Rapport
Gutenberg projekt

Lavet af:
Yosuke Ueda (cph-yu173)
Benjamin Schultz Larsen (cph-bl135)
Jacob Sørensen (cph-js284)

Indholdsfortegnelse

Forord

Vi har i denne besvarelse hovedsageligt været fokuseret på de 2 områder af databehandling, vi har fundet mest relevante. Rigtigt-heden (integriteten) af data og svarhastigheden for forespørgsler foretaget på data.

Valg af databaser

Vores gruppe valgte at bruge Mysql og MongoDb databaser i løsningen af denne opgave. Ingen af os i gruppen har tidligere haft andet end overfladisk erfaring med MongoDb, og vi var alle 3 interesseret i at lære teknologien bedre at kende. På den anden side følte vi os alle 3 godt orienteret i brugen af en "klassisk" relations database, og selvom det ikke var os alle i gruppen der havde så meget erfaring med lige akkurat geo-spartial funktionaliteten, var det vores opfattelse, baseret på hvad vi havde set i undervisningen, at Mysql har et rigt udvalg af metoder til behandling af denne type.
Vi mente derfor at det kunne være interessant at sammenligne hvor godt MongoDb håndterede data i forhold til Mysql.
En *key-value store* type database såsom f.eks. Redis ville nok også have været velegnet da den dels lever 100% i ram (hastighed), og dels fordi de queries vi skal besvare i opgaven kun er read-baseret (igen hastighed).
Selvom Neo4j også er ny teknologi for os, faldt valget dog på MongoDb mellem de to.

Arbejdsplan

1) data requisition - hente data fra Gutenberg
2) data exploration - undersøge filerne fra Gutenberg
3) data preparation, manipulation - frasortere ikke brugbart data
4) data importation - indlæse data i vores 2 valgte database teknologier
5) data query performance examination - bygge og optimere queries på data
6) data presentation - bygge en applikation til fremvisning af resultatet

Importering af data (requisition/exploration)

Til importering af data brugte vi det udleverede script fra opgavebeskrivelsen (https://github.com/datsoftlyngby/soft2018spring-databases-teaching-material/blob/master/book_download/download.sh).
Dette gav os 37348 zipfiler. Af disse filer konstaterede vi at nogle af dem ikke var tekstfiler som annonceret men derimod *.jpg filer - andre filer fulgte ikke den forventede konvention for navngivningen af tekstfiler; langt den overvejende del af filerne har et navn der består af et tal mellem 1 og 100.000 og en .txt extension. Følgende kommandoer blev kørt i konsollen for at "rydde op" i de downloaded filer:

```
unzip \*.zip       (udpakke alle zipfiler i mappen)
rm *.zip           (sletter alle zipfiler fra mappen)
rm *.jpg           (sletter alle filer med *.jpg extension)
```

```
rm -R -- */        (sletter alle mapper fra mappen)
rm [a-z]*          (sletter alle filer hvis filnavn ikke består af tal)
rm Common-README   (sletter filen "Common-README")
```

Dette gav os 37001 bøger som tekstfiler på formen [xxxxx.txt], hvor x er et ciffer mellem 0-9

(1 - (37001/37348)) * 100 = **0.92 %**

Cirka 1% af det data vi hentede fra Gutenberg er blevet frasorteret.
Det er lidt svært at sætte en tidsramme på hvor lang tid der gik med at downloade filerne fra Gutenberg, da vores script i første omgang blev afbrudt efter kun at have hentet ca 1.100 filer, og resten af filerne blev downloadet i anden omgang. *Baseret på den indbyggede ventetid på 2 sekunder i download-scriptet anslår vi at det har taget os ca. 21 timer.*

Udtræk af data (preparation, manipulation).

Ved nærmere undersøgelse af filerne vi havde hentet fra Gutenberg, fandt vi, at fælles for dem var at de alle havde den sammen afslutning på teksten, en *footer* tilføjet af folkene fra Gutenberg, og ikke en del af selve bogen - Lidt uheldigt for denne opgave, indeholdt den tilføjede footer flere navne som NER genkender som bynavne

*"...The Project Gutenberg Literary Archive Foundation is a non profit 501(c)(3) educational corporation organized under the laws of the state of **Mississippi** and granted tax exempt status by the Internal Revenue Service. The Foundation's EIN or federal tax identification number is 64-6221541. Contributions to the Project Gutenberg Literary Archive Foundation are tax deductible to the full extent permitted by U.S. federal laws and your state's laws.*

*The Foundation's principal office is located at 4557 Melan Dr. S. **Fairbanks**, AK, 99712., but its volunteers and employees are scattered throughout numerous locations. Its business office is located at 809 North 1500 West, **Salt Lake City**, UT 84116, (801) 596-1887. Email contact links and up to date contact information can be found at the Foundation's web site and official page at www.gutenberg.org/contact ..."*
(uddrag af footeren)

Heldigvis var begyndelsen på footeren markeret tydeligt i hver fil så den var nem at fjerne med simpel RegEx
Fælles for tekst filerne var ligeledes begyndelsen på filerne, *headeren* - dette er ligeledes tekst tilføjet af folkene fra Gutenberg. Headeren indeholder oplysninger om bogen, og blandt dem var vi interesseret i forfatter, title, part (hvilken del af bogen - hvis det er en bog i flere dele).
Disse oplysninger udtrak vi ligeledes ved brug af RegEx - og da det er givet at headeren altid kommer som det første i tekstfilen, begrænsede vi os til kun at udtrække de første 25 linjer- dels for ikke at få misvisende resultater (*hvis noget af teksten i bogen faktisk*

indeholder ordet "Author") og også for at optimere hastigheden (*der er ingen grund til at RegEx skal gennemløbe hele filen, når vi allerede har resultatet efter de første 25 linjer*)

Samtlige RegEx brugt i forbindelse med udtræk/fjernelse af oplysninger på tekstfilerne kan findes her (https://github.com/benjaco-edu/db-guttenburg/blob/master/extractData/extract.js)

Name Entity Recognizer

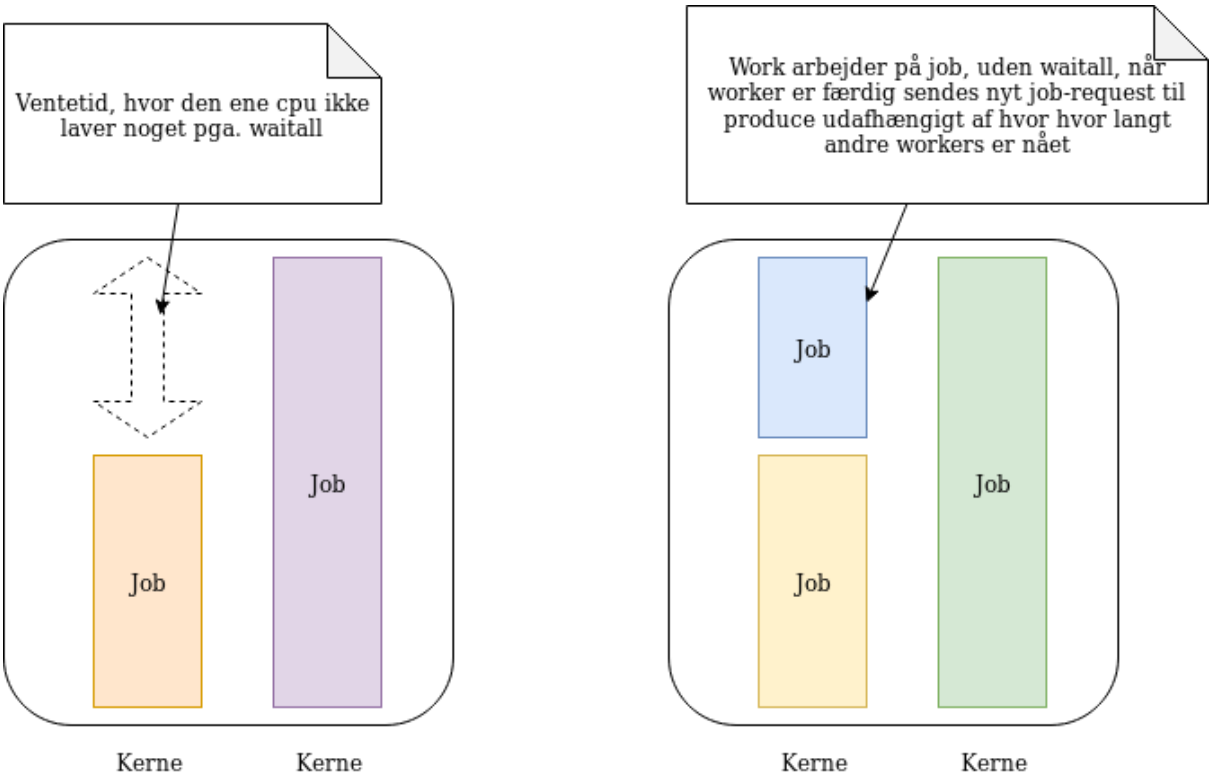
Fase 2 bestod nu af, at få gennemløbet alle de tekstfiler vi havde fra Gutenberg for at undersøge hvilke af filerne indeholdt bynavne, og hvilke bynavne de indeholdte. Endnu en gang var der inkluderet et hjælpsomt *hint* i opgavebeskrivelsen (https://nlp.stanford.edu/software/CRF-NER.html). Name Entity Recognizer (NER) udviklet på Stanford vha. machine learning.
Meget kort fortalt; er NER en algoritme der kan genkende navne i tekst (online demo kan afprøves her: http://nlp.stanford.edu:8080/ner/). Den har 3 forskellige operations niveauer, hvor søge evnen - på bekostning af tidsomkostningen kan justeres.
Ved test af NER på vores data material fandt vi i første omgang at den kunne gennemløbe ca 4 tekstfiler pr. minut.
En hurtig overslagsberegning for en "almindelig" hjemmecomputer med 4 kerner skulle derfor have

((37001 filer / 4 pr. minut) / 60 minutter) = **154.2 timer (6.4 dage)**

uafbrudt beregningstid til at kører NER på samtlige af de bøger(tekstfiler) vi havde fra Gutenberg.

Første iteration af vores algoritme der havde ansvaret for at kalde NER på en tekstfil blev lavet som et flertråds algoritme, men benyttede *WaitAll*, hvilket introducerer kunstig(unødvendig) ventetid for de tråde der allerede har færdiggjort deres job.

I anden iteration skiftede vi til at bruge en *producer/consumer* - hybrid, hvor hver *worker* spørger produceren om et nyt job så snart den er blevet færdig, uden at vente på de resterende tråde bliver færdige. *Se illustration nedenfor*

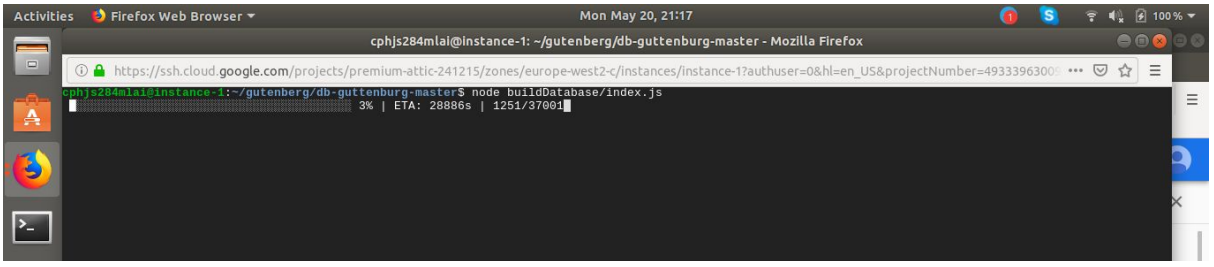


Flertråds-algoritme med waitAll til venstre, og producer/consumer hybrid uden waitAll til højre.

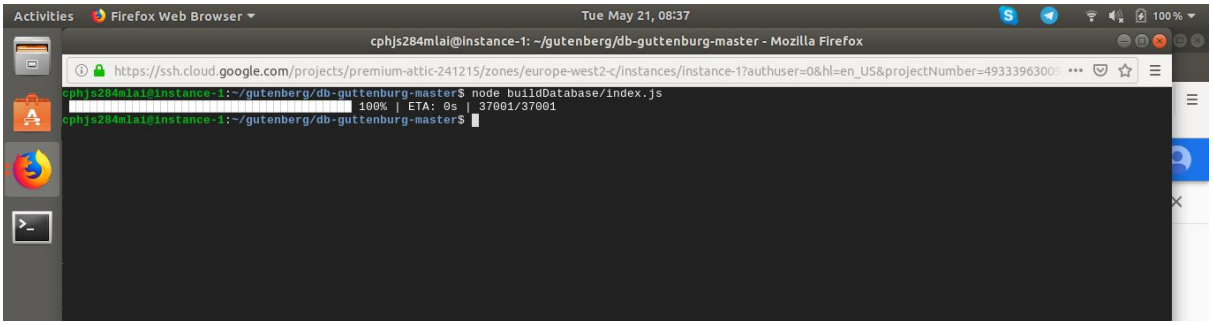
Denne forbedring bragte algoritmens løbetid ned på at den nu kunne nå ca 13 bøger pr. minut.

((37001 filer / 13 pr. minut) / 60 minutter) = **47.4 timer (1.9 dage)**

Stadigvæk ganske belastende for en hjemme-PC at køre uafbrudt i 2 dage, og nu var der fuld belastning på alle kernerne næsten 100% af tiden.
Derfor oprettede vi en konto hos Google Cloud, hvor vi, for den medfølgende kredit på 300$ købte os til computer-kraften der kunne køre NER på samtlige af vores tekstfiler på ca 11 timer (24 kerner, SDD, 80GB ram)



Progressbar for NER på tekst filerne - Bemærk tiden i toppen af screenshot



Progressbar for NER på tekst filerne - Bemærk tiden i toppen af screenshot. Ovenfor ses algoritmen har arbejdet færdig.

Vi har valgt at fremhæve koden vi kørte på Google Cloud i en artefakt for sig, den kan findes i artefakten
(https://github.com/benjaco-edu/db-guttenburg/blob/master/Artefakt%20ProducerConsumer.pdf)
Den kan selvfølgelig også findes i den medfølgende kode i mappen "extractData" med navnet index.js.
(https://github.com/benjaco-edu/db-guttenburg/blob/master/extractData/index.js)

Resultatet af NER.

Outputtet fra vores workers var filen "booksAndCities.json" (findes i github-repo'et som booksAndCities.zip, da størrelsen på filen overstiger de tilladte 100MB.)
Som extensionen antyder indeholder den et (ganske stort) json objekt på formattet:

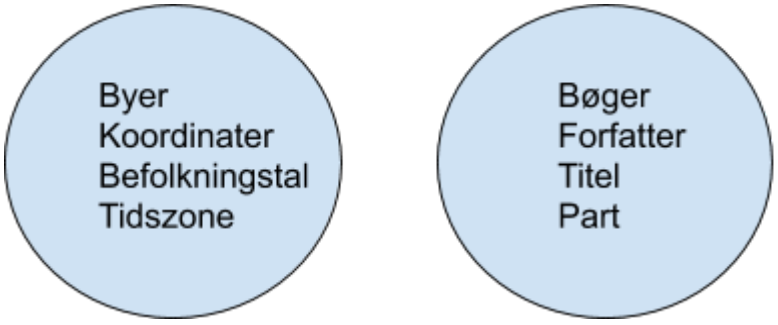
```
{
        filnavn:{
                Part: <bogens part>, (null hvis bogen ikke har en part)
                AuthorName: <forfatternavn>,
                Title: <bogens titel>
                Cities: [{cityIndex: <By-index>, (index svarer til nummer fra CSV-fil)
                        index: <location in text>}]
        }
        .
        .
        .
}
```

hvor filnavn er navnet på den fil som indeholder et bynavn.
Efter dette gennemløb fandt vi ud af at den havde fejlede på 72 filer, disse filer var store og skulle derfor bruge mere ram end hvad standard tillod. Disse filer blev kørt separat på en af vores egne computere med større hukommelse begrænsninger.
Alt i alt bliver bynavne nævnt **5.309.839** gange i de 37001 bøger vi har hentet.
Det eneste vi manglede nu var at importere vores data til vores 2 databaser.

Modellering af data (Importation)

Vores applikation er skrevet i javascript, og sammenfaldet mellem modelleringen af vores data i koden og modelleringen af data i mongoDb er påfaldende, faktisk kan man i selve *shell'en* til mongoDb skrive javascript - og selv om det ikke er json objekter der ligger i mongo, men istedet objekter af typen Bson, er notationsformen for dem begge den samme. I det følgende afsnit ser vi nærmere på vores modellering af data i databaserne og koden.

Som udgangspunkt har vi delt data op i 2 kategorier:
data der er relevant for byer (bynavn, koordinater, befolkningstal) og data der er relevant for bøger (forfatter, titel, part) - vi har konstateret at nogle af bøgerne fra Gutenberg indeholder oplysninger om hvilken *part* (del) tekst-filen er en del af.

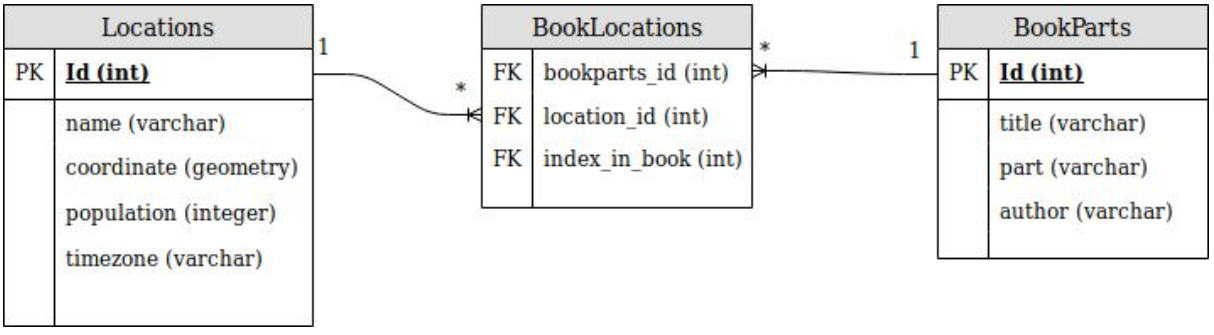


en visualisering af vores 2 grupper af data.

Alle oplysninger om byerne vi bruger i vores besvarelse kommer fra CSV-filen (https://github.com/benjaco-edu/db-guttenburg/blob/master/cities15000.txt) der blev udleveret sammen med opgaven. Vi har valgt at begrænse os til byer med 15.000 eller flere indbyggere.

Bindeledet mellem disse 2 grupper af data er oplysninger om hvilke bøger der indeholder hvilke bynavne.

I MySql databasen, som er en relations-database ser vores modellering af data derfor således ud:



Denne opdeling af data; i tabeller med 1-til-mange relationer i mellem tabellerne, giver os tabeller på 3 normalform som er særdeles velegnede til skalering hvis en fremtidig udvidelse skulle ske.

For Mongo-databasen skal vi træffe et valg mellem at lægge alle vores data en i collection, eller dele data op på flere collections.

BooksAndLocationsCollection

```
{
  author:<"Abraham Lincoln">
  part: <" ">
  title: <"Lincoln's Second...">
  locations:
    [
      {
        {
          name: ""
          coordinate:[-28.94 153.48]
          population: 591473
          timezone:"Europe/Brussel"
         booksRef:[<....>]
        },
        indexInBook:<1728667>
      }
    ]
}
```

Ovenfor ses et eksempel hvordan data kunne modelleres i mongo hvis det skulle opbevares som embedded dokument

BookCollection

```
{
 author:<"Abraham Lincoln">
 part: <" ">
 title: <"Lincoln's Second...">
 locations:
  [
    {
      locationref: <6>
      indexInBook:<1728667>
    }
  ]
}
```

LocationCollection

```
{
      name: ""
      coordinate:[-28.94 153.48]
      ref(id): 6
      population: 591473
      timezone:"Europe/Brussel"
      booksRef:[<....>]
}
```

Ovenfor ses et eksempel på data opdelt i 2 collections og forbundet med en reference.

Vælger vi den første løsning(alt samlet i et dokument) er der en besparelse at hente i form af database operationer, idet vi kan komme omkring opdatering af et dokument i en atomisk handling. Læsning af dokumenter er generelt også hurtigere hvis alle data ligger samlet (embedded) i et enkelt dokument ("...*In general, embedding provides better performance for read operations, as well as the ability to request and retrieve related data in a single database operation. Embedded data models make it possible to update related data in a single atomic write operation.*...").
(fra: https://docs.mongodb.com/manual/core/data-model-design/)

Ulempen er dog at dokumenterne vil indeholde gentagende oplysninger (hver gang en bog omtaler byen København, skal den medtages på dokumentet). Herudover er vi ikke istand til at forudsige hvor store vores bynavn-array vil blive på hvert bog-dokument i det vi ikke kan sige noget om hvad der vil blive skrevet af omfangsrige tour-guides senere, dette bliver ikke nødvendigvis til et problem for vores applikation, men repræsenterer dog et område vi skal være opmærksom på ved skalering.

Vælger vi derimod den anden løsning(data opdelt i flere(2) collections), skal vi vedligeholde 2 collections hver gang en ny bog bliver tilføjet, idet vi både skal opdatere den collection der indeholder bog data, og den collection der indeholder byer, derudover skal der også oprettes ekstra felter (*field*) i dokumenterne, så der kan skabes en reference mellem oplysningerne i de 2 collections (en slags relation om man vil). Til gengæld kan vi helt undlade at indlejre vores by-data objekt i vores bog-dokument og blot have en liste (*array*) af referencer.

Efter første gennemkørsel af data på vores RegEx til udtræk af data kunne vi konstatere at der var ca 2000 tekstfiler som på den ene eller anden måde ikke fulgte konventionen vi tidligere havde set i "headeren".
Vi opsamlede disse filer og kørte en udvidet RegEx på dem - resultatet efter denne kørsel var nu at kun ca 500 filer, der ikke umiddelbart fulgte konventionen for "header", tilbage.
Som gruppe tog vi den beslutning ikke at synke mere tid ned i at jagte udtømmelige RegEx og valgte i stedet at lave en scraper til at slå title og author op direkte på Gutenberg's hjemmeside.
Da scrapperen havde gjort sin gerning hvade vi nu anvendeligt data for alle vores tekstfiler som nu skulle lægges ind databasen.

Tilføjelse af Index.

Når der skal tilføjes et index på et field/kolonne i en database er det vigtigt at indsætte det korrekt - specielt i mongoDb er indexering af field omkostningsfuldt for pladsforbruget.
Faktisk var det i de tidligere versioner af mongo ikke muligt at tilføje index mere end et vist antal gange før man stødte på en det tilladte grænse.
Vi har valgt at sætte index på følgende fields i mongoDb:

name i **Location** tabellen er blevet indexeret da man skal kunne finde en by på den 1. query
title i **BookParts** tabellen er indexeret fordi vi laver et opslag på title field som svar på den 2. query
author i **BookParts** tabellen er indexeret fordi vi laver et opslag på title field som svar på den 3. query

coordinate i **Location** tabellen er indexeret for at kunne finde byer i nærheden af det valgte punkt i query 4

- **name** i Locations collection, fordi vi laver et opslag på title field som svar på den 1. query
- **id** i Books collections, dette er tilføjet fordi id'et det field der skabes en reference til når vi bruger $lookup i vores aggregate pipeline
- **title** i Books collection, fordi vi laver et opslag på title field som svar på den 2. query
- **id** i Location collection, dette er tilføjet fordi id'et er det field der skabes en reference til når vi bruger $lookup i vores aggregate pipeline
- **author** i Books collection, fordi vi laver et opslag på title field som svar på den 3. query
- **coordinate** i Locations collection, flere årsager til at dette field bliver indexeret, for det første eksekverer mongoDb ikke en query med geospartial data uden at der er sat index på, for det andet bruges dette field i til besvarelse af query 4.

Et særtilfælde af query 4 og mysql workbench, vi er stødt på
Vi startede med at bygge den 4 sql query op ved at tilføje et where hvor vi testede for om et givent elements distance til det valgte punkt var mindre end den specificeret rækkevidde - denne funktion var simpel, men den kunne ikke benytte sig af vores index.
Derefter forsøgte vi med et alternativ og se om punkterne var inde for et vist område; vi lavede det forespurgte punkt om til et polygon ved hjælp af ST_Buffer funktionen.

Denne nye query kører noget hurtigere, da den benytter vores index, og den har en enormt reduceret query cost (fra ca 2mill. ned til 200 query cost) - dog virker query cost ikke til at kunne stole på ved denne query.

Dog virker ST_Buffer kun i et fladt koordinatsystem, så derfor skal punktet transformeres frem og tilbage. Dette introducerer dog en lille præcisions usikkerhed i denne manøvre, den kunne omgås ved at køre ST_Distance funktionen på de matchede punkter.

Se link (https://github.com/benjaco-edu/db-guttenburg/blob/master/Artefakt%20%20specielcaseQuery%204.pdf)

Indexes i mongo

Som skrevet tidligere, så kræver MongoDB et spacial *2dsphere* index for at kunne arbejde med de geo funktioner. Med dette index var det muligt at bruge *$geoNear* som kan modtage et punkt og en maks distance fra punktet, og herefter filtrere alle dokumenter væk som ikke opfylder kriteriet.
Vi brugte også et index på *bookId* og *locationId* for at kunne binde dokumenterne sammen med *$lookup*, vores erfaringer med denne kombination er at dokumenter kan forbindes nærmest instantant

Modellering af data i applikationen

De queries der bliver brugt til at udtrække data fra til databaserne kan findes linket (https://github.com/benjaco-edu/db-guttenburg/blob/master/List%20of%20used%20queries.pdf)

“Outputtet” fra vores queries er næsten på samme form, når de sendes fra serveren; Grunden til de ikke er på fuldstændig samme form er at indsatsen det ville kræve at skabe data på uniformt format er ikke udbyttet værd, sagt på en anden måde, det ville skabe unødig kompleksitet i databaserne. I stedet tilpasses data så det forstår på samme format når det når til applikationslaget - vi er interesseret i at have data på samme format fordi så kan vi nøjes med et sæt metoder i applikationslaget til at behandle data fra begge typer af databaser og dermed skabe en løs kobling mellem disse lag.
Generelt er data på lidt fladere form i vores applikation i forhold til hvordan det bliver opbevaret i vores databaser.

Tidsmåling

Samtlige tidsmålinger kan findes i på linket (https://github.com/benjaco-edu/db-guttenburg/blob/master/Tidsm%C3%A5lingDatabaser.pdf)

I tabellen nedenfor er hver af de 4 queries vi skulle besvare i opgaven blevet kørt 5 gange på begge vores database teknologier.
Alle målinger er foretaget på en hjemme PC der kører med windows 10.

I tabellen nedenfor angiver kolonnen "QueryType" nummeret på den query der er blevet kørt notationen "MI" og "UI" betyder henholdsvis med index og uden index. Eksempelvis vil notationen Q1UI betyder "query nummer 1 kørt uden index".

QueryType	måling 1 ms	måling 2 ms	måling 3 ms	måling 4 ms	måling 5 ms	GNS.
DB: MySql						
Q1UI	47	47	47	47	47	47
Q1MI	31	32	16	15	31	25
Q2UI	15	0	0	15	16	9,2
Q2MI	15	15	15	0	0	9
Q3UI	47	78	32	47	78	56,4
Q3MI	16	47	31	47	32	34,6
Q4UI**	767	775	748	774	763	765,4
Q4MI**	19	19	21	22	20	22,2
DB: MongoDB						
Q1UI	42980	42980	44812	43466	44484	43744,4
Q1MI	647	726	719	697	702	698,2
Q2UI	165	158	122	124	125	138,8
Q2MI	5	4	3	2	2	3,2
Q3UI	61816	58125	64073	59499	60141	60730,8
Q3MI	1427	1544	1342	1548	1343	1440,8
Q4UI	ikke	ikke	ikke	ikke	ikke	ikke

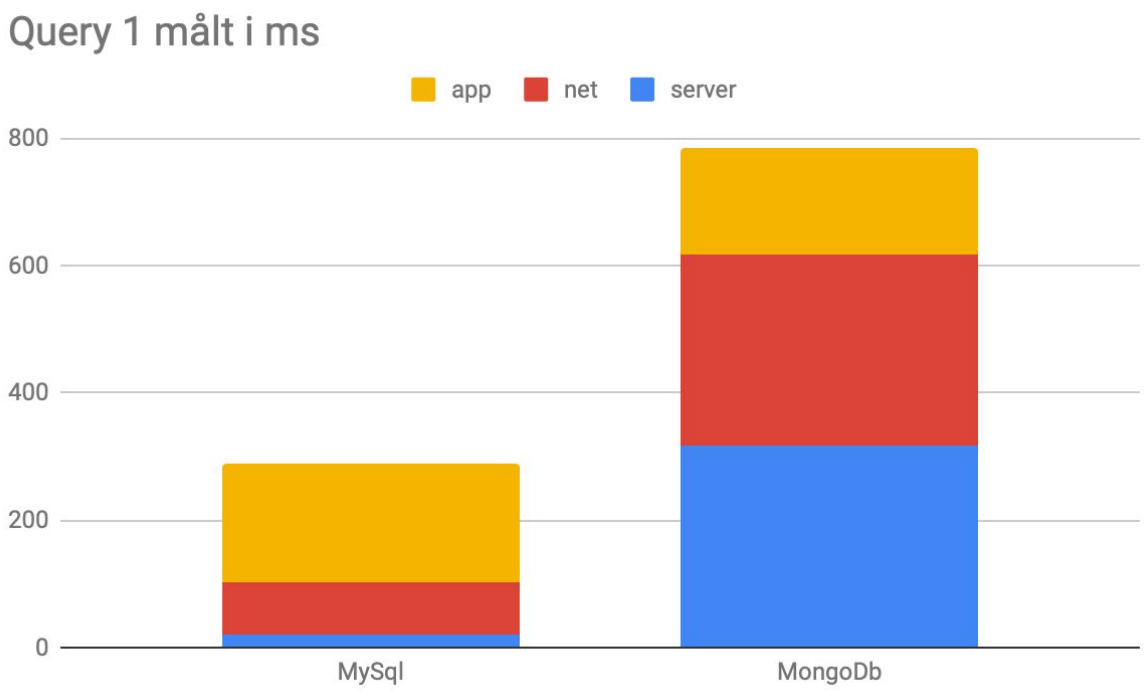
	muligt	muligt	muligt	muligt	muligt	muligt
Q4MI*	157	159	155	166	158	157

** Da det ikke var muligt at få explain("executionStats") til at virke sammen med brug af geospartial data, har vi istedet brugt nodejs til at time hvor lang tid mongodb er om at svare tilbage på en query*

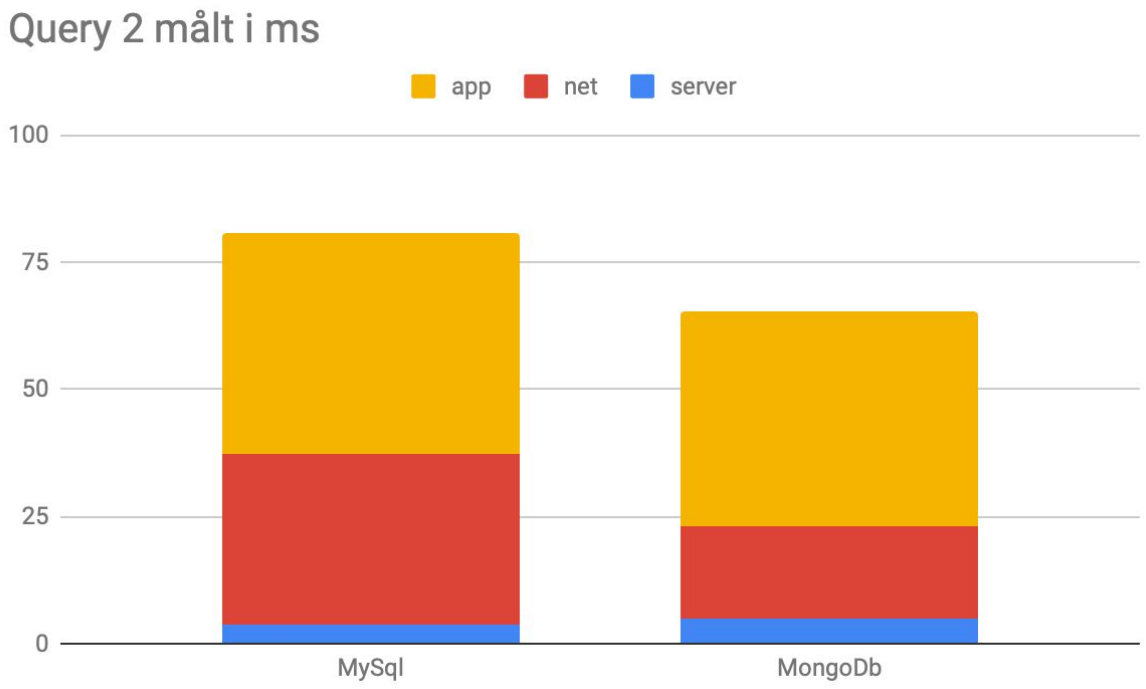
*** for at kunne sammenligne målingerne fra MySql med resultatet fra mongo, er disse målinger også målt vha. nodejs (selvom executionplaner ikke har nogle problemer med at give feedback på geospartial data)*

Applikationstimings diagrammer

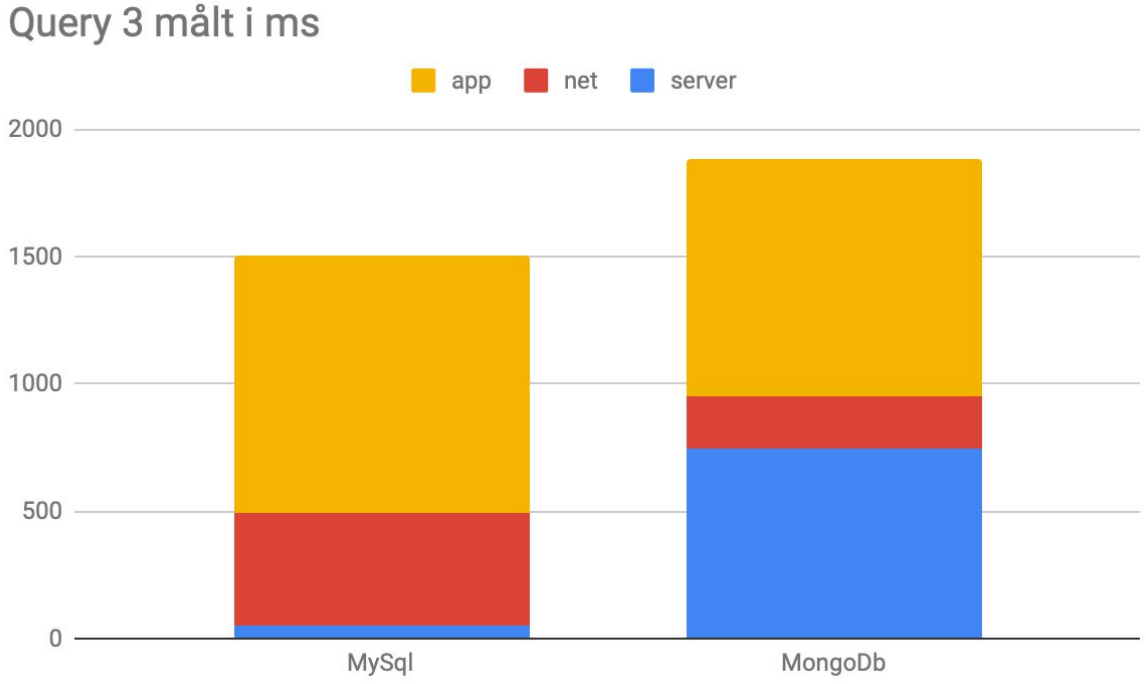
De viste målinger er baseret på et gennemsnit over 5 gennemkørsler, de underliggende tal kan ses på https://github.com/benjaco-edu/db-guttenburg/blob/master/Artefakt%20Applikationstiming.pdf



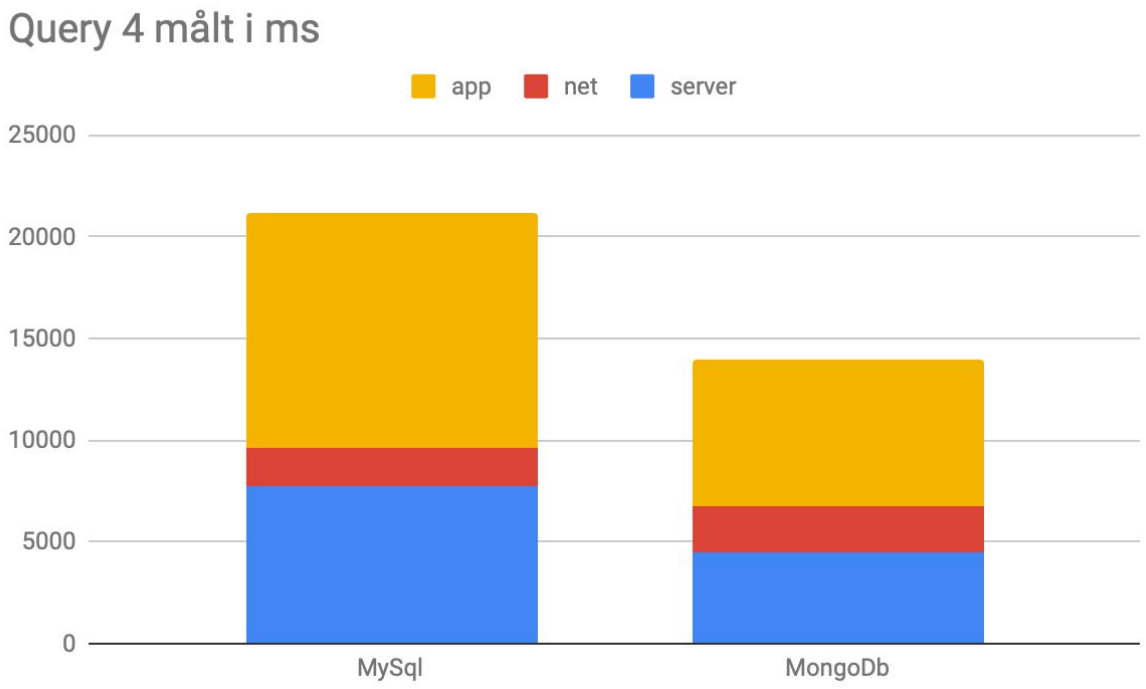
Vi kan konstatere at MySql udfører query væsentlig hurtigere en mongoDb



Ovenfor kan vi konstatere at mongodb er hurtigt, tilsyneladende pga netværks payload er mindre



Ovenfor ses det at mongo skal have mange gange mere eksekveringstid på serveren, men på trods bruger mindre netværks kapacitet



Begge teknologier bruger meget eksekveringstid på query 4, nok pga. geospatial data typen MySql server tiden er utrolig høj i forhold til den tidligere viste statistik, det er fordi den første test blev udført for en lokation uden så mange relevante punkter i nærheden (120 km fra odense) og det overståede diagram er baseret på data tæt på London (N51W0 200km omkreds).

Opsummering og konklusion

Tidsmåling for queries

Ud af tabellen fra ovenstående afsnit ses det tydeligt at MySql har hurtigere eksekveringstid i forhold til MongoDb. Både med og uden index på tabellerne(collections)
En af årsagerne til dette kan være at mongoDB er hurtigere til at returnere data når det hele ligger samlet som et stort dokument, og ikke som vi har gjort hvor vi har opdelt data over flere collections (jævnfør: https://docs.mongodb.com/manual/core/data-model-design/)
Til gengæld er der sparet plads ved ikke at indlejre et helt location dokument i hvert bog dokument.
Vi kan også konkludere at et af de salgspunkter mongoDb slår sig op på - "at dokument databaser kan laves uden nogen form for *schema*" kun i teorien er rigtigt nok, i praksis ender man med at have en "schema" sat op (som minimum) i koden.

Tidsmåling for server, netværk og applikation

Som en sidebemærkning bør det nævnes at der er lagt et kunstigt delay ind i applikationen html'en er sendt til browseren. Delay'et er på ialt 4 frames og vi estimerer at det ikke ligger mere end 15-20 ms ekstra oveni den samlede tid. Delay'et er lagt ind for at "tvinge" browseren til at rendere henholdsvis div'en med resultat data og kortet hvorpå data visualiseres før tidsmålingen stoppes.

Som det ses i diagrammerne ovenfor ligger det overvejende tidsforbrug i applikationen (rendering af liste i venstre side og kortet). Noget af tidsforbruget for applikationen skal også tilskrives det faktum at det er en Google map API vi bruger, så applikationen skal også vente på at få svar tilbage fra Google.
En anden ting er når vi tilføjer flere tusind elementer ude i venstre side af applikationer, kan vi konstatere at tidsforbruget vedr. renderingen af denne liste er signifikant.
Vi ser yderligere at server tidsforbruget fylder meget for mongoDb specielt for query 1 og 3. Sandsynligvis fordi vi har delt op i 2 collectioner og serveren skal oprette en reference imellem dem før den kan komme tilbage med et svar. Vi ser høj aktivitet på serveren for begge databaser for query 4, dette skyldes formentlig at geospartial data typerne er "tungere".
Vi konstaterede yderligere at Mysql tilsyneladende bruger mere netværkskapacitet end mongoDb når der skal sendes data tilbage til klienten (applikationen).
Vi kan konstatere at mongoDb er mere velegnet til at håndtere query 4 hvis det drejer sig om store datamængder, ellers er Mysql bedre egnet til at besvare query 4.

Anbefaling af teknologi.

Når vi skal anbefale en teknologi for denne type applikation, handler det for os om 2 ting: skalering, og eksekverings hastighed.
Fra vores tidsmålinger fremgår det tydeligt at eksekveringen af queries falder ud til Mysqls favør.
Vi mener ikke at der er den store forskel på hvor godt disse to typer af databaser skalere for denne type projekt - en måde man kunne tjekke dette på; kunne enten at skabe ekstra kunstig data, eller kører de samme queries på et halveret datasæt og se på resultatet.

Vi har valgt at lave applikationen med NodeJs og Express, en anden mulig undersøgelse man kunne udføre var at teste drivere for andre programmeringssprog.

På baggrund af samtlige ovenstående tabeller, undersøgelse og diagrammer samt ved brug af NodeJs og Express **anbefaler vi at bruge Mysql teknologien.**

Præsentation af data (Presentation)

For en gennemgang af applikationen se linket (https://github.com/benjaco-edu/db-guttenburg/blob/master/Artefakt%20Applikationens%20interface.pdf)

Efterord:

Ud over den faktuelle konklusion vi har gjort os om de 2 databaser ovenfor, vil vi godt afslutningsvis benytte lejligheden til at nævne, at der har været bred enighed, i gruppen, om at dette har på mange måder været en god opgave.
For det første, og ganske simpelt, har det været sjovt, at arbejde 2 meget forskellige typer af databaser (relation vs dokument). For det andet har det også givet gode erfaringer at have mulighed for at mærke forskellen mellem en optimeret og ikke optimeret query når den bliver eksekveret som en del af en applikation - *hang-time* på mere end et par sekunder bliver meget åbenbart når man sidder med en applikation, hvor man venter på et svar. Også fra et design perspektiv, har det været interessant at bygge en hel *stack* op (selvfølgelig med fokus på databasen) og få en fornemmelse af hvordan hvert element er en vigtig komponent i det samlede hele.