

Dokumentbaseret database

Implementering og optimering af en
dokumentbaseret database

Jonas Vittrup Biegel

01/01-1980

Semesterprojekt



PROFESSIONSHØJSKOLEN



Afdeling

Professionshøjskolen UCN

www.ucn.dk

Titel:

Dokumentbaseret database

Tema:

Semesterprojekt

Projektperiode:

Efterårssemstret

Projektgruppe:

Gruppe 9

Deltager:

Jonas Vittrup Biegel

Vejleder:

Brian Hvarregaard

Oplagstal: 1

Sidetal: 0.9 sider af 2083 anslag

Afleveringsdato:

01/01-1980

Resumé

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magnam aliquam quaerat voluptatem. Ut enim aequale doleamus animo, cum corpore dolemus, fieri tamen permagna accessio potest, si aliquod aeternum et infinitum impendere malum nobis opinemur. Quod idem licet transferre in voluptatem, ut postea variari voluptas distinguere possit, augeri amplificari non possit. At etiam Athenis, ut e patre audiebam facere et urbane Stoicos irridere, statua est in quo a nobis philosophia defensa et collaudata est, cum id, quod maxime placeat, facere possimus, omnis voluptas assumenda est, omnis dolor repellendus. Temporibus autem quibusdam et.

Indhold

1. Indledning	1
2. Foranalyse	2
2.1. Database typer	2
2.1.1. Relationel database	2
2.1.2. Ikke-relationel database	2
3. Problemformulering	3
4. Metode(?)	4
5. Analyse	5
5.1. Database struktur	5
5.1.1. Binary Search Tree	5
5.1.2. Self-balanced Binary Search Tree	6
5.2. Læsning fra disk	6
5.2.1. Heltal i bytes	7
5.2.2. Struktur af en page på disken	7
5.3. Programmering	8
5.3.1. Memory Leak	8
5.3.2. Use-After-Free	9
5.3.3. Buffer Overflow	10
5.3.4. Data Race	11
5.3.5. Noter	12
6. Design	13
7. Implementering	14
8. Test	15
9. Overdragelse (?)	16
10. Konklusion	17
Bibliografi	18

1. Indledning

Indlæning til rapporten

- databaser, datastrukturer etc
- optimering
- Memorysikkerhed

2. Foranalyse

En database er brugt næsten alle steder ude i den rigtige verden i næsten alt software. De bliver brugt til at opbevare en masse data, både informationer om f.eks. kunder men også relevante informationer om de systemer som programmerne holder i gang. Der er mange store database systemer som f.eks. PostgreSQL, SQLite og Microsoft SQL Server. Disse tre har det til fælles at de er relationelle databaser. Der findes dog også ikke-relationelle databaser, såsom MongoDB.

Disse databaseformer vil der nu analyseres nærmere.

2.1. Database typer

Der er to primære typer af database: Relationel database og en ikke-relationel database, også kendt som en NoSQL database (Not only SQL). Disse to har hver deres fordele og ulemper som der vil beskrives nærmere nu.

2.1.1. Relationel database

Relationelle databaser bruger primært det standard sprog for relationelle databaser, SQL. SQL er sprog der blev opfundet i løbet af 1970'erne til at interagere med databaser der indeholdte strukturerede data opbygget på relationel algebra. De bliver brugt i databaser med en tabelstruktur hvor alt data er gemt under et kolonnenavn. Dette gør det nemt at styre hvilke data hører til hvor. [1]

En stor fordel ved relationelle databaser er også at de opfylder ACID. ACID er et sæt af egenskaber som mange relationelle databaser opfylder under deres transaktioner. Dette gør at data kan redigeres og flyttes rundt uden anomalier som ender med at skabe forkert data. Derfor er disse databaser brugt meget i kommercielle programmer hvor der er brug for personstyring etc. [2]

2.1.2. Ikke-relationel database

Ikke-relationelle databaser, modsat relationelle databaser, er ikke tvunget til at være opsat i en tabelstruktur. Disse er databaser der typisk skal holde på meget data som man enten er usikker på størrelsen af, eller som er i en datastruktur som kan ændre hurtigt på sigt. De er også brugt steder hvor det er vigtigt at man kan læse og skrive data i så hurtig en hastighed som muligt. [3]

Et eksempel på noget software som bruger en ikke-relationel database er ting som Facebook og Twitter. Disse bruger databasen til at holde styr på ting som brugergenereret indhold, kommentarer og interaktioner. Disse databaser skal kunne skalere meget hurtigt og holde på mange forskellige typer af data, hvilket er derfor de bliver brugt. [3]

3. Problemformulering

Der kan ud fra foranalysen og kravsspecifikationen udarbejdes følgende problemformulering:

Hvordan kan en dokumentbaseret database udvikles og implementeres med mindre risiko for memory fejl og optimeres mest muligt?

Følgende underspørgsmål kan nu opstilles til problemformuleringen:

- Hvilken datastruktur er bedst til en database?
- Hvordan optimere man læsning og skrivning til databasen?
- Hvordan lagres ens data bedst?
- Hvordan sikrer man sig mod memory sårbarheder?

4. Metode(?)

- Iterativt
- Benchmarking af de forskellige implementeringer
- Først en MVP, derefter optimering herfra

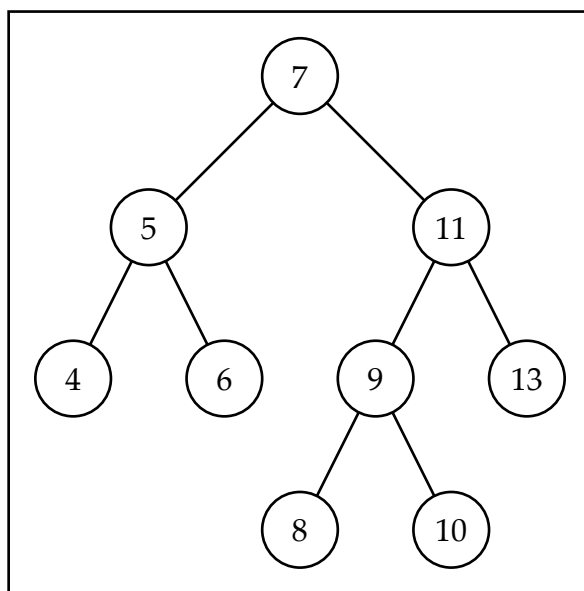
5. Analyse

5.1. Database struktur

At gemme, læse og skrive så store mængder data kan være svært og kræver en god datastruktur. Disse strukturer er typisk baseret på en binær træstruktur. Det normale binære søgtræ har dog den ulempe at den ikke er særlig skalerbar, da den i længden kan blive langsom i forhold til andre træstrukturer. Der er derfor blevet udviklet andre former for træstrukturer som skalere bedre i forhold til læse- og skrivehastighed. Det normale binære søgetræ vil nu analyseres.

5.1.1. Binary Search Tree

På Figur 5.1 kan et binært søgetræ ses.



Figur 5.1: Binært søgetræ med værdierne 4, 5, 6, 7, 8, 9, 10, 11 og 13

Det binære søgetræ er en træstruktur hvor alle keys er større end deres venstre undertræ og mindre end deres højre undertræ. Dette gør at den har relativt hurtige insert, update og removal hastigheder end f.eks. en singly-linked list. Tidskompleksiteten af et binært søgetræ kan ses på Tabel 5.1.

Tabel 5.1: Tidskompleksiteten af et BST i Big O notation, hvor n er antallet af noder i træet og h er højden af træet

Operation	Average	Worst Case
Search	$O(h)$	$O(n)$
Insert	$O(h)$	$O(n)$
Delete	$O(h)$	$O(n)$

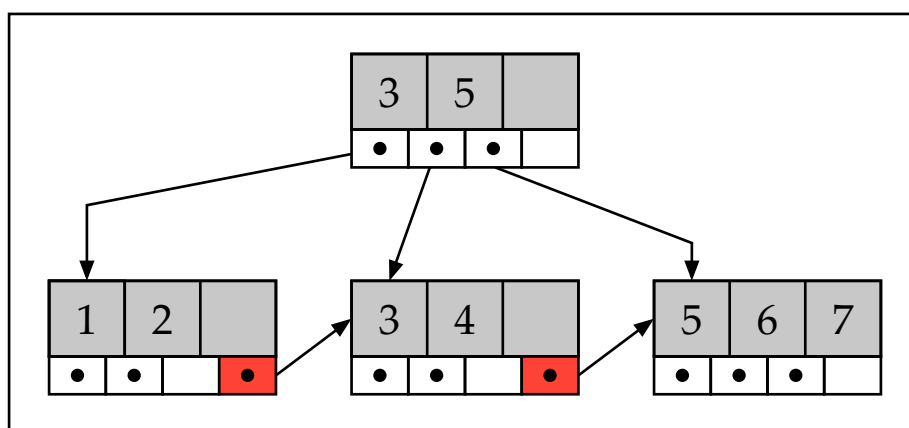
Det kan ses i ovenstående tabel at den gennemsnitlige tidskompleksitet af et binært søgetræ er $O(h)$, altså højden af træet. Dette er hurtigere end en usorteret liste, hvilket gør at det er bedre at køre operationer mod. Dog har det binære søgetræ den ulempe at der er mulighed for at tidskompleksiteten kan nå $O(n)$ hvis det ikke bliver balanceret. Dette skyldes at et ubalanceret binært søgetræ kan være "skævt", hvilket vil sige at det enten kun går til venstre eller højre.

På grund af det normale binære søgetræs dårlige tidskompleksiteter er derfor fundet bedre metoder at lave træstrukturer på kaldet selvbalancerede træer. Disse vil nu analyseres.

5.1.2. Self-balanced Binary Search Tree

Kig i bogen for det her

På Figur 5.2 Kan et B+ Tree ses.



Figur 5.2: Et B+ Tree med værdierne fra 1 til 7

Jeg finder information om B+ Tree fra [4].

5.2. Læsning fra disk

For at læse og skrive data fra og til disken, skal det data først ligge i hukommelsen af computeren. Hukommelsen læser disken i én "page" ad gangen, hvilket er 4096 bytes på

de fleste styresystemer [5] (**RET TIL [4]**). Dette betyder at ens kode skal optimeres ved kun at læse de data som der er relevante for den operation der bliver kørt mod databasen ind i hukommelsen, én page ad gangen.

B+ træstrukturen kan gemmes i disken, hvor ordren af træet kan bestemmes af størrelsen af en page, hvilket er 4kB. For at gøre dette er det vigtigt at forstå bytestrukturen af forskellige heltalsrepræsentationer.

5.2.1. Heltal i bytes

En byte er opbygget af 8 bits, og derfor vil disse beskrives først.

En bit er det mindste stykke data computerhukommelsen indeholder. Dette er et stykke info som kun indeholder en ud af to forskellige stadier, sandt eller falsk, beskrevet som 0 eller 1. Dette er træk fra det binære talsystem, som kan bruges til at beskrive alle tal. Se Figur 5.3 for én bitstruktur af et heltal.

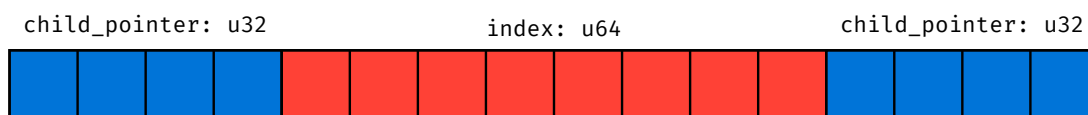
1	1	1	0	0	1	1	0
---	---	---	---	---	---	---	---

Figur 5.3: Tallet 230 skrevet i det binære talsystem, repræsenteret i 8 bits

Det er dog meget ineffektivt konstant at læse og skrive rene bits, da det hurtigt bliver store dele data. Derfor bliver de skrevet ind i bytes, som er en hexadecimal beskrivelse for 8 bits som nemmere kan læses og rykkes rundt. Maksværdien af 8 bits er 11111111, hvilket også er maksværdien af to hexadecimale tal, skrevet som FF. Derfor kan en byte beskrives som to hexadecimale tal der beskriver 8 bits. Eksemplet i Figur 5.3 kan skrives som E6 i bytes. Disse er også usigneret bytes, hvilket vil sige at de kun er positive heltal. Da det kun er 8 bits der beskriver værdien er det af typen u8, altså et 8 bit heltal.

5.2.2. Struktur af en page på disken

Der blev tidligere nævnt af B+ træet kan gemmes på disken, hvor hver node fylder én page. Denne page skal dog holde forskellige informationer, såsom indeks, børn et cetera. En mulig måde at holde denne information kan ses på Figur 5.4.



Figur 5.4: En mulig struktur af én page på disken, der viser en node med 2 børn og ét indeks. Pointersne bruger et ikke-signeret 32 bit heltal, mens indekset bruger et ikke-signeret 64 bit heltal.

Som vist i Figur 5.2 har et B+ træ flere nøgler og børn. Det kan ses i ovenstående figur at strukturen her har ét indeks og to børn. Dette kan så skaleres op til flere indeks og børn, og så kan et helt B+ træ gemmes på en disk.

5.3. Programmering

Mange databaser er i dag skrevet i programmeringssprogene C og C++. Disse sprog er "low-level", forstået som at man arbejder tæt med hardwaren af computeren. Dette er f.eks. ved at man selv skal allokere hukommelse dynamisk. Dette gør også at ting kan optimeres rigtig meget, da sprogene ikke selv bruger en garbage-collector til at sørge for at der ikke er nogle memory fejl, og ikke håndtere det i runtime.

De mest hyppige memory bugs vil nu beskrives, og nogle kodeeksempler der udløser fejlen vil vises. Kodeeksemplerne vil blive skrevet i C.

5.3.1. Memory Leak

Et memory leak opstår når der bliver allokeret memory uden at det bagefter bliver frigivet. Dette kan gøres ved at lave en pointer og så give den en værdi. Dette vil medføre at det memory er optaget indtil slutningen af programmet.

Koden i Liste 5.1 viser et meget overdrevet eksempel på et memory leak.

```
1 int main() {
2     while(true) {
3         leak_memory(100);
4     }
5
6     return 0;
7 }
8
9 void leak_memory(int i) {
10     int *ptr = malloc(sizeof(int));
11     *ptr = i;
12     return;
13 }
```

Liste 5.1: En funktion i C der allokerer memory kontinuerligt

Hvis man er meget uopmærksom på sine pointers og ikke holder styr på at frigive dem efter de er brugt, ender det ofte i et memory leak. Disse kan ende med at være dyre, da de vil have en virkning på ydeevne, og i værste tilfælde vil crashe programmet.

5.3.2. Use-After-Free

```
1 int main() {
2     int *ptr = malloc(sizeof(int));
3     *ptr = 10;
4     free(ptr);
5     do_something(ptr);
6     return 0;
7 }
8
9 void do_something(int *ptr) {
10     // does something with the pointer ...
11 }
```

Liste 5.2: Et program der bruger memory efter det er frigivet

5.3.3. Buffer Overflow

```
1 #include <stdio.h>
2
3 int main() {
4     char buf[64] // create a buffer that holds 64 characters
5     gets(buf); // get input from user
6
7     return 0;
8 }
```

Liste 5.3: Et program der kan lave et buffer overflow

5.3.4. Data Race

```
1 #include <pthread.h>
2
3 int main() {
4     // define 2 threads
5     pthread_t thread1;
6     pthread_t thread2;
7
8     int shared_value = 0;
9
10    // make the threads do some operation on a shared value
11    pthread_create(&thread1, NULL, foo, &shared_value);
12    pthread_create(&thread2, NULL, bar, &shared_value);
13
14    // wait for threads to finish
15    pthread_join(thread1, NULL);
16    pthread_join(thread2, NULL);
17
18    return 0;
19 }
20
21 void* foo(void* arg) {
22     // read and write continually to and from arg ...
23 }
24
25 void* bar(void* arg) {
26     // read and write something else continually to and from arg ...
27 }
```

Liste 5.4: Et program der viser et data race

5.3.5. Noter

- Databaser
- Problemer i databaser (skrevet i C, C++, sikkerhed i memory)
 - SQL Server (C og C++)
 - PostgreSQL (C og C++)
 - MySQL (C og C++),
 - MongoDB (C, C++, JavaScript og Python(?))
 - SQLite (C)
- Strukturen af databaser måske?

```
1 fn main() {  
2     println!("lol");  
3  
4     for i in 0..100 {  
5         println!("lol {i}")  
6     }  
7 }
```

6. Design

- Der bliver brugt et B+ træ
- Vis filstruktur, altså header, nodestruktur og leafstruktur i pages
- Rust bliver brugt da det forsøger at løse de memory problemer som C og C++ har

7. Implementering

- Vis implementering (kode)
- Vis noget fra modules måske, vis de vigtige dele af koden
 - Træstruktur
 - Skriv til filen i bytes, læs fra filen i bytes ned af strukturen med pointers til index i filen
- Måske et eksempel program der bruger databasen? Kunne være en todo list etc.
- Hvis jeg når så langt, så vis implementeringen af API'en og hvordan den tager generiske parametre, opretter tabeller og indsætter data etc.

8. Test

- Vis tests
 - Unit tests, integration tests, benchmarks

9. Overdragelse (?)

- Ved ikke hvad det her er

10. Konklusion

Konklusion af rapporten

Bibliografi

- [1] Wikipedia, "SQL". [Online]. Tilgængelig hos: <https://en.wikipedia.org/wiki/SQL>
- [2] Wikipedia, "ACID". [Online]. Tilgængelig hos: <https://en.wikipedia.org/wiki/ACID>
- [3] What Is A Database?, "What Is a Non-Relational Database? Understanding Its Key Features". [Online]. Tilgængelig hos: <https://www.whatisdatabase.com/what-is-a-non-relational-database-understanding-its-key-features>
- [4] A. Petrov, *Database Internals: A Deep Dive into How Distributed Data Systems Work*. O'Reilly, 2019.
- [5] Wikipedia, "Page (computer memory)". [Online]. Tilgængelig hos: https://en.wikipedia.org/wiki/Page_%28computer_memory%29