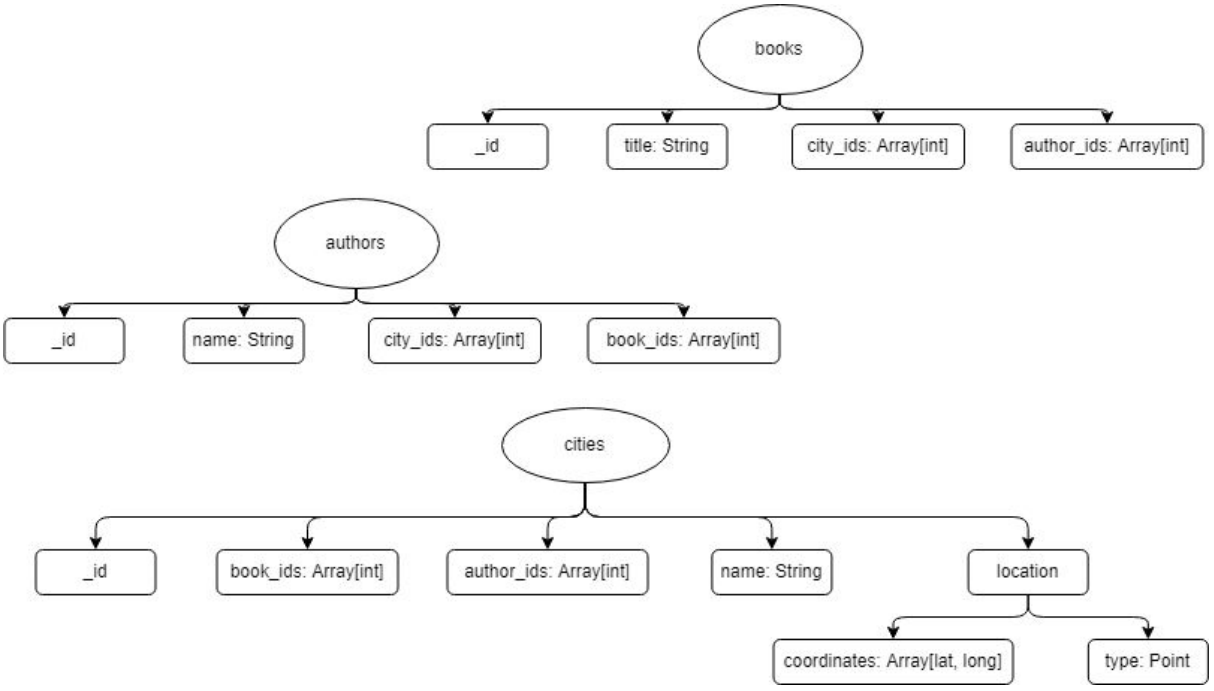


Som vi kan se på diagrammet har vi seks tabeller hvor tre indeholder det data vi kommer til at bruge. De tabeller er books, authors og cities. De tre andre tabeller fungerer som many-to-many tabeller i mellem de tre andre tabeller. Books_authors tabellen indeholder id'er fra books og authors. Så books_authors holder styr på hvilke forfattere har skrevet hvilke bøger. På samme måde holder books_cities styr på hvilke byer forekommer i hvilke bøger. Books_authors indeholder relationen mellem hvilke byer forekommer i hvilke forfatteres bøger. Denne tabel blev lavet for at undersøge, om vi kunne lave vores queries hurtigere og mere læselige. Men det viser sig, at det hverken gør queries hurtigere eller meget mere læselige, så tabellen bliver ikke brugt til noget.

Vi bruger også flere indexes i vores PostgreSQL database. Vi har prøvet med flere forskellige index typer og hvilke kolonner vi satte dem på for at se hvad ville gøre vores queries hurtigere. Postgresql laver automatisk B-tree indexes på alle primary keys. Vi har også lavet et B-tree index for books title, cities name, authors name og alle id'erne i many to many tabellerne. Dette gør at vi får en hurtigere search når vi skal match up de felter. Vi har også lavet en gist index der består af typen point lavet af x_cord og y_cord fra tabellen cities. Vi har valgt at beholde x og y koordinaterne i kolonner i stedet for at lave en kolonne med typen point, fordi at det er mere fleksibelt og query hastigheden er den samme.

Mongodb

I vores MongoDB benytter vi os af databasen *book_database*, denne database indeholder tre collections: *books, authors* og *cities*.



Billedet ovenfor er et vertikalt træ, som beskriver de tre collections og deres fields. Dette er blot for at give en nogenlunde visuel forståelse for hvordan dataen i vores collections er modelleret. Som vi kan se har alle tre collections *_id* og de har har to lister af *_ids*. Vi kan forklare det lidt dybere ved at gå ind i en specifik collection. Så lad os tage *books* som eksempel. *books* har *_id* som er vores index, så har den vores bog titel og til sidst de to arrays med *_ids* og det er disse to arrays, som vi bruger som en "relation" til de andre collections. Da books *city_ids* indeholder en liste over alle byers *_id*, som er nævnt i denne bog kan vi på denne måde matche *books.city_ids* med *cities._id* og hvis vi får et match kan vi dermed bekræfte at det er den by med denne id, som vi leder efter. Denne model holder stik gennem hele databasen data alle tre collections har denne liste med id'er, som matcher op med den collection som vi skal finde.

Dog stikker cities lidt ud, da den har et sub document *location* som er vores GeoJSON objekt. Eftersom vi benytter os af en enkelt lokation (latitude, longitude) bruger vi typen *Point*. Vi har brugt MongoDB's create index metode til at lave en

2dsphere index (*db.cities.createIndex({location : "2dsphere" })*), som gør det muligt at lave queries på en "*earth-like sphere*" givet to koordinater.

How data is modeled in your application.

Vi har modelleret resultaterne af vores queries fra MongoDB og PostgreSQL, så de kan blive behandlet på samme måde. Begge queries fra databaserne giver os en cursor. En cursor er en pointer til vores resultset, som vi så kan iterere over. Dog er måden vi itererer over cursor objektet forskelligt. I MongoDB iterere vi over cursor objektet ved f.eks at sige *item['title]* hvorimod i postgres kan vi sige *item[0]*.
Vi bruger PyMongo og SQLAlchemy til at connecte til databaserne og på den måde kan vi få vores cursor results som vi derigennem kan fetch vores data, som vi bruger i applikationen.

Når vi har fået dataen ud af cursor objektet behandler vi det efter hvor meget og hvordan vores applikation skal bruge det. Til de to første queries når vi iterere over cursor objektet, appender vi det, til en liste, og derefter returnere denne liste. Til den tredje query, laver vi et dictionary, eftersom vi skal bruge bogtitler og tilsvarende koordinater, som vi kan plotte på vores google maps.

How the data is imported

For at få dataene ind i databasen brugt vi et python script. Først kørte vi over alle filerne og for hver af dem hentede vi titlen og forfatter/forfattere ud af bogen. Med den information lavede vi en transaktion hvor vi indsætter bogen og forfatteren sammen med en tsvector af bogen ind i databasen. I samme transaktion sætter vi id'erne på bøgerne og forfatterne som tilhører hinanden ind i books_authors tabellen. Når det script er færdigt kører vi et andet python script hvor vi læser csv filen som indeholder mange byer. Bynavnene og deres koordinaterne lægger vi ned i cities tabellen. Når det er færdigt kan vi bruge PostgreSQL til at lave tekst søgning ved hjælp af tsvectors. Vi laver et select statement hvor vi henter id'erne fra books og cities om bynavnet forekommer i bogen og inserter dem ind i books_cities. Et problem vi fandt ud af senere er at tsvector ikke er case sensitiv. Dette betyder at vi får flere matches forkerte matches.

For at få dataene ind i MongoDB lavede vi et python script hvor vi hentede alt data fra PostgreSQL databasen og lægger det i MongoDB databasen. Vi laver many to many tabellerne om til array i MongoDB. Så hvert document i databasen indeholder to array af id som de relaterer til.

Query test

PSQL

```
get_titles_for_city
0.023286008834838887
get_cities_for_title
0.018172836303710936
get_titles_and_cords_for_author
0.050129938125610354
get_title_for_cords
2.0506751537323
```

MongoDB

```
get_titles_for_city
0.056097912788391116
get_cities_for_title
0.0037331342697143554
get_titles_and_cords_for_author
0.007534456253051758
get_title_for_cords
0.28402645587921144
```

Vi har kørt nogle forsøg på vores queries i hver database for teste hvor hurtigt de kører. Hvis vi tager postgres' to øverste queries ligger de meget tæt sammen på ~0.02 sekunder. MongoDB's to øverste queries har meget stor forskel mellem hvor hurtige de er. Her har vi 0.056 sekunder og 0.003 sekunder. Så PSQL er hurtigere ved den første query men langsommere på den anden.
Ved den tredje query er MongoDB dog meget hurtigere og tager kun ~0.007 sekunder hvor PSQL tager ~0.05 sekunder. Her er MongoDB altså omkring 6 gange så hurtigt.
Den sidste query er MongoDB igen meget hurtigere med 0.28 sekunder mod PSQL's 2 sekunder. Igen er MongoDB cirka 7 gange så hurtigt i forhold til hvad PSQL kan præstere.
Så vi har altså MongoDB som den hurtigste database med disse test med 3 ud af 4 queries der er hurtigst og gennemsnittet af alle 4 queries er MongoDB også meget hurtigere.

Recommendation

I forhold til vores query tests kan vi fastslå at MongoDB er hurtigst til at læse dataen. Selvfølgelig er dette kun i læsehastighed og vi har ingen test på omkring hvor hurtigt de to databaser ville klare sig i INSERT og UPDATE.
Derfor vil vi foreslå MongoDB til denne problemstilling som er givet i dette projekt. Dog kan det ikke udelukkes at opbygningen af PostgreSQL kunne laves om for at opnå en højere hastighed for at gøre det lige så hurtig som MongoDB. Det virker dog til at MongoDB er 5 gange hurtigere end PSQL over de 4 queries som er givet som opgave.