Universitetet i Oslo
Institutt for informatikk

Rindex

RAM-basert indekshåndtering — en effektivitetsstudie

Tomas Are Haavet

Masteroppgave

1. februar 2005

Forord

Jeg ønsker å rette en stor takk til alle som har hjulpet meg underveis i arbeidet med masteroppgaven. En spesiell takk til amanuensis Ragnar Normann ved Institutt for Informatikk, som har vært en meget god og kompetent veileder. I tillegg fortjener Kjell-Magne Øierud en stor takk for godt samarbeid med fellesdelen av oppgaven. Tusen takk til Ingvild Heggen Støa, Jakub Bogumil Warszawski og Gunnfrid Øierud, samt familie, venner og kollegaer, for korrekturlesning og god støtte.

Oslo, 1. februar 2005
Tomas Are Haavet

Innhold

Introduksjon ix

I Rindex — beskrivelse av plattformen 1

1 Innledning 3
- 1.1 Plattformen . . . 4
  - 1.1.1 Begrensninger . . . 4
- 1.2 Valg av programmeringsspråk . . . 5

2 Librindex 7

3 Programmet 9
- 3.1 Brukergrensesnittet . . . 10
- 3.2 Kommunikasjonsprotokollen . . . 10

4 Indeksene og operasjonene 13
- 4.1 Indeksene . . . 13
- 4.2 Inverterte indekser . . . 15
- 4.3 Generering av indeksene . . . 17
- 4.4 Operasjonene . . . 17
  - 4.4.1 Seleksjon . . . 18
  - 4.4.2 Union . . . 18
  - 4.4.3 Equi-join . . . 19
  - 4.4.4 Projeksjon . . . 20
  - 4.4.5 Substitusjon . . . 20
  - 4.4.6 Indekssortering . . . 20

5 Globale datastrukturer 21
- 5.1 Trær . . . 21
  - 5.1.1 Syntakstre . . . 21
  - 5.1.2 Spørretre . . . 22
  - 5.1.3 Plantre . . . 22
- 5.2 Databaseskjemaet . . . 22

5.3 Støtte for ulike DBMSer 23

6 Statisk analyse **25**
6.1 Skanner 26
6.2 Parser 26
6.3 Semantikksjekker 30

7 Optimalisering og plangenerering **33**
7.1 Spørretreet 33
7.2 Optimalisering 34
7.2.1 Implementasjonen 35
7.3 Plangenerering 37
7.3.1 Implementasjonen 37

8 Planeksekvering **41**

9 Mangler og forbedringer **45**

II Substitusjonsoperatoren **47**

10 Problemstilling **49**
10.1 Substitusjonsoperatoren 49
10.2 Problemstillingene 50

11 Analyse av substitusjonsalgoritmen **51**
11.1 Substitusjonsalgoritme 1 51
11.1.1 Datastrukturen 52
11.1.2 Algoritmen 54
11.1.3 Hash-funksjonen, versjon 1 57
11.1.4 Hash-funksjonen, versjon 2 59
11.1.5 Sammenligning av hash-funksjonene 60
11.2 Substitusjonsalgoritme 2 61
11.2.1 Datastrukturen 61
11.2.2 Algoritmen 63
11.3 Substitusjonsalgoritme 3 64
11.3.1 Datastrukturen 64
11.3.2 Algoritmen 64
11.4 Substitusjonsalgoritme 4 og 5 66
11.4.1 Datastrukturene 67
11.4.2 Algoritmene 69
11.5 Sammenligning 71
11.5.1 Teoretisk sammenligning 71
11.5.2 Empirisk sammenligning 73
11.6 Konklusjon 88

12 Plassering av substitusjonsoperatoren **91**
12.1 Eksisterende algoritme 91
12.2 Forbedret algoritme 92
12.2.1 Beskrivelse av algoritmen 93

13 Avslutning **101**
13.1 Oppsummering 101
13.1.1 Problemstilling 1 101
13.1.2 Problemstilling 2 102
13.1.3 Styrker og svakheter 102
13.2 Videre arbeid 103

Figurer

2.1 Skjematisk framstilling av modulene i `librindex` 8

3.1 Kontrollflyten i Rindex 11

5.1 Klassehierarki for de ulike frontendene mot DBMSene . . . 24

6.1 Syntakstreet 30

7.1 Et generelt spørretre. 34
7.2 Spørretreet 34
7.3 Det optimaliserte spørretreet 36
7.4 Den genererte planen 39

8.1 Beregning av aritmetiske uttrykk 42

11.1 Datastrukturer for substitusjonsalgoritme 1 53
11.2 Datastrukturer for substitusjonsalgoritme 2 62
11.3 Datastrukturer for substitusjonsalgoritme 3 65
11.4 Datastrukturer for substitusjonsalgoritmene 4 og 5 68
11.5 Fire testspørringer 77
11.6 Plott for spørring 1 79
11.7 Plott med logaritmisk skala for spørring 1 79
11.8 Plott for spørring 2 80
11.9 Plott med logaritmisk skala for spørring 2 80
11.10 Plott for spørring 3 81
11.11 Plott med logaritmisk skala for spørring 3 81
11.12 Plott for spørring 4 82
11.13 Plott med logaritmisk skala for spørring 4 82
11.14 Minnebruken 85
11.15 Utsnitt av minnebruken 85
11.16 Tidsforbruk ved generering av indeksene 87
11.17 Utsnitt av tidsforbruket ved generering av indeksene 87

12.1 Effekt ved bruk av ombyttingsreglene 92
12.2 Eksempelspørringen 95

Introduksjon

Denne masteroppgaven er siste del av mastergradsstudiet i informatikk, og er utført ved forskningsgruppen "Objektorientering, modellering og språk" ved Institutt for Informatikk, Universitetet i Oslo. Oppgaven er veiledet av amanuensis Ragnar Normann ved Institutt for Informatikk. Denne rapporten er todelt. Den første delen er gjort i samarbeid med Kjell-Magne Øierud, og omhandler designen og implementasjonen av Rindex-plattformen. Den andre delen er gjort av meg, og den drøfter problemstillinger rundt effektiviteten til en spesiell operator i Rindex, kalt substitusjonsoperatoren.

Siden oppgaven baseres på temaet 'Indekshåndtering i databasehåndteringssystemer', vil det være en fordel å ha noe kunnskap om emnet, for eksempel gjennom kurset INF212/INF3100/INF4100, for å få fullt utbytte av denne rapporten.

Denne rapporten, og koden som ble produsert under arbeidet med oppgaven, er gjort tilgjengelig på en ressursside for prosjektet:
`http://www.ifi.uio.no/forskning/grupper/oms/rindex`

Språk og notasjoner

Rapporten er skrevet på norsk, men siden informatikk er et fag som er preget av mange engelske faguttrykk, har jeg valgt å bruke dem uoversatt der jeg ikke har funnet noe tilsvarende godt og velkjent norsk uttrykk. Dette gjelder for eksempel ordene *array* og *join*. Koden for implementasjonen av Rindex-plattformen er derimot skrevet og kommentert på engelsk.

Alle navn på klasser, objekter, programmer, verktøy, biblioteker og nøkkelord i SQL er uthevet med maskinskrift, mens definisjoner er skrevet i kursiv skrift. Videre avsluttes eksemplene med symbolet $\triangle$. Manualsider i UNIX refereres til med seksjonshenvisning i parentes, for eksempel *time*(1).

Del I

Rindex — beskrivelse av plattformen

Kapittel 1

Innledning

Vi ønsker i dette dokumentet å beskrive hvordan og hvorfor Rindex[1] ble til, men kanskje viktigst, hvordan den fungerer. Det vil ved flere anledninger bli referert til et dokument vi skrev under planleggingen av Rindex. Dette var ment å skulle beskrive designen av programmet. Der skrev vi følgende om hensikten til plattformen:

> Formålet med dette prosjektet er å lage en plattform (Rindex) som skal understøtte vårt og andres videre arbeid med hovedoppgaver innen temaet 'Indekshåndtering i databasehåndteringssystemer'.
>
> Vi skal undersøke hvordan man kan utnytte at maskiner nå blir utstyrt med mye mer RAM enn hva som var vanlig for få år siden. Spesielt skal vi se på hvordan dette kan utnyttes ved å legge hele indekser permanent i minne, og problemstillinger rundt dette.

Tradisjonelt sett er et databasehåndteringssystem (DBMS) designet med tanke på at RAM er en begrenset ressurs. En slik design er basert på intensiv bruk av harddisken, med de følger dette har for ytelsen. Som en løsning på dette har rene primærlager-DBMSer (main memory DBMS) blitt utviklet. I disse databasene ligger alle data og indekser permanent i minnet, og det oppnås derfor en meget god ytelse. Problemet med denne designen er at de egner seg dårlig til virkelig store databaser og til store datatyper (for eksempel filmer). I tillegg vil ikke dataene overleve en eventuell omstart av maskinen eller strømstans, fordi minnet er flyktig (volatile).

Rindex er et forsøk på å ta det beste fra begge verdener, ved beholde ytelsen til main memory DBMSer uten de overnevnte begrensningene.

[1] Rindex er en forkortelse for RAM index.

1.1 Plattformen

Ut fra målene for Rindex-plattformen, vurderte vi følgende fire realiseringsmuligheter:

> 1 Manipulere på kildekoden til et DBMS med åpen kildekode (for eksempel PostgreSQL eller MySQL),
>
> 2 Gå under skallet på en proprietær DBMS (for eksempel Oracle eller SyBase),
>
> 3 Lage en egen DBMS (for eksempel basert på Bercley DB) eller
>
> 4 Bygge et program på toppen av en eksisterende DBMS.
>
> Slik vi ser det, vil de tre første alternativene gi et ganske stort merarbeid innenfor områder som ikke er direkte relevante for oppgaven. For punkt (1) måtte vi ha satt oss inn i mye kildekode, og det ville være uvisst på forhånd hvor mye som måtte skrives om for å tilpasse DBMSen til våre krav. Punkt (2) ville kreve komplisert «reverse engineering» og (3) ville bli alt for omfattende.
>
> Valget falt på det siste alternativet, siden det virket mest realistisk å gjennomføre.

Plattformen kan altså sees på som et overbygg til en eksisterende DBMS. Av praktiske hensyn har vi her valgt Oracle som den underliggende DBMSen, men Rindex er designet for å enkelt kunne utvides med støtte for flere ulike underliggende DBMSer, se kapittel 5.3.

1.1.1 Begrensninger

Selv om den valgte realiseringsmetoden sannsynligvis var den minst krevende løsningen, var det likevel nødvendig å definere klare begrensninger med plattformen tidlig i prosessen. Under planleggingen skrev vi:

> Følgende hovedbegrensninger er valgt ut fra målene med plattformen:
>
> - For det første velger vi å bare implementere støtte for statiske indekser, og dermed ikke støtte *oppdateringer* av dataene. Dette fører til enklere datastrukturer og algoritmer.
> - Den andre begrensningen er delvis en konsekvens av dette: Rindex støtter bare et lite subsett av SQL-standarden, det vil si essensielt bare enkle `SELECT`-spørringer.
> - En tredje begrensning er at Rindex-plattformen ikke gjør noen optimaliseringer av spørretreet.

- For å eliminere en potensielt stor feilkilde, er kommunikasjon over nettverk ikke støttet i Rindex.

I tillegg til disse forhåndsdefinerte begrensningene, måtte vi definere flere mindre begrensninger underveis. De viktigste er oppsummert til slutt, i kapittel 9.

1.2 Valg av programmeringsspråk

I utgangspunktet noterte vi oss følgende argument for valg av programmeringsspråk:

> Når vi skal realisere Rindex, må vi bestemme oss for hvilket språk den skal programmeres i. Vi er avhengig av å kunne manipulere minnet på et meget lavt nivå. Dette taler for å bruke C. Vi ønsker i tillegg å kunne benytte oss av programmeringskonstruksjoner som er litt mer høynivå enn det man har i C (klasser, objekter, templates m.m.). Valget faller derfor på C++.

Det er kun i `IndexManager` at vi er avhengige av å drive med lavnivåprogrammering. I ettertid ser vi at det ville vært lurere å skrive `IndexManager` i ren C, mens resten med fordel kunne vært skrevet i et språk som er på høyere nivå enn hva selv C++ er.[2] Et slikt språk måtte i så fall hatt gode bindinger mot C for lett og effektivt å kunne samspille med C.

[2] C++ mangler blant annet automatisk minnehåndtering.

Kapittel 2

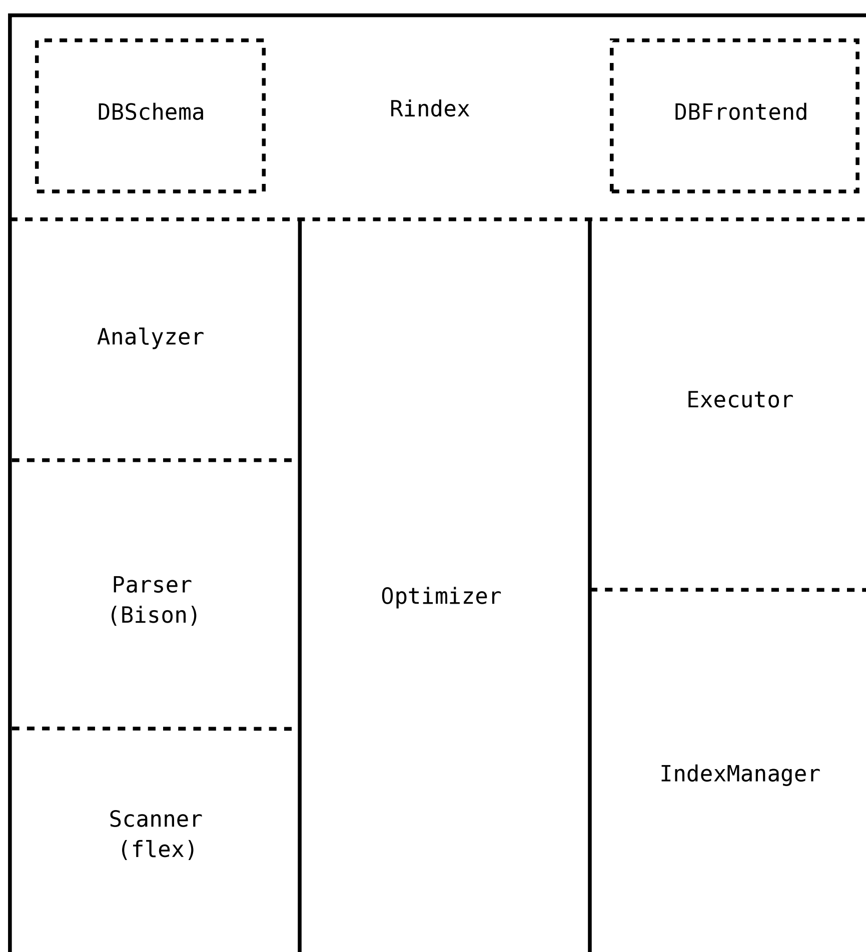
Librindex

Sentralt i Rindex er et programmeringsbibliotek kalt `librindex`. Dette biblioteket utgjør selve ryggmargen i plattformen, og gjør at Rindex blir fleksibel med tanke på forskjellige bruksmønstre. Man kan for eksempel bruke dette biblioteket til å lage et program med tanke på å teste ulike aspekter ved indekshåndtering, eller man kan lage en mer tradisjonell løsning med en kommandotolker. Biblioteket består av flere moduler, men grensesnittet er definert av klassen `Rindex`. All kommunikasjon med `librindex` foregår via metodene definert i denne klassen og de tre tegnstrømmene `in` (for kommandoer), `out` (for resultater) og `err` (for debug- og feilmeldinger).

Figur 2.1 gir en kort oversikt over de ulike komponentene i `librindex`. Den belyser hvordan de forskjellige komponentene samspiller med hverandre, og skal forstås slik:

- To moduler som har en stiplet grense kan kommunisere med hverandre, mens det ikke er noen direkte interaksjon mellom to som bare har heltrukne grenser mellom seg.
- Inne i `Rindex`-modulen er det flere stiplede bokser. Disse angir globalt tilgjengelige moduler.

I figuren er det også angitt en hovedgruppering av modulene. Øverst har vi *kontrollklassen*, `Rindex`, og de globale modulene. Under denne er det tre kolonner. Den første inneholder moduler som bygger opp en intern struktur av spørringen og analyserer denne. Den andre kolonnen har moduler som utfører optimaliseringer og plangenerering. Tilsammen utgjør disse to kolonnene *kompilatoren*. Den siste kolonnen har moduler som brukes i forbindelse med eksekvering.



Figur 2.1: Skjematisk framstilling av modulene i `librindex`

Kapittel 3

Programmet

Rindex kan startes i to forskjellige modi for brukerinteraksjon. Dersom man gir Rindex argumentet `-i`, vil den starte opp i en *interaktiv modus*, hvor den selv lytter på tastaturet. Denne modusen egner seg best til testing og debugging av systemet, siden dette ikke egentlig er noen form for brukergrensesnitt. Den vil derfor ikke bli nærmere beskrevet.

I den andre modusen oppfører Rindex seg som en server. Da lytter ikke programmet lenger på tastaturet, men i stedet på en spesiell pipe-fil. Hvis programmet startes i en slik modus (altså uten bruk av `-i` opsjonen), er det meningen at man benytter en klient som kommandotolker.

Bakgrunnen for å bruke piper for «InterProcess Communication» (IPC), er at det er en løsning som medfører at lite kode er forskjellig i interaktiv- og servermodus. En mer fleksibel løsning vil være å erstatte pipes med sockets.[1] Det ville da vært svært enkelt å modifisere Rindex slik at den kunne brukes via et nettverk, hvis det skulle være ønskelig. Vi valgte den enkleste løsningen, pipes, siden det å støtte nettverk er en funksjon som faller utenfor hensikten med prosjektet.

Programmet, `rindex-bin`, startes ved hjelp av et shellskript som heter `rindex`. Dette skriptet setter opp den nødvendige omgivelsen til plattformen, før det eksekverer selve programmet.

`rindex-bin` linkes opp mot `librindex` og håndterer i tillegg kommandolinjeargumenter og initialisering av terminerings- og signal-håndterere. Disse passer på at temporære filer slettes. For å parse argumentene, bruker vi `argp`[2]. Videre initialiseres `librindex`, og til slutt startes kommandoløkka. Her kalles metoden `query()` i `librindex`, én gang for hvert input (for eksempel en spørring eller en kommando), inntil den returner «false». Dette

[1] For en mer detaljert beskrivelse av ulike metoder for kommunikasjon mellom prosesser, se [16, kapittel 5 og 6].

[2] `argp` finnes bare i GNUs C-bibliotek (glibc) og er derfor ikke standard. Dette betyr at dersom plattformen skal portes, må `argp` byttes ut med for eksempel `getopt` som er definert i POSIX. Men `argp` er mer fleksibelt og enklere å bruke, derfor valgte vi bort portabilitet.

er et signal på at en `quit`-kommando er gitt. Figur 3.1 viser kontrollflyten i plattformen under en spørring.

3.1 Brukergrensesnittet

Rindex er, i dagens form, en SQL-plattform. Det vil si at den tar imot kommandoer i form av SQL-spørringer. I tillegg har vi definert noen andre Rindex-kommandoer: `quit` og `time`.

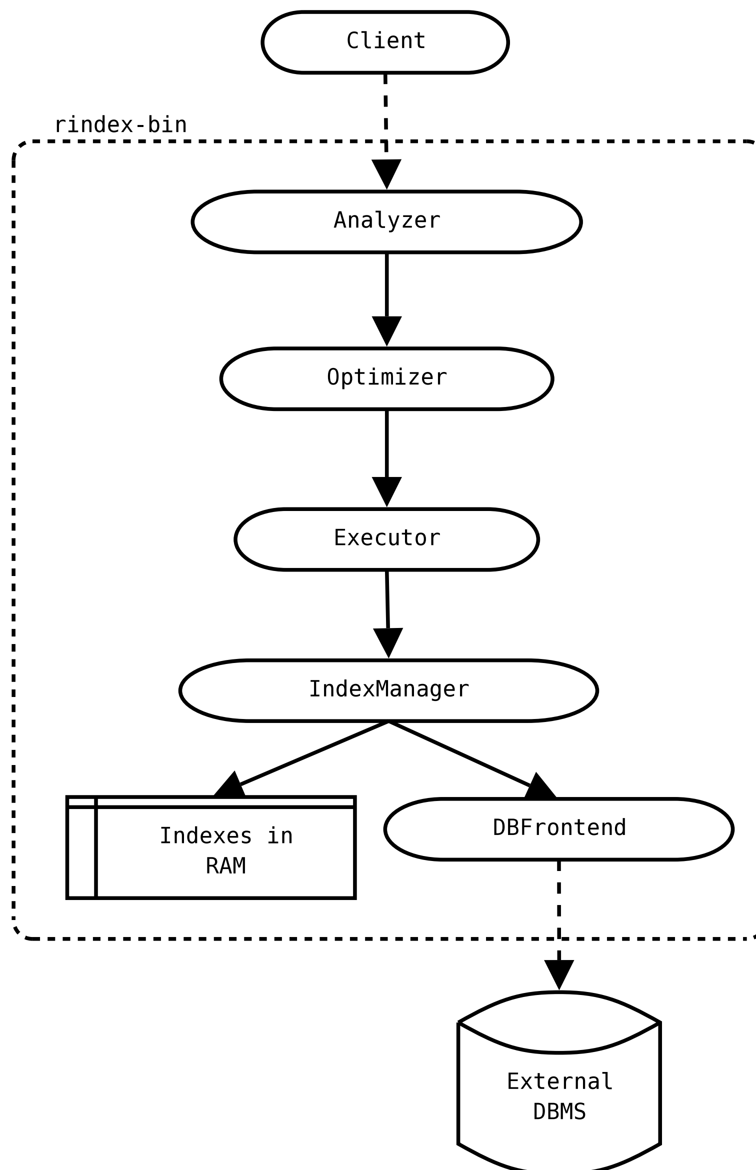
Som kommandotolker har vi laget et perlskript, kalt `client.plx`, som baserer seg på GNUs readline-bibliotek. Readline er et kraftig verktøy for å implementere shell-lignende funksjonalitet, og brukes blant annet i `bash`[3]. Klienten har innebygget kommandoer for å starte og stoppe (ved kommandoene `start` og `stop`) Rindex, og for å avslutte klienten (`quit`, `q` eller `exit`). Kommandoer som ikke gjenkjennes av klienten, blir antatt å være Rindex-kommandoer og sendes videre. En alternativ måte til å lese Rindex-kommandoer fra tastaturet, er å lese dem fra fil ved å gi kommandoen `load <filnavn>` til klienten. Alle kommandoene som er gitt i en slik fil, tolkes som Rindex-kommandoer.

Da vi planla plattformen, var kommandotolkeren en del av Rindex og den skulle sende spørringene direkte til parseren. Dette la vi også opp til i planleggingsfasen, men dette ble imidlertid ikke implementert. Årsakene til at vi gikk bort fra denne designen var flere, men hovedgrunnen var at hver gang Rindex startes, må indeksen bygges opp. Dette tar noe tid, og det er derfor gunstig å ikke starte Rindex oftere enn nødvendig. Den løsningen vi endte opp med løser dette ved at serveren kjøres som en bakgrunnsprosess som klienten kan sende kommandoer til (se avsnitt 3.2), slik at serveren ikke må restartes hver gang. I `rindex-bin` har vi implementert tjenerdelen av en slik løsning.

3.2 Kommunikasjonsprotokollen

Kommunikasjonsprotokollen er en meget enkel *stop and wait*-protokoll. Klienten sender en kommando og venter deretter på at Rindex skal sende en ACK (via en egen dedikert pipe-fil) som et signal på at den er klar for en ny kommando. Vi har ikke designet noen annen form for kommunikasjon fra Rindex til klienten. Dette er en ganske stor begrensning siden det gjør at klienten blant annet ikke kan ha noen kontroll med outputet fra Rindex uten å ty til «ad-hoc» løsninger.

[3] `bash` er blant de vanligste UNIX-shellene.



Figur 3.1: Kontrollflyten i Rindex

Kapittel 4

Indeksene og de tilhørende operasjonene

Den viktigste datastrukturen i Rindex er indeksene, i og med at det er på indeksene at alle beregninger av resultatet utføres. I de følgende avsnittene vil vi derfor gå mer detaljert gjennom hvordan disse er oppbygd. Siden plattformen skal understøtte forskning på indekshåndtering, er det essensielt at det er enkelt å eksperimentere med forskjellige typer indekser. Vi har derfor definert et grensesnitt i `IndexManager` hvor alle indeksoperasjonene er definert. Dette gjør at moduler som benytter grensesnittet ikke trenger å forholde seg til implementasjonen av indeksene.

4.1 Indeksene

Å avgjøre hvordan indeksene skulle se ut var det vi brukte mest tid på å planlegge. Viktige kriterier var at:

1 De må være enkle.

2 Operasjoner på og med dem må være effektive med tanke på CPU-tid.

3 De må støtte operasjoner som er hensiktsmessige i forhold til de typer spørringer som muliggjøres av grammatikken.

Tradisjonelt sett er B^+-trær en meget vanlig datastruktur [5, avsnitt 13.3].[1] Siden Rindex ikke støtter dynamisk aktivitet[2] i databasen, medfører dette at

[1] Vi ble også oppmerksomme på at CSB+-trær («Cache-Sensitive B+-Trees») sannsynligvis er en bedre indeksstruktur enn rene B+-trær, siden de er konsturet for å kun ligge i minnet og for å utnytte cachen bedre.

[2] Dynamisk aktivitet forekommer hvis man har spørringer som endrer innholdet i databasen. Det vil si «Data-Manipulation Language» (DML)-kommandoene `INSERT`, `UPDATE` og `DELETE` og «Data-Definition Language» (DDL)-kommandoer som `ALTER`, `CREATE` og `DROP`.

B^+-treets egenskaper er overflødige. Dette betyr at vi klarer oss med en mye enklere struktur for indeksene. En slik struktur står blant annet beskrevet i [5, avsnitt 13.1]:

> One of the simplest index types relies on the file being sorted on the attribute(s) of the index. Such a file is called a *sequential file*.

På grunn av at Rindex er bygd på toppen av en annen DBMS, må indeksene nødvendigvis være *sekundære* [5, avsnitt 13.2].

> [...] a secondary index is distinguished from a primary index in that a secondary index does not determine the placement of records in the datafile. Rather the secondary index tells us the current locations of records. [...] secondary indexes is always dense.

Den nåværende lokasjonen finner vi ved å bruke primærnøkkelen til attributtet som «peker». Denne kan brukes for å hente ut verdier vi ikke har indeksert fra databasen. Primærnøklene brukes også aktivt i projeksjoner og substitusjoner (se senere avsnitt).

Da vi planla Rindex skrev vi følgende:

> Vi tenker oss at vi indekserer tekstattributter med inntil 256 tegn. En ulempe med sekvensielle filer blir da at de egner seg dårlig til sammenligning av nøkler basert på slike lange tekststrenger. Det blir tregt å teste på likhet fordi man, i verste fall, må gjøre 256 tester per tuppel. En annen ulempe er at indeksene blir store når man lagrer hele verdien til attributtet. En løsning på disse problemene er å bruke en *hashverdi* (for eksempel `md5`[3]) som nøkkel i indeksene i stedet for hele tekststrengen. En hashverdi er typisk mye kortere enn tekststrengen (for eksempel 4 byte) og har konstant lengde. Dette gir imidlertid noen nye problemer:
>
> - Kollisjoner gjør at man ikke kan være helt sikker på om man har likhet. En kollisjon oppstår når to forskjellige tekststrenger får samme hashverdi.
> - Intervall-spørringer blir ineffektive siden hasverdiene ikke bevarer ordningen til de opprinnelige strengene.
>
> Kollisjoner håndteres ved å lage en kollisjonsliste for hver hashverdi. Disse listene inneholder arrayoffseten til indeksen på de kolliderende attributtene. Når man gjør en seleksjon på en hash hvor det forekommer kollisjoner, kan man få et resultatsett som inneholder gale verdier. Man må da gå igjennom sluttresultatet og fjerne tupler som ikke skulle ha vært der. Eventuelt kan dette gjøres underveis.

[3] `md5` er en enveis hash-algoritme.

For å raskere kunne avgjøre ulikhet, kan man lagre en ekstra hashverdi. Denne verdien må beregnes utfra en annen hashalgoritme enn den opprinnelige. Sannsynligheten for at to ulike tekster skal gi samme hashverdi ved bruk av begge algoritmene er vesentlig mindre enn sannsynligheten for at to tekster skal gi samme hashverdi for bare én av disse. Det er derimot ikke sikkert at dette totalt sett vil være en optimalisering. Dette er et interessant spørsmål hvor svaret antakeligvis vil være avhengig av «Universe of Discourse» (UoD).

Et annet argument for å hashe tekstattributtene, er at man kan bruke hashverdien modulo tabellstørrelsen for å finne den tilhørende posisjonen i hashtabellen. Man trenger dermed ikke å gjøre binærsøk på tabellen, men kan i stedet gjøre direkte oppslag i $\mathcal{O}(1)$ tid. Vi må da sannsynligvis regne med flere kollisjoner (blant annet avhengig av hashfunksjonen og tabellstørrelsen), men det håndteres av kollisjonslisten. Et problem er attributter med et lite domene av strenger, for eksempel kan en person ha et rolleattributt som tar verdier fra domenet skuespiller, regissør, produsent, manusforfatter. Slike attributter vil ofte gi mange kollisjoner i hashtabellen, og man må da gjøre et binærsøk på den tilhørende kollisjonslisten.

Intervall-spørringer er ønskelige, også på tekst. Det er derfor ikke akseptabelt at disse ikke støttes. Vi tvinges derfor til å lagre attributtverdiene også.

Fremdeles er det et problem at vi må lagre hele attributtverdien også for lange nøkler. For å hjelpe på dette kan man prøve å komprimere strengene, men dette fører til en ekstra kostnad med å dekomprimere. Dessuten ville man ikke få noen effekt av å komprimere tekststrenger som ikke er lengre enn 256 tegn (de kan faktisk bli lengre på grunn av metadata som lagres av komprimeringsalgoritmen).

Vi har ikke effektivisert strenger slik vi beskriver her, noe som skyldes at vi ikke prioriterte å bruke tid på dette. Dette ville dessuten ført til en mer komplisert indekshåndtering.

Indeksstrukturen vi har endt opp med, er altså sorterte sekvensielle filer hvor hver indekspost består av et par med attributtverdi og primærnøkkel.

4.2 Inverterte indekser

For at blant annet projeksjoner og substitusjoner skal kunne utføres på en effektiv måte, har vi også lagd inverterte indekser. En invertert indeks er en indeks hvor primærnøklene og attributtverdiene har byttet rolle. Disse indeksene vil kun bli brukt til direkte oppslag, og dette favoriserer en datastruktur

som hashtabeller [8]:

> A third possibility is to avoid all this rummaging around by doing some arithmetical calculation on K, computing a function $f(K)$ that is the location of K and the associated data in the table.

Håndtering av kollisjoner gjøres ved å bruke en teknikk som kalles lenking. Hashtabellen blir da implementert som en array med kollisjonslister. Et alternativ kunne vært åpen adressering, men vi valgte lenking siden dette er enklest å implementere korrekt. Elementene i kollisjonslistene er strukter bestående av pekere til primærnøkkelen og attributtverdien, samt neste element i listen. Hashnøkkelen beregnes her på bakgrunn av primærnøkkelen.

Når det gjelder valg av hashfunksjon, har Knuth satt opp følgende kriterier [8]:

> A good hash function should satisfy two requirements:
>
> a) Its computation should be very fast.
>
> b) It should minimize collisions.

Siden en primærnøkkel er en mengde med attributtverdier, er det nødvendig med et ekstra krav:

c) Resultatet av å applisere hashfunksjonen på primærnøkkelen må være uavhengig av verdienes rekkefølge i primærnøkkellisten.

Vi oppfyller krav a) og samtidig tilleggskravet c) ved å benytte følgende formel for å beregne hashverdien:

$$\left(\sum_{i=1}^{n} t(key[i])\right) \bmod N$$

N er størrelsen på hashtabellen og n er antall elementer i primærnøkkelen. Funksjonen t gir verdien som ligger i $key[i]$ tolket som et positivt heltall med samme antall bit som ordlengden til den underliggende arkitekturen (altså en `unsigned int` i C-sjargong). Siden *key* alltid er en array med pekere, vil det være minneadressene som summeres.

Hvorvidt vi tilfredstiller b) har vi ikke rukket å undersøke grundig nok, men noen små tester indikerer at den muligens holder mål. Noe som bygger opp under dette, er at før vi implementerte hashing, var de inverterte indeksene usorterte sekvensielle filer. Etter å ha reimplementert dem med hashing, opplevde vi en ytelsesforbedring for substitusjoner på ca. elleve ganger.

4.3 Generering av indeksene

I databasen genereres det indekser på alle attributter som har en datatype som plattformen støtter. Dette gjelder 32- og 64-bits heltall, 32- og 64-bits flyttall og strenger. Når man starter Rindexgenereres indeksene og legges permanent i minnet. Plattformen må derfor midlertidig hente ut alle relasjonene med attributter som vi skal lage indekser på. Et sentralt spørsmål for om denne strategien er brukbar i praksis, blir hvor lang tid denne genereringsprosessen tar. Ved å bygge indekser for filmdatabasen[4] på viisi.ifi.uio.no[5], rapporterte *time*(1):

```
18.30 user, 5.43 system, 1:24.31 elapsed
```

Av dette ser vi at mesteparten av tiden går med på å hente dataene fra Oracle, inkludert nettverkskommunikasjon. Tiden Rindex selv bruker (user + system) er ganske akseptabel (ca. 24 sekunder).

Et annet alternativ ville vært å generere indeksene underveis. En slik indeks må genereres hver gang det blir gitt en spørring med et attributt som ikke allerede er indeksert. Vi vurderte det slik at det er bedre at oppstarten tar tid, enn at systemet får en veldig variabel ytelse.

4.4 Operasjonene

I planleggingsfasen skrev vi lite om selve algoritmene for indeksoperasjonene. Disse ble derfor til underveis. I de følgende avsnittene tar vi for oss operasjonene, én etter én. Først noen definisjoner:

Definisjon 4.1 (Mellomlagring)**.** *En* mellomlagring *er et midlertidig resultat som følger av å applisere en indeksoperasjon på en eller flere indekser. En mellomlagring har nøyaktig samme struktur som en indeks.*

Det at vi ikke skiller mellomlagringer fra indekser, gjør at vi ikke trenger å lage spesialtilfeller avhengig av om det er en indeks eller en mellomlagring som er input til en indeksoperasjon.

Definisjon 4.2 (Indeksoperasjon)**.** *En* indeksoperasjon *gjør en* transformasjon *på en indeks/mellomlagring. I denne kategorien faller* substitusjon *og* sortering*, samt operasjoner som har en analog i relasjonsalgebraen. Ariteten til transformasjonene varierer, men må bestå av minst én indeks/mellomlagring. Resultatet er derimot alltid en (ny) mellomlagring.*

Vi vil ikke skille indekser fra mellomlagringer i beskrivelsene av indeksoperasjonene.

[4]Dette er databasen som er brukt i kurset INF212/INF3100/INF4100, se [12]. Filmdatabasen har 543 557 rader fordelt på 11 tabeller.

[5]viisi er en dual Intel Xeon 2.40 GHz prosessor med 2 GB RAM

4.4.1 Seleksjon

Slik vi har laget plattformen, kan en seleksjon kun bestå av én betingelse for å eksekveres. Dette er ganske uproblematisk siden følgende identiteter gjelder:[6]

Definisjon 4.3. *Oppsplittingsregler for seleksjon:*

- $\sigma_{A \text{ AND } B}(X) = \sigma_A(\sigma_B(X))$
- $\sigma_{A \text{ OR } B}(X) = \sigma_A(X) \cup_S \sigma_B(X)$

Videre sier [5] at oppsplittingsregelen for OR bare gjelder hvis relasjonen seleksjonen utføres på, er en mengde. Våre indekser er ikke mengder, og derfor kan vi ikke bruke regelen direkte. Vi løser dette ved å benytte den modifiserte unionsoperatoren (se avsnitt 4.4.2).

En type seleksjoner vi ikke har implementert, er seleksjoner hvor man sammenligner to attributter med hverandre. En slik seleksjon, sammen med en join, er det samme som theta-join. Vi ønsker å bare støtte equi-joins – derav denne begrensningen. Dette medfører dessuten at seleksjonsalgoritmen blir ganske enkel, og man trenger bare å operere på én indeks/mellomlagring som input.

Vårt seleksjonsbegrep dekker egentlig en familie med nært beslektede algoritmer, én for hver av de mulige sammenligningsoperatorene. Algoritmen innebærer først et binærsøk etter den verdien man sammenligner mot. Binærsøket er modifisert på en slik måte at dersom verdien det søkes etter ikke eksisterer, returneres posisjonen til den verdien som ligger nærmest over eller under i indeksen. Deretter gjøres det lineærsøk for å finne verdiene som skal med, avhengig av hvilken sammenligningsoperator som brukes i seleksjonen.

4.4.2 Union

Unionen er en binær operasjon som oppstår når man bruker OR i spørringen.[7] Dette følger av oppsplittingsregelen (se definisjon 4.3).

En vanlig måte å definere unionen på er gitt i [5]:

> $R \cup S$, the union of R and S, is the set of elements that are in R or S or both. An element appears only once in the union even if it is present in both R and S.

Hvis man ser på indeksene som en samling med par av attributtverdier og primærnøkler, kan man bruke en slik definisjon. For at algoritmen skal bli effektiv er det ønskelig å i størst mulig grad bare se på attributtverdiene i indeksen. Fra en slik synsvinkel blir indeksene multimengder (bags) og unionen blir dermed definert slik:[8]

[6] jf. [5, avsnitt 16.2.2].
[7] Denne må ikke forveksles med unionsoperatoren mellom to spørringer.
[8] Definisjonen for union på multimengder er også hentet fra [5].

> When we take the union of two bags, we add the number of occurrences of each tuple. That is, if R is a bag in which the tuple t appears n times, and S is a bag in which the tuple t appears m times, then in the bag $R \cup S$ tuple t appears $n + m$ times.

Å bruke multimengdeunionen som basis for algoritmen vil derimot ikke gi riktig resultat, fordi indeksene vil gjennom dette miste mengdeegenskapen. Vi må derfor ta utgangspunkt i den første definisjonen.

Selve algoritmen utnytter at indeksene er sorterte, og derfor trenger den bare å løpe sekvensielt gjennom begge inputindeksene samtidig og sammenligne deres attributtverdier. Så lenge attributtverdiene til de to operandene er forskjellige, er dette trivielt. Man ser hele tiden på den minste verdien, kopierer denne til resultatet, og fortsetter med neste forekomst i indeksen verdien ble hentet fra.

Problemet oppstår når verdiene er like. Siden vi ikke har en sortering av tupler med like attributter, må algoritmen gjøre et lineært søk for å finne de som også har like primærnøkler. Det må passes på at disse ikke blir kopiert mer enn én gang til resultatet.

I algoritmen settes det av plass i minnet til det maksimale antall verdier som resultatet kan få. Dette maksimumet nås når to indekser I_1 og I_2 er slik at $I_1 \cap I_2 = \emptyset$. Da er kardinaliteten til $I_1 \cup_I I_2$ lik summen av antall verdier i inputindeksene. Overflødig minne frigis ikke før spørringen er ferdig prosessert.

4.4.3 Equi-join

Dette er en «inner join». Det vil si at joinattributter med NULL-verdier ikke kommer med i resultatet, og den skiller seg bare fra en «natural join» ved at joinattributtene kan ha forskjellige navn. Man må derfor spesifisere i spørringen hvilke attributter det skal joines på fra hver relasjon. Et krav for å joine to indekser er at attributtene må ha det samme domenet.

Algoritmen som benyttes kalles «merge join» og står beskrevet i [5, avsnitt 15.4.7], med den forskjell at vi utnytter at indeksene allerede er sorterte. I hovedsak løpes det sekvensielt gjennom de to inputindeksene, I_1 og I_2, og algoritmen forsøker å finne den minste verdien som er lik for de to indeksene (ulike verdier ignoreres). Når man har funnet to like verdier, joines verdien i I_1 med alle verdier i I_2 som har samme verdi. Deretter fortsettes det med neste forekomst i I_1. Denne prosessen repeteres til man har løpt gjennom begge indeksene.

Det settes av plass til det teoretisk maksimale antall verdier som resultatet kan få. Dette oppstår hvis hvert attributt i den ene indeksen joines med hvert attributt i den andre. Dette antallet er lik produktet av antall verdier i de to inputindeksene. På samme måte som i unionen blir overflødig minne ikke frigjort før spørringen er ferdig prosessert.

4.4.4 Projeksjon

En projeksjon i Rindex er ikke det samme som en projeksjon i relasjonsalgebraen. I Rindex er den definert på følgende måte:

Definisjon 4.4 (Projeksjon)**.** *En* projeksjon *er en funksjon* $\pi : (\mathbf{I} \times \mathbf{A}) \rightarrow \mathbf{R}$*, som gitt en indeks* $I \in \mathbf{I}$ *og en mengde attributter* $A \subseteq \mathbf{A}$*, gir relasjonen* $R \in \mathbf{R}$ *hvor* A *er attributtene, og forekomstene er gitt av primærnøklene i* I*.*

Med andre ord genererer projeksjonen en relasjon, med gitte attributter, fra en mengde primærnøkler. Det som er forskjellig her i forhold til relasjonsalgebra, er blant annet at vi får en indeks og ikke en relasjon som input.

Resultatet er en relasjon med de gitte attributtene. Verdimengden til disse attributtene er gitt av primærnøkkelverdiene i indeksen. Projeksjonen skriver bare resultatrelasjonen direkte ut til skjermen, og lagrer den altså ikke.

4.4.5 Substitusjon

En del av de omtalte operasjonene krever at inputet skal være et bestemt attributt, for eksempel krever en union at inputindeksene har felles attributt. For å oppnå dette må det ofte utføres en substitusjon på den ene av inputindeksene først. Denne operasjonen bytter ut attributtet i en indeks med et annet attributt som har samme primærnøkkel.

Algoritmen utføres ved å løpe sekvensielt gjennom inputindeksen, og slå opp i den inverterte indeksen for å finne den nye attributtverdien som skal substitueres inn for den gjeldende verdien. Man kopierer den nye verdien, samt primærnøkkelen, til resultatindeksen. Til slutt sorteres resultatet.

4.4.6 Indekssortering

For å sortere indeksene bruker vi «quick sort» med avskjæring (cut off). Avskjæringen består i at algoritmen benytter innstikkssortering på små arrayer. En indeks sorteres på attributtverdiene. Sortering er nødvendig både når vi genererer indeksene, og etter en substitusjon.

Kapittel 5

Globale datastrukturer

5.1 Trær

Trær er en viktig datastruktur i Rindex. Vi bruker et syntakstre for å representere spørringen slik den blir gitt fra brukeren. Videre har vi et spørretre, som er en transformasjon av syntakstreet hvor SQL er byttet ut med relasjonsalgebra. Til slutt har vi et tre for å representere eksekveringsplanen.

Det eksisterer aldri mer enn ett tre om gangen i Rindex, og vi valgt å gjøre transformasjoner på det eksisterende treet for å få et nytt tre.

Trærne er bygget opp av ulike noder. Tretypen bestemmes av hvilke typer noder treet består av. Nodene er implementert som C++-klasser, med en hierarkisk struktur. Vi har definert noen ulike grensesnitt i filen `interfaces.hpp` som nodene utvider for å sikre en bestemt oppførsel.[1] Noen noder implementerer flere grensesnitt fordi de skal støtte ulike typer oppgaver. Slike noder kan være med i flere enn én type trær.

Noder som representerer samme operatorklasse er gruppert sammen ved at de arver fra en felles abstrakt klasse. Eksempler på slike grupperinger er nodene for logiske operatorer, aritmetiske operatorer, sammenligningsoperatorer og relasjonsalgebraoperatorer.

5.1.1 Syntakstre

Resultatet av å parse en SQL-spørring er et syntakstre. Vi har valgt å bruke et abstrakt syntakstre (heretter bare kalt syntakstre) uten å gå veien om et parsetre først. Dette er vanlig praksis, jf. [10, avsnitt 3.3.2]. Roten i syntakstreet er en `QueryNode`. Denne noden har pekere til de forskjellige klausulene i en SQL-spørring, og har som eneste funksjon å binde disse klausulene sammen.

Semantikksjekk av spørringer gjøres på dette treet. Disse nodene må

[1] For eksempel brukes grensesnittet `Optimizable` for at noden skal støtte optimalisering.

implementere grensesnittet `Analyzable` siden det definerer metoden `check_-semantic()` for bruk i sjekken.

5.1.2 Spørretre

Spørretreet representerer relasjonsalgebraekvivalenten til syntakstreet. Treet er nødvendig for å kunne gjøre optimaliseringer, og det blir til ved en transformasjon fra syntakstreet.

Siden nodene i dette treet skal støtte optimalisering, må de implementere grensesnittet `Optimizable`.

5.1.3 Plantre

Plantreet konstrueres ved en transformasjon av spørretreet, og har i tillegg til nodene fra det treet flere noder som styrer eksekveringen (substitusjonsnoder).

Nodene i dette treet må implementere grensesnittet `Executable` for å støtte eksekvering.

5.2 Databaseskjemaet

Definisjon 5.1 (Databaseskjema)**.** *Relasjonens navn og mengde med attributter kalles skjemaet til relasjonen. Mengden av alle slike relasjonsskjemaer, samt de ulike integritetsreglene, utgjør* databaseskjemaet*.*

Vi bruker klassen `DBSchema` for å holde på skjemainformasjonen. Klassen inneholder strukturer for å ta vare på dette. For relasjonene inneholder den informasjon om navn, antall attributter som utgjør primærnøkkelen og antall attributter og tupler i relasjonen selv. Til hvert attributt er det assosiert et navn og en datatype. Blant integritetsreglene har vi valgt å bare ta vare på informasjon om primærnøkler.

For å referere til de ulike relasjonene og attributtene brukes unike tall (IDer) i steden for navn internt i Rindex. Dette er gjort av effektivitetshensyn. Vi har også implementert datastrukturer for å raskt kunne oversette begge veier mellom navn på relasjoner og attributter, og deres tilsvarende interne ID.

Datastrukturene i skjemaet bygges under oppstart av Rindex, og blir initialisert fra `IndexManager`-objektet. Oppbygningen utføres ved å referere til databaseskjemaet i den underliggende DBMSen. Kommunikasjon skjer via et generelt «Application Programming Interface» (API) mot databasen, representert ved `DBFrontend`-objektet (se avsnitt 5.3 for nøyere beskrivelse av denne).

`DBSchema`-objektet brukes særlig under semantikksjekken (se avsnitt 6.3) og under generering av indeksene (se avsnittet 4.3). Derfor inneholder klassen

funksjoner for slik bruk, blant annet funksjoner for å hente ut informasjon fra de interne datastrukturene, og en funksjon for å sjekke om et gitt attributtnavn er entydig over alle aktuelle relasjoner (det vil si alle relasjoner som er nevnt i `FROM`-klausulen). Eventuelle feil blir håndtert ved å kaste unntak (exceptions).

5.3 Støtte for ulike DBMSer

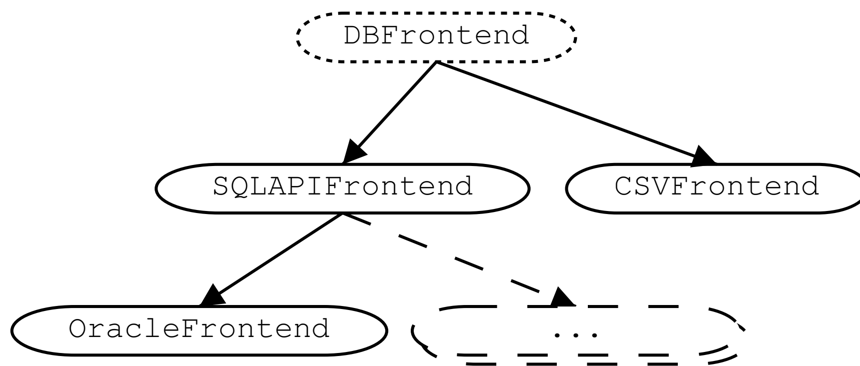
Rindex er konstruert for å enkelt kunne utvides med støtte for flere ulike underliggende DBMSer. Til dette har vi valgt å lage et felles API. Klassen `DBFrontend` er en abstrakt klasse som definerer dette APIet. Det inneholder funksjoner for å utføre nødvendige spørringer mot DBMSen, samt funksjoner for å iterere over og hente ut data fra resultatsettet. Vi har valgt å bruke mellomvaren (middleware) `SQLAPI++` [17] for å støtte dette. På denne måten abstraherer vi bort kompleksisten i å støtte ulike underliggende DBMSer, i tillegg til å unngå å måtte gjøre denne tidkrevende implementasjonsoppgaven selv. Vi har derimot laget en klasse `SQLAPIFrontend` som arver fra `DBFrontend`. Denne klassen inneholder felles kode for alle DBMSer som bruker `SQLAPI++`. Dette gjelder informasjon for oppkobling mot databasen, samt et tilkoblingsobjekt og et kommandoobjekt fra `SQLAPI++`. I tillegg er iteratorfunksjonen og funksjonene for å hente ut data fra resultatsettet felles, og de er derfor implementert i `SQLAPIFrontend`. De operasjonene som er spesielle for hver underliggende DBMS, for eksempel spørringene som kreves for å hente ut informasjon fra databaseskjemaet, er implementert i egne frontendklasser for hver DBMS. Dette er gjort fordi forskjellige DBMSer ofte har ulike måter å hente ut slik informasjon på.

I versjon 0.1 av Rindex har vi implementert støtte for Oracle og «Comma Separeted Values» (CSV)-filer[2] i henholdsvis klassene `OracleFrontend` og `CSVFrontend`.

Figur 5.1 viser hvordan klassehierarkiet er ordnet. Pilene indikerer arv mellom klassene. Ellipsen rundt `DBFrontend` er prikkete for å indikere at den er abstrakt, mens samlingen med stiplede ellipser under `SQLAPIFrontend` indikerer hvor eventuelle nye klasser for å støtte andre DBMSer vil passe inn i figuren.

Denne designen gjør det enkelt å implementere støtte for en annen DBMS. Det er kun nødvendig å konstruere en ny klasse tilsvarende `OracleFrontend` for denne. I tillegg skal det også være mulig å eventuelt erstatte `SQLAPI++` med et annet bibliotek uten å gjøre store endringer i koden.

[2]De bruker et format som følger UNIX-konvensjonene for slike filer, se [15, avsnitt 5.2.1]. Slike kommaseparerte filer er ikke en DBMS, men de har blitt brukt som en enkel datakilde under utviklingen av Rindex for å forenkle testingen.



Figur 5.1: Klassehierarki for APIene og de ulike frontendene mot DBMSene

Kapittel 6

Statisk analyse (syntaks og semantikk)

Definisjon 6.1 (Statisk analyse)**.** Statisk analyse *omfatter all analyse som kan utføres før en spørring eksekveres. Dette innebærer all syntakssjekking og deler av semantikksjekken.*

Under planleggingen av Rindex skrev vi følgende om denne delen av plattformen:

> Kompilatoren (`Analyzer` og `Parser`) skal utføre følgende oppgaver:
>
> 1 Sjekke at spørringen er syntaktisk korrekt.
>
> 2 Bygge syntakstreet til spørringen.
>
> 3 Utføre semantikksjekken. Dette innebærer å sjekke at de attributtene og relasjonene som det er referert til finnes, at ingen attributter nevnt i spørringen er tvetydige, og at relasjonen til hvert nevnte attributt også er med i spørringen. I tillegg må det kontrolleres at typene brukt i de ulike aritmetiske operatorene og sammenligningsoperatorene er kompatible.
>
> Når det gjelder punkt en og to, er `flex`[1] i kombinasjon med `bison`[2] mulige verktøy.

Vi valgte å bruke disse verktøyene istedenfor å gjøre jobben fra grunnen av, fordi vi ventet at arbeidet ville være enklere, mindre tidkrevende og lettere å få riktig. Men det viste seg å ta noe mer tid enn forventet, spesielt siden vi måtte lære oss mye av teorien verktøyene bygger på.

De to neste avsnittene beskriver skanneren og parseren mer i detalj. Siste avsnitt omhandler semantikksjekken.

[1] GNU-versjonen av `lex`.

[2] GNU-versjonen av `yacc`.

6.1 Skanner

Definisjon 6.2 (Skanner)**.** *En* skanner *skal lese et input, bryte det opp i gjenkjente leksikografiske enheter (leksemer) og tilordne disse til logiske enheter (tokens).*

`flex` er et verktøy som brukes for å generere skannere, og den bruker regulære uttrykk (jf. [10, avsnitt 2.2] og [11]) for å beskrive de leksikografiske mønstrene den skal gjenkjenne. I `flex` kan man tilordne C-kode til hvert av disse mønstrene, kalt regler. Denne koden kopieres inn i den genererte skanneren og utføres når det korresponderende regulære uttrykket gjenkjennes i teksten.

Vi brukte `flex` til å generere en skanner der de lovlige tokenene enten er Rindex-kommandoer (for eksempel `quit`) eller deler av en SQL-spørring (for eksempel `SELECT`). Hver gang den gjenkjenner et lovlig token, signaliserer den dette til parseren (se avsnitt 6.2). Dette fører til en nært samspill mellom skanneren og parseren, der parseren har kontroll over skanneren og ber om neste token når den trenger det.

Skanneren var den første komponenten vi implementerte. Vi begynte med å konstruere et enkelt eksempel, satt sammen av eksempler fra [9] og [11]. Deretter utvidet vi dette til å gjenkjenne de enhetene vi ønsket, og koblet den til slutt sammen med parseren.

6.2 Parser

Definisjon 6.3 (Parser)**.** *En* parser *skal avgjøre den syntaktiske strukturen til inputet, utfra tokenene den mottar fra skanneren. Det er også vanlig at parseren konstruerer et parsetre eller syntakstre som representerer denne strukturen. Parsingsprosessen blir ofte kalt* syntaktisk analyse.

Rindex støtter et lite subsett av SQL-standarden. I hovedsak støtter den enkle `SELECT`-setninger, med `FROM`-, `INNER JOIN`-, `WHERE`- og/eller `ORDER BY`-ledd. Da vi designet plattformen, definerte vi en Backus-Naur Form (BNF)-grammatikk[3]. Vi gjorde noen få endringer av grammatikken underveis (under implementasjonen), og følgende punkter gjelder for versjon 0.1 av Rindex:

- Vi har valgt å ikke støtte aritmetiske operasjoner på attributter, verken mellom to attributter eller på ett attributt og en konstant skalarverdi. Dette på grunn av tidsmangel og høyere prioritering av andre deler av plattformen.[4] Vi hadde planlagt å implementere dette ved å alltid

[3] BNF ble utviklet for å spesifisere programmeringsspråket Algol60.

[4] Vi prioriterte generelt å få ferdig et sammenhengende og nyttig (men altså ikke komplett) system, og utsatte dermed de mindre brukte og mer tidkrevende og kompliserte delene.

flytte konstanten og den inverterte aritmetiske operatoren fra attributt-siden til den andre siden av sammenligningsoperatoren (som alltid må eksistere i denne konteksten). Et eksempel: $a + 2 > 4$ ville ha blitt oversatt til $a > 4 - 2$.

- Den opprinnelige grammatikken støttet seleksjon på to attributter, det vil si bruk av sammenligningsoperatorene på to attributter, men dette har vi heller ikke implementert i denne versjonen. Begrunnelsen for dette er gitt i avsnitt 4.4.1.
- Vi gikk også bort fra den opprinnelige planen om at en join kunne ha flere betingelser (alle påkrevd med `AND`), og støtter i denne versjonen kun én joinbetingelse. Dette gjorde vi fordi det viste seg vanskeligere å implementere flere joinbetingelser enn først antatt, og derfor fikk vi ikke tid til å gjøre dette.
- Vi bestemte oss underveis for å legge til støtte for `ORDER BY` (kun for ett attributt i denne versjonen). Dette gjorde vi fordi det viste seg at det var enkelt å implementere, samtidig som det ville hjelpe på oversiktligheten (det vil si på utskriften) for en del resultater.

Dette er den endelige versjonen av grammatikken vi har implementert i versjon 0.1:[5]

<query>	::=	<select> <from> <where> <order-by>
<select>	::=	`SELECT` <attribute-list>
<attribute-list>	::=	<attribute>, <attribute-list> | <attribute> | `*`
<attribute>	::=	<attribute-name> | *relation-name*`.*`
<attribute-name>	::=	*relation-name*`.`*attribute-name* | *attribute-name*
<from>	::=	`FROM` <join-relation>
<join-relation>	::=	`(` <join-relation> `INNER JOIN` <join-relation> `ON` <join-condition> `)` | *relation-name*

[5] Terminerende symboler er notert i skrivemaskinskrift, produksjonsymboler er notert <. . . > og navn er notert i kursiv.

<join-condition> ::= <attribute-name> `=` <attribute-name>

<where> ::= `WHERE` <condition>
| ϵ

<order-by> ::= `ORDER BY` <attribute-name>
| ϵ

<condition> ::= <boolean-term>
| <condition> `OR` <boolean-term>

<boolean-term> ::= <boolean-factor>
| <boolean-term> `AND` <boolean-factor>

<boolean-factor> ::= <boolean-pri>
| `NOT` <boolean-pri>

<boolean-pri> ::= <comparison>
| `(` <condition> `)`

<comparison> ::= <scalar-exp> <comp-opr> <scalar-exp>
| <attribute-name> <comp-opr> <scalar-exp>

<scalar-exp> ::= <term>
| <sign> <term>
| <scalar-exp> <add-opr> <term>

<term> ::= <factor>
| <term> <mult-opr> <factor>

<factor> ::= *token*
| `(` <scalar-exp> `)`

<comp-opr> ::= `=` | `<>` | `<` | `>` | `<=` | `>=`

<mult-opr> ::= `*` | `/`

<add-opr> ::= `+` | `-`

<sign> ::= <add-opr>

For å lage en parser som sjekker syntaksen ut fra denne grammatikken, valgte

vi å bruke verktøyet `bison`. Det er en generell parser-generator som konverterer en beskrivelse av en grammatikk for LALR(1) kontekst-frie grammatikker (jf. [10, avsnitt 5.4.3]) til et C-program som parser denne grammatikken. `bison` aksepterer denne beskrivelsen på en BNF-form kalt «machine-readable BNF» i manualen [4]. Hver produksjon, en regel i `bison`, kan i denne grammatikken ha en mengde med aksjoner knyttet til seg. Aksjonene er skrevet i C/C++-kode og utføres når `bison` klarer å matche den tilsvarende regelen mot deler av inputet (fra skanneren). `bison` benytter en nedenfra-opp (bottom-up) parsingsalgoritme, jf. [10, kapittel 5] og [4].

Vi kunne oversette grammatikken vår mer eller mindre direkte til regler i `bison`-grammatikken. Dette ble første versjon av parseren – den sjekket syntaksen og enten godkjente eller ikke godkjente inputet. Fordi parseren også skulle generere syntakstreet til spørringen, måtte vi skrive kode (aksjoner for de aktuelle produksjonene) for å generere og linke sammen nodene (se avsnitt 5.1.1). Generelt vil hver mengde med aksjoner knyttet til en regel generere en ny node i treet, sette pekere til eventuelle barnenoder, og returnere den nye noden til neste regel som matches. På denne måten bygges treet nedenfra og opp. Hvis parseren ikke klarer å matche en regel, innebærer dette en syntaksfeil i inputet, og et unntak (exception) kastes til kontrollobjektet. Ellers, hvis hele inputet godkjennes av parseren, vil man til slutt ha et fullstendig syntakstre. Dette treet brukes videre i semantikksjekken (se neste avsnitt). Parseren vil også kunne gjenkjenne en kommando (for eksempel `quit`) og vil i dette tilfellet ikke generere noe tre, men heller signalisere denne kommandoen til kalleren (kontrollobjektet).

Følgende eksempel kan illustrere resultatet av parseprosessen:

Eksempel 6.1 Utgangspunktet for eksemplet er følgende spørring:

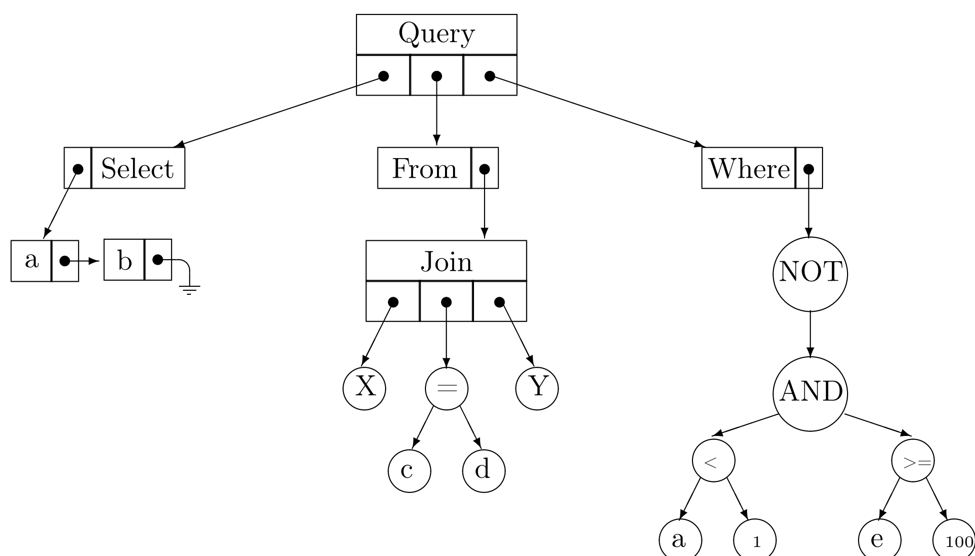
```
SELECT a, b
FROM X INNER JOIN Y ON c = d
WHERE NOT (a < 1 AND e >= 100)
ORDER BY b;
```

der relasjonene er definert som $X(\overline{a}, c)$ og $Y(\overline{b, d}, e)$, hvor $X.c \subseteq Y.d$ og alle attributter er av typen `INTEGER` (for å gjøre eksemplene enklere).

Denne spørringen er valgt fordi den inneholder de fleste og mest interessante elementene for videre diskusjon, og brukes også som utgangspunkt for flere senere eksempler.

Figur 6.1 viser syntakstreet som parsingen av SQL-setningen over resulterer i.

△



Figur 6.1: Syntakstreet

6.3 Semantikksjekker

Dersom parseren godkjenner inputet, det vil si at syntaksen er riktig, utføres siste steg i den statiske analyseprosessen — semantikksjekken. Dette skjer ved en postfiks traversering av syntakstreet ved hjelp av rekursive kall på `check_semantic()`-funksjonen i hver node.

Semantikksjekken består, som nevnt i tredje punkt i innledningen av kapitlet, hovedsakelig av følgende punkter:

- Verifisere at alle attributter og relasjoner som er nevnt i spørringen, eksisterer i skjemaet til databasen. Dette gjøres ved å konsultere skjemaet i `DBSchema`-objektet.

- Kontrollere at ingen attributter er tvetydige, dersom det i SQL-spørringen ikke er eksplisitt angitt hvilken relasjon attributtet skal finnes i. Entydigheten sjekkes også her ved et kall på `DBSchema`-objektet.

- Ekspandere hver forekomst av $*$ i `SELECT`-klausulen til de tilsvarende attributtene.

- For å forsikre seg om at relasjonen til hvert nevnte attributt også er med i spørringen, sjekkes semantikken i `FROM`-noden først. Hver relasjon får her en referanse i en midlertidig, intern datastruktur. Dermed er det enklere, og raskere, senere å sjekke at hvert attributt finnes i en av disse relasjonene. Hvert attributt vil dermed få en referanse til den tilhørende relasjonen.

- Nodene som representerer de aritmetiske operatorene og sammenligningsoperatorene må selv også sjekke at barnas typer er kompatible, slik at operatoren de representerer er gyldig og kan utføres. De logiske operatornodene behøver ikke gjøre en slik sjekk fordi de forsvinner i optimaliseringsprosessen (se avsnitt 7.2).

Ved eventuelle semantiske feil kastes et unntak (exception) til kontrollobjektet. Dette fører til at resten av semantikksjekken ikke blir utført og at eventuelle andre semantiske feil ikke blir rapportert.[6] Vi har ikke sett behovet for å forbedre dette siden det er praksis i de fleste andre DBMSer, for eksempel Oracle.

Ellers fullføres sjekken, og det korrekte og gyldige treet returneres til kontrollobjektet for videre bruk. I dette treet vil alle attributt-noder med $*$ være ekspandert til nye attributtnoder. I tillegg vil alle attributtnoder i treet ha en entydig referanse til attributtet og den tilhørende relasjonen i `DBSchema`-objektet.

Følgende eksempel kan illustrere hvordan semantikksjekken utføres for en spørring:

Eksempel 6.2 Dette eksemplet tar utgangspunkt i spørringen fra eksempel 6.1.

- Algoritmen for semantikksjekken vil først kontrollere `FROM`-noden. Her ligger de to relasjonene X og Y. Siden disse er distinkte, vil semantikken være entydig. Begge relasjonene får tilordnet referanse i den midlertidige, interne datastrukturen. I tillegg må attributtene nevnt i joinbetingelsen sjekkes, men dette støttes ikke i versjon 0.1 av Rindex på grunn av tidsmangel.
- Deretter kontrolleres attributtene a og b i `SELECT`-noden. Begge er entydige, og relasjonene de er definert i eksisterer og er med i `FROM`-leddet i spørringen. Attributtene får satt en referanse til den tilhørende relasjonen (denne brukes blant annet under eksekveringen).
- I betingelsene til `WHERE`-klausulen, sjekkes attributtene a og e på samme måte som de i `SELECT`-klausulen. I tillegg må '<'-noden sjekke at barnas typer er kompatible, slik at sammenligningen kan utføres. Siden a er av typen `INTEGER` vil det ikke oppstå en konflikt her. Det samme gjelder for '>='-noden, siden e også er av samme type.
- Til slutt sjekkes b-attributtet i `ORDER BY`-leddet på samme måte som de i `SELECT`-leddet.

△

[6] Det er vanlig, for eksempel i språkkompilatorer, å forsøke å fortsette med sjekken for å kunne rapportere om flere feil til brukeren, jf. [10, avsnitt 5.7].

Kapittel 7

Optimalisering og plangenerering

Optimalisering og plangenerering er to viktige og sentrale deler i en DBMS. Under planleggingen av Rindex skrev vi følgende:

> Optimaliseringsenheten i et databasesystem har typisk følgende oppgaver:
>
> 1 Generere et spørretre ut fra syntakstreet.
>
> 2 Optimalisere dette spørretreet.
>
> 3 Generere en plan ut fra spørretreet.

Dette har vi implementert i klassen `Optimizer`, og hvert punkt beskrives i de følgende avsnittene.

7.1 Spørretreet

Etter at parseren returnerer syntakstreet, gis dette som input til `Optimizer`-objektet. Det første som gjøres er å oversette syntakstreet til et spørretre, det vil si et tre med relasjonsoperatorene innsatt, for å kunne jobbe videre med treet ved bruk av algebraiske regler. De ulike relasjonsoperatorene vi bruker er projeksjon (π), seleksjon (σ), equi-join ($\bowtie$) og i tillegg vår egen substitusjonsoperator (S).

Følgende sitat er hentet fra notatet vi skrev under designen av Rindex:

> Spørretreet vil ha den generelle formen vist i figur 7.1. Her er s listen med attributter gitt i `SELECT`-klausulen, og c er betingelsene gitt i `WHERE`-klausulen. En spørring uten `WHERE` gir opphav til en seleksjon hvor c alltid er sann. Figuren viser trær henholdsvis med og uten join.

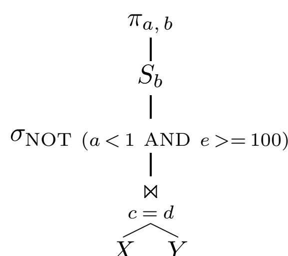
Figur 7.1: Et generelt spørretre.

For å oppnå en slik struktur gjøres det noen enkle transformasjoner på syntakstreet. Først brukes `Query`-noden til å nå `Select`-noden, som transformeres til en `Projection`-node. Denne noden blir roten i det nye treet. Dersom en `OrderBy`-node eksisterer erstattes denne med en `Substitution`-node, fordi dette resulterer i en sortert struktur (se avsnitt 4.4.5), som dermed blir barnet til projeksjonen. En eventuell `Where`-node transformeres til en `Selection`-node, med betingelsene i `Where`-noden som barn, og legges deretter til i det nye treet. `From`-noden behandles til slutt, fordi denne alltid skal ligge nederst i treet (siden dette er det opprinnelige datagrunnlaget for alle andre operasjoner).

Følgende eksempel kan illustrere denne transformasjonen:

Eksempel 7.1 Dette eksemplet bygger videre på eksempel 6.1. Figur 7.2 viser spørretreet som blir resultatet av transformasjonen beskrevet over.

△



Figur 7.2: Spørretreet

7.2 Optimalisering

Etter at spørretreet er blitt generert, begynner selve optimaliseringsprosessen. Som allerede nevnt, er optimalisering en viktig del av en DBMS. Det er utarbeidet mange algebraiske regler for ulike forbedringer av treet, jf. [5,

avsnitt 16.2]. Slik vi planla før implementasjonen, har vi ikke gjort noen optimaliseringer i denne versjonen av Rindex, men rammeverket for senere forbedringer er lagt. Vi skrev følgende i planleggingsfasen:

> Når det gjelder optimalisering av spørretreet, har vi minst tre alternativer. Plattformen kan enten gjøre optimaliseringer selv, få servert ferdigoptimaliserte spørringer fra brukeren, eller en kombinasjon av disse.
>
> I første versjon er nok det mest realistisk at Rindex selv ikke gjør flere optimaliseringer enn hensiktsmessig. Dette fordi vi har begrenset tid, og for å slippe å måtte forholde seg til problemer som kostestimering og lignende. På den annen side er det ikke usannsynlig at det vil komme en slik utvidelse på et senere tidspunkt. Vi kommer derfor til å bygge et rammeverk i plattformen som gjør at denne type funksjonalitet enkelt kan legges til. En type optimalisering som er både enkel og svært effektiv, er å flytte seleksjoner så langt ned i spørretreet som mulig. For å få til dette er det nødvendig at seleksjonen splittes opp slik at AND eller OR ikke forekommer i uttrykket. Dette oppnås ved å anvende oppsplittingsreglene[1].

Vi har bare implementert oppsplittingsreglene, ikke det å flytte seleksjoner nedover i treet, siden vi prioriterte andre deler av plattformen. Vi har også valgt å ikke splitte opp og flytte projeksjoner nedover i treet fordi strukturen vi har brukt på indekser og mellomlagringer, jf. avsnitt 4.1, gjør denne optimaliseringen unødvendig. I tillegg til dette transformerer vi bort alle `NOT`-uttrykk. Det er gjort ved å la hver enkelt `NOT`-node synke nedover i treet. Den vil negere hver node den passerer (for eksempel vil en addisjonsnode bli oversatt til en subtraksjonsnode), inntil den møter en annen `NOT`-node og begge fjernes, eller til den kommer til bunnen av treet og trygt kan fjernes.

7.2.1 Implementasjonen

Vi har valgt å implementere optimaliseringsprosessen ved å la hver node som kan inngå i spørretreet deklarere en `optimize()`-funksjon. Hver funksjon har ansvar for å optimalisere den noden den tilhører i den gjeldende konteksten (for eksempel posisjon i treet, foreldrenoder og barnenoder), samt å returnere noden som er resultatet av optimaliseringene utført på den opprinnelige noden. Dette innebærer at hver funksjon kan opprette nye noder og eventuelt fjerne seg selv (jf. oppsplittingsreglene for seleksjoner med `AND` og/eller `OR`). Dermed vil en rekursiv traversering av spørretreet, ved kall på nodenes `optimize()`-funksjoner, føre til en generering av et nytt, optimalisert tre.

[1] Definert i 4.3.

Det er bare `NOT`-noden som utføres prefiks, resten gjøres postfiks. Alt dette gjør algoritmen for optimaliseringer generell og utvidbar.

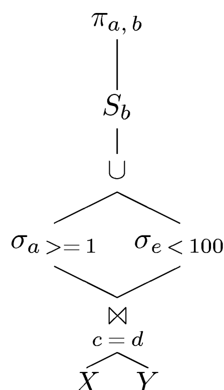
Følgende eksempel illustrerer optimaliseringsprosessen:

Eksempel 7.2 Dette eksemplet bygger videre på eksempel 7.1. Følgende optimalisering vil bli utført på betingelsen i seleksjonen:

- Ved en rekursiv traversering av treet vil `NOT`-noden bli behandlet først blant seleksjonsbetingelsene. `NOT`-noden vil, siden denne noden utføres prefiks, bli flyttet nedover og dermed invertere de andre nodene underveis. Dette betyr at `AND`-noden vil bli gjort om til en `OR`-node, '`<`'-noden til en '`>=`'-node og '`>=`'-noden til en '`<`'-node.
- Deretter skal den nye `OR`-noden optimaliseres. Siden denne noden behandles postfiks, vil den optimalisere barna først.
- Dermed er det '`>=`'-noden som behandles først. Sammenligningsnoder optimaliseres ikke. Det er heller ikke nødvendig å forsøke å optimalisere etterfølgernodene, siden dette kun kan være andre sammenligningsnoder eller aritmetiske noder.
- '`<`'-noden og dens barn optimaliseres heller ikke, av samme grunn som '`>=`'-noden i forrige punkt.
- Videre vil man, i dette eksemplet, fortsette å behandle `OR`-noden. Denne noden skal, ifølge regel to i definisjon 4.3, splittes opp i en unionsnode med to seleksjonsnoder som barn. Seleksjonsnodene får hvert sitt barn fra `OR`-noden.

Resultatet av denne optimaliseringsprosessen vises i figur 7.3.

△



Figur 7.3: Det optimaliserte spørretreet

7.3 Plangenerering

Det optimaliserte spørretreet gir ingen entydig representasjon av en lineær eksekveringsrekkefølge. Derfor må en DBMS generere en eller flere alternative lineære planer for mulige eksekveringsrekkefølger. Det vil ofte eksistere svært mange mulige planer, og DBMSen bør derfor bare vurdere de mest lovende og interessante. Hver plan angir en rekkefølge på operasjonene som skal utføres, samt annen informasjon (for eksempel ressursbruk/-krav). Ut fra tilgjengelig informasjon, for eksempel ulike statistikker og heuristikker, vil DBMSen prøve å estimere kosten til de ulike planene og velge den planen med lavest forventet kost. Det er forventet at dette gir den raskeste utførelsen av spørringen. Total kost for hver plan består hovedsakelig i summen av kostene for hver av operasjonene i planen, basert på relasjonsstørrelser og lignende.

Vi har valgt å foreløpig bare generere én plan i Rindex, uten å ta hensyn til statistikker og andre faktorer. Dette har vi gjort ut fra den overordnede strategien om å lage et enkelt, utvidbart og fungerende system.

7.3.1 Implementasjonen

På samme måte som med optimaliseringsprosessen, har vi valgt å implementere plangenereringen ved en rekursiv postfiks traversering av treet. Hver node som kan forekomme i et optimalisert spørretre, deklarerer til denne oppgaven en `make_plan()`-funksjon. Denne tar seg av plangenereringen i den tilhørende noden.

Algoritmen har følgende hovedoppgaver:

- Finne ut hvilke indekser og mellomlagringer som skal brukes som input og output til hver operator. Enkelte operatorer (for eksempel join) er binære og tar to input, mens andre (for eksempel substitusjonen) er unære og tar kun ett input. Alle operasjoner resulterer imidlertid i ett output, som skal være input til neste operator.[2]

- Bestemme hvor mange mellomlagringer som totalt trengs under eksekveringen, slik at nødvendig plass kan allokeres på forhånd. Dette innebærer gjenbruk av mellomlagringer. Utfordringen her er å, underveis mens man genererer planen, finne ut om en mellomlagring trygt kan brukes om igjen. Den kan gjenbrukes dersom ingen andre senere operatorer er avhengige av den, siden den da trygt kan fjernes. På dette punktet er ikke algoritmen helt optimal. Dette illustreres i første og tredje punkt i eksempel 7.3.

- Legge inn substitusjonsnoder der det trengs, og bestemme attributtene det skal byttes fra og til. I figurene har vi brukt notasjonen S_t, der t

[2] Unntaket er den siste noden i treet — projeksjonsnoden.

er attributtet den skal substituere inputet til. Også antall substitusjoner er det mulig å minimalisere, men dette har vi valgt å ikke gjøre her. Dersom det er nødvendig, legger algoritmen inn en substitusjonsnode før barnet, eller før høyre barn hvis operatoren er binær[3]. Dette må gjøres for et input-attributt hvis det ikke er kompatibelt med det forventede attributtet.

Hver `make_plan()`-funksjon må derfor sørge for å utføre punktene over. Kommunikasjonen mellom nodene skjer hovedsakelig ved parametere og returverdier. Hver funksjon returnerer hvilken mellomlagring noden skal legge resultatet i under eksekveringen. Dermed kan en node beregne hvilke indekser eller mellomlagringer den skal bruke som input ved å analysere returverdien fra plangenereringen i barnet/barna.

For å styre gjenbruk av og totalt antall mellomlagringer, brukes noen globale variable som simulerer en slags stakk. «Toppen av stakken» indikerer den høyeste IDen på en mellomlagring som er i bruk i den gjeldende posisjonen i treet. Dette betyr at alle mellomlagringer med ID over denne trygt kan brukes om igjen. I tillegg vil en global variabel holde oversikt over totalt antall samtidige mellomlagringer som trengs. Algoritmen vår er ikke helt optimal, det vil si at vi sløser med mellomlagringer, men ikke spesielt mye eller ofte, og det oppstår som regel bare i spesielle situasjoner.

Vi bruker et eksempel for å illustrere algoritmen:

Eksempel 7.3 Dette er en fortsettelse på eksempel 7.2.

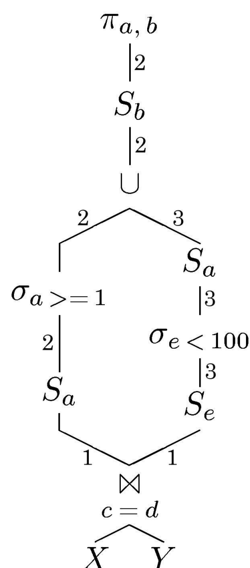
Resultatet av plangenereringsalgoritmen er planen vist i figur 7.4. Planen er visualisert som en graf, men den representerer en entydig, lineær eksekveringsplan fordi grafen alltid gjennomløpes på samme måte (postfiks traversering). Hver gren i planen er nummerert med IDen på mellomlagringen som er output (resultatet) fra en operator og input til operatorer over den i grafen. Tallene referer til de som er brukt i dette eksemplet. Algoritmen beskrevet over vil gjennomgå følgende steg for å generere planen, når den utføres på spørretreet i figur 7.3:

- Algoritmen bygger, som sagt, på en postfiks traversering, og derfor vil `Join`-noden bli behandlet først.[4] Det er av og til mulig å komme til samme `Join`-node ved å følge ulike stier i treet (det er her egentlig en graf), som i dette eksemplet utfra `Union`-noden. Derfor sørger vi for at plangenereringsalgoritmen bare utføres én gang for den første `Join`-noden (og dens etterfølgere). De resterende gangene den øverste `Join`-noden nås, brukes resultatet fra den første gangen.

  Hvis input-attributtet fra barnet ikke er det forventede, det vil si det attributtet joinen skal utføres på, settes det inn en `Substitution`-node

[3] Unntaket er `Join`-noden, se første punkt i eksempel 7.3.

[4] Relasjonsnodene bidrar ikke selv med noe til planen.



Figur 7.4: Den genererte planen

før barnet. Dette skjer ikke i dette eksemplet, siden begge barna er relasjoner og dermed gir riktig input.

Dersom begge barna til `Join`-noden er relasjoner, som i dette eksemplet, kan ingen mellomlagringer gjenbrukes, og den må bruke en ny. Hvis minst ett av barna ikke er en relasjon, men en mellomlagring, kan denne overskrives med resultatet. I dette eksemplet ville mellomlagring 1 ha blitt brukt.

- Deretter er det `Selection`-noden (med '`>=`' i betingelsen) som behandles.

  Først må det vurderes om det må settes inn en `Substitution`-node mellom seleksjonen og joinen. Dette må gjøres hvis attributtet som er resultatet av joinen ikke er det samme som det seleksjonen skal utføres på. I dette eksemplet må en slik node settes inn her.

  Deretter bestemmes bruk av mellomlagringer. `Substitution`-noden bruker mellomlagring 1 som input (fordi denne ble returnert fra `Join`-noden). Siden det er flere noder som skal bruke denne mellomlagringen, i dette eksemplet den andre `Selection`-noden, kan den ikke gjenbrukes, og resultatet må legges i en ny mellomlagring, her nr. 2. Man finner deretter at denne `Selection`-noden kan bruke mellomlagring 2 som input (resultatet fra `Substitution`-noden) og gjenbruke denne for outputet.

- Etter dette kaller `Union`-noden på `make_plan()`-funksjonen i den andre `Selection`-noden (med '`<`' i betingelsen).

På samme måte som for den første `Selection`-noden, må også denne ha en `Substitution`-node mellom seg og `Join`-noden.

`Substitution`-noden vil også bruke mellomlagring 1 fra joinen. Den må legge resultatet i neste ledige mellomlagring, nr. 3. Dette viser at algoritmen vår for gjenbruk av mellomlagringer ikke er optimal. Grunnen er at man nå kunne gjenbrukt nr. 1, fordi ingen flere noder er avhengige av originalinnholdet i denne. På samme måte som i forrige punkt, vil `Selection`-noden kunne bruke mellomlagring 3 både som input og output.

- På dette stadiet kan `Union`-noden behandles ferdig.

  Denne noden må også legge inn en `Substitution`-node, men bare på høyre barn, slik at attributtene er like og unionen kan utføres.

  `Substitution`-noden vil kunne bruke mellomlagring 3 både som input og til resultatet (output). `Union`-noden vil bruke mellomlagringene 2 og 3 fra henholdsvis det venstre og høyre barns returverdier. Resultatet vil gjenbruke mellomlagring 2 og dermed også frigjøre nr. 3 (nr. 1 vil ikke bli brukt av samme grunn som i forrige punkt: vi vet ikke om eventuelle andre, uplanlagte noder også er avhengige av resultatet fra `Join`-noden).

- Over `Union`-noden i treet ligger det allerede en `Substitution`-node på grunn av `ORDER BY`-leddet i spørringen. Denne kan ganske enkelt bruke returverdien fra `Union`-noden (2) både som input og output, det vil si at resultatet kan overskrive inputet.

- Til slutt behandles `Projection`-noden på samme måte som `Substitution`-noden i forrige punkt. Den vil også bruke mellomlagring 2 som både input og output[5] for eksekveringsplanen.

△

[5]Siden vi alltid bare har én projeksjonsnode i treet, vil aldri outputet bli brukt, resultatet skrives bare til skjermen (se avsnitt 4.4.4).

Kapittel 8

Planeksekvering

I de foregående avsnittene har vi vist hvordan plattformen transformerer en SQL-spørring til en spørreplan. Vi beskriver her den siste fasen i spørreprosesseringen: planeksekveringen. Her utføres de enkelte operasjonene gitt i spørreplanen (beskrevet i avsnitt 5.1.3 og 7.3) for til sist å kunne levere resultatrelasjonen som er svaret på spørringen.

Det første som må gjøres før en eksekvering kan starte, er å re-initialisere `IndexManager`. Det går i korte trekk ut på at det klargjøres plass til det antall mellomlagringer man trenger. Deretter starter selve eksekveringen.

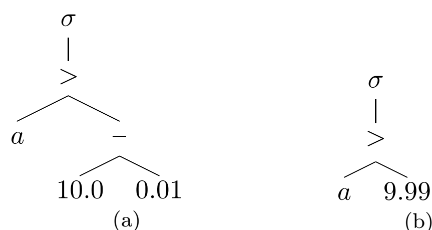
Definisjon 8.1 (Planeksekvering)**.** *En* planeksekvering *innebærer en postfiks traversering av plantreet. Hver node i dette treet tilsvarer en indeksoperasjon.*[1]

For de fleste nodene i treet er det tilstrekkelig å kalle direkte på en funksjon i `IndexManager` som er entydig bestemt av typen til noden. Men noen noder vil kreve noe ekstra bearbeidelse.

Det første unntaket skyldes at planen strengt tatt ikke er et tre, men en mer generell graf. Hvis man har minst én union i grafen, vil man finne to noder som har et felles barn. Dette barnet er en join hvis spørringen inneholder en, ellers er det en relasjonsnode. Hvis det er en join, vil joinen bli forsøkt eksekvert to ganger, noe vi ikke tillater (se figur 7.4 for et eksempel). Løsningen er et flagg i `JoinNode` som settes når den har blitt eksekvert første gang. På denne måten vil senere forsøk på eksekvering av noden bli unngått, og resultatet fra den første gangen blir gjenbrukt.

Et annet unntak er noder av typen `ArithmeticNode`. Disse blir beregnet i `Executor`. Et aritmetisk utrykk er et binærtre (med unntak av fortegnsnoder som er unære). En beregning tilsvarer å traversere dette treet postfiks. Hver av de indre nodene er her aritmetiske operasjoner som beregner et mellomresultat som foreldrenoden kan bruke i sine beregninger. Til slutt blir det laget

[1] Indeksoperasjon er definert i 4.2.



Figur 8.1: Beregning av aritmetiske uttrykk (a) før og (b) etter traverseringen

en `ScalarNode` med resultatet som verdi, og denne erstatter det aritmetiske uttrykket.

Eksempel 8.1 Å teste likhet av flyttallsverdier er ikke uproblematisk, og man må derfor oppgi en presisjon. Sett at man vil hente fra databasen et attributt a med flyttalsverdier, og bare er interessert i tupler hvor a har verdien 10. For å ta hensyn til presisjonen, kan man i SQL gi følgende where-betingelse:

```
WHERE a > 10.0 - .01 AND a < 10.0 + .01
```

I figur 8.1 ser vi på $>$ betingelsen. Figur 8.1(a) viser hvordan den er i utgangspunktet, mens 8.1(b) viser delplanen etter at uttrykket er beregnet. Figuren viser at alle aritmetiske noder har blitt fjernet i (b).

△

Vi vil nå bruke et eksempel for å illustrere algoritmen:

Eksempel 8.2 Dette er en fortsettelse på eksempel 7.3, og tar utgangspunkt i planen vist i figur 7.4. I dette eksempelet vises den rekkefølgen elementene i planen blir utført i. M_x betyr mellomlagringen med ID x, og IDene tilsvarer numrene på grenene i figur 7.4.

1 $M_1 \leftarrow X \underset{c=d}{\bowtie} Y$. Her er det viktig at det omtalte flagget i joinnoden blir satt.

2 $M_2 \leftarrow S_a(M_1)$

3 $M_2 \leftarrow \sigma_{a \geq 1}(M_2)$

4 Traverseringen har nå kommet til joinnoden for andre gang, men ingenting blir gjort siden flagget i denne noden allerede er satt.

5 $M_3 \leftarrow S_e(M_1)$

6 $M_3 \leftarrow \sigma_{e < 100}(M_3)$

7 $M_3 \leftarrow S_a(M_3)$

8 $M_2 \leftarrow M_2 \cup_I M_3$.

9 $M_2 \leftarrow S_b(M_2)$

10 $Svar \leftarrow \pi_{a,b}(M_2)$

$\triangle$

Dette eksemplet viser at mellomlagringer (M_1) må bevares inntil alle noder som trenger dem, det vil si alle som har `Join`-noden som direkte barn, er blitt utført.

En utfordring som oppstår når man støtter seks forskjellige datatyper, er at vi får seks nesten identiske funksjoner for alle sammenligningsoperatorene ($\neq$, $<$, $\leq$, $=$, $\geq$ og $>$). Måten vi har løst dette på, er ved å lage en eller flere makroer som genererer funksjonene. Dermed unngår vi å måtte vedlikeholde 36 nesten like funksjoner. Denne løsningen er langt fra optimal siden makroer samspiller dårlig med debuggere og lignende verktøy, samt at koden i selve makroen kan være tung å lese. Vi har ikke funnet noen gode alternativer, og i følge Kernighan & Pike [7, kapittel 9.6], er dette et tilfelle hvor bruk av makroer er en god løsning. De skriver:

> Throughout this book, we've cautioned against using macros [. . .] But they do have their place; sometimes textual substitution is exactly the right answer to a problem. One example is using the C/C++ macro preprocessor to assemble pieces of a stylized, repetitive program.
>
> [. . .] Macro processing can be used to generate production code, too.

Kapittel 9

Mangler og forbedringer

Vi har kommet langt, og har klart å oppfylle målene for hva denne versjonen skulle kunne gjøre. Likevel er plattformen langt fra komplett. Vi vil her oppsummere de vesentligste manglene, og peke på aktuelle forbedringer av plattformen.

- Indeksstrukturene vi har implementert er enkle, og støtter ikke dynamisk aktivitet. Dette medfører at alle endringer i databasen gir inkonsistenser i indeksene til Rindex. Følgelig må indeksene regenereres.
- Implementasjonen mangler støtte for relasjoner med attributter som ikke blir indeksert. Dette gjelder i hovedsak binærdata som bilder, lyd og film, men også store tekster.
- Det gjøres ingen forsøk på å optimalisere spørreplanen. Mange av optimaliseringsteknikkene som brukes av tradisjonelle DBMSer, vil også kunne brukes av Rindex. Unikt for Rindex er oppgaven med å minimalisere antall substitusjoner.
- Rindex bør støtte en større del av SQL standarden for å kunne uttrykke mer interessante spørringer. Spesielt gjelder dette en uforutsett begrensning; en relasjon kan ikke joines med seg selv på en naturlig måte, og for å få til dette må Rindex støtte spørringslokal navngivning (aliasing) av relasjoner.
- Rindex trenger en mer dynamisk tilnærming til minnehåndtering i indeksoperasjonene. Dette gir seg særlig utslag i forbindelse med joins, hvor det allokeres minneplass tilsvarende produktet av antall tupler i relasjonen som joines. Dette er et worst-case-estimat, og fører til et for høyt minneforbruk i de fleste tilfeller. I tillegg frigjøres ikke for mye allokert minne før spørringen er ferdig eksekvert.
- NULL-verdier håndteres ikke av Rindex. NULL-verdier som hentes fra en DBMS ved hjelp `SQLAPI++`-biblioteket blir oversatt til tallet 0 eller

en tom streng. Derfor blir ikke disse skilt fra andre attributtverdier.

- Rindex må sjekke attributtene nevnt i joinbetingelsen under semantikksjekken, se avsnitt 6.3.

- `librindex`skriver resultatet av spørringer bare til `stdout`, og dette gjør at det blir vanskelig for programmet som bruker `librindex`å ha kontroll med resultatet. En mer fleksibel løsning ville ha vært en mekanisme hvor resultatet ble hentet via metodekall (f.eks. ved bruk av iteratorer).

- Måten Rindex håndterer eksekveringsplanen på kunne vært ryddigere. For det første burde denne ikke vært et tre, men en liste. For det andre bør planen inneholde mer kunnskap om hva som konkret må gjøres, siden dette forenkler rutinene som tar seg av eksekveringen av hver node (særlig `JoinNode`).

- Det er ønskelig med noen brukerkommandoer for å sette diverse parametre i plattformen under kjøring, som for eksempel å skru av og på tidtagning og debug-informasjon.

- Algoritmen som bestemmer hvilke mellomlagringer som skal brukes som input og output til indeksoperasjonene i en plan, er ikke optimal. Se avsnitt 7.3 for en beskrivelse av algoritmen.

Del II

Substitusjonsoperatoren

Kapittel 10

Problemstilling

I denne delen av rapporten skal vi studere ulike problemstillinger med hensyn på effektiviteten til substitusjonsoperatoren som ble beskrevet i forrige del (se avsnitt 4.4.5 på side 20).

10.1 Substitusjonsoperatoren

Substitusjonsoperatoren er nødvendig siden andre operatorer krever at mellomlagringen de skal behandle, inneholder verdier fra ett bestemt attributt. Dermed vil det oppstå situasjoner der mellomlagringen som er resultatet av en operator, ikke inneholder attributtet som neste operator forventer som input. I slike tilfeller brukes substitusjonsoperatoren for å transformere denne mellomlagringen. Hvor mange substitusjonsoperatorer som er nødvendig og plasseringen av dem, er avhengig av eksekveringsplanen Rindex velger.

Operatoren har som oppgave å bytte ut attributtverdier i en mellomlagring[1] med attributtverdier fra en annen indeks. De nye attributtverdiene skal ha samme primærnøkkel som verdiene de er substituert med har. Den er implementert som funksjonen `substitute()` i klassen `IndexManager`. Substitusjonsalgoritmen gjennomløper attributtverdiene i en mellomlagring sekvensielt, og for hver verdi gjøres et oppslag for å finne den attributtverdien som har samme primærnøkkel i en annen indeks. For å gjøre dette oppslaget effektivt, er det nødvendig med en spesiell datastruktur, inverterte indekser, som støtter direkte oppslag (se avsnitt 4.2 på side 15).[2] De inverterte indeksene er altså en datastruktur hvor primærnøklene og attributtverdiene har byttet rolle i forhold til en indeksstruktur, slik at det er mulig å bruke en primærnøkkel for å finne attributtverdien. I Rindex, versjon 0.1, valgte vi å bruke hash-tabeller for å implementere en slik datastruktur (se avsnitt 4.2).

[1] En mellomlagring har, som tidligere nevnt, samme struktur som en indeks.

[2] Denne strukturen brukes også i projeksjonsoperatoren, men dette er ikke relevant i denne sammenhengen, bortsett fra at effektiviteten til projeksjonsoperatoren også er sterkt avhengig av de inverterte indeksene.

Etter at substitusjonen er utført, må den nye mellomlagringen sorteres med hensyn på attributtverdiene, siden de nye attributtverdiene er lagt inn i samme rekkefølge som primærnøklene i den opprinnelige mellomlagringen hadde.

10.2 Problemstillingene

Ved å kjøre en del tester med ulike spørringer mot Rindex, ble det klart at substistusjonsoperatoren stod for mellom 85% og 98% av den totale tiden som ble brukt på å eksekvere en spørring internt i programmet.[3] Kravet og ønsket om å forbedre effektiviteten til denne operatoren er årsaken til at jeg har valgt å fokusere på de to følgende problemstillingene:

1 *Hvordan kan selve substitusjonsalgoritmen forbedres?*
Bakgrunnen for å se nærmere på denne problemstillingen, er at denne algoritmen er den sentrale og tidkrevende komponenten i substitusjonsoperatoren. For å kunne besvare spørsmålet, er det altså nødvendig å studere og analysere substitusjonsalgoritmen som er brukt i Rindex.

2 *Hvordan kan antall substitusjonsoperatorer i en spørring reduseres?*
Dette er et naturlig spørsmål å stille, selv om det ikke gjelder en forbedring av selve substitusjonsoperatoren. En reduksjon i antallet slike operatorer vil forbedre effektiviteten til plattformen når det gjelder eksekveringen av en spørring. Siden substitusjonsoperatoren blir introdusert av Rindex, er det mulig at antall nødvendige substitusjonsoperatorerer kan reduseres. Dette ble antydet i avsnitt 7.3 på side 37.

Disse problemstillingene blir behandlet i hvert sitt kapittel, henholdsvis kapittel 11 og 12.

[3]Projeksjonsoperatoren er ikke tatt med i denne beregningen, siden utskrift til skjerm nødvendigvis tar lang tid. Dessuten vil det være naturlig i en fremtidig versjon av Rindex at projeksjonen ikke skriver resultatet direkte til skjermen, men heller returnerer en peker til resultatsettet slik at en klient kan iterere over resultatsettet etter behov.

Kapittel 11

Analyse av substitusjonsalgoritmen

Siden substitusjonsalgoritmen er spesielt avhengig av implementasjonen til de inverterte indeksene, vil vi her se på ulike måter å implementere disse på. Fra hvert av disse alternativene følger det en egen substitusjonsalgoritme. Både algoritmene og de ulike implementasjonsalternativene vil bli beskrevet i de neste avsnittene. Til slutt kommer et avsnitt hvor de sammenlignes teoretisk og empirisk.

Kompleksitetsfunksjonen: Hver substitusjonsalgoritme vil bli analysert med hensyn på tids- og plasskompleksiteten. For denne analysen vil ikke de tradisjonelle kompleksitetsfunksjonene (som $\mathcal{O}$) være tilstrekkelig detaljerte for å skille de ulike algoritmene.[1] Derfor er det i denne rapporten brukt en mer detaljert kompleksitetsfunksjon $\mathcal{C}$. For tidskompleksiteten uttrykker denne funksjonen maksimalt antall operasjoner algoritmen trenger for en inputstørrelse n, mens den for plasskompleksiteten uttrykker størrelsen på de ulike datastrukturene. Videre vil notasjonen $T(n)$ bli brukt som betegnelse for tidskompleksiteten og $S(n)$ for plasskompleksiteten.

11.1 Substitusjonsalgoritme 1: Inverterte indekser med hash-tabeller

Den første substitusjonsalgoritmen er identisk med den som er brukt i versjon 0.1 av Rindex. Denne algoritmen benytter seg av at de inverterte indeksene er implementert ved hjelp av hash-tabeller, for å kunne bruke dirkete oppslag.

Definisjon 11.1 (Hash-funksjon)**.** *En* hash-funksjon *er en funksjon som transformerer en input-verdi fra et (typisk) stort domene til en output-verdi*

[1] Faktisk vil alle algoritmene ha en tidskompleksitet på $\mathcal{O}(n)$.

fra et (typisk) mindre domene. Resultatet (det vil si output-verdien) fra hash-funksjonen kalles hash-verdi *eller* hash-nøkkel. *En god hash-funksjon er rask og distribuerer verdier fra det forventede input-domenet uniformt over output-domenet, slik at det oppstår færrest mulig kollisjoner*[2]*.*

Definisjon 11.2 (Hash-tabell)**.** *En* hash-tabell *er en datastruktur, en assosiativ array, der en hash-nøkkel kan brukes for å gjøre direkte oppslag i tabellen. Nøklene beregnes ved hjelp av en hash-funksjon. For oppslag i tabellen kan en hash-tabell ha en gjennomsnittlig/forventet tidskompleksitet på* $\mathcal{O}(1)$*, mens den i verste tilfellet (som er lite sannsynlig) kan bruke* $\mathcal{O}(n)$ *tid.*

En mer detaljert beskrivelse av de inverterte indeksene og bakgrunnen for valget av hash-tabeller som datastruktur, er beskrevet i avsnitt 4.2 på side 15.

For denne substitusjonsalgoritmen ser vi på to ulike måter å implementere hash-funksjonen på. Kollisjonshåndteringen i Rindex (det vil si bruk av kollisjonslister) blir ikke vurdert og sammenlignet mot andre metoder, for eksempel åpen adressering, lineær hashing og utvidbar hashing. Det er heller lagt vekt på selve hash-funksjonen fordi den er sentral for effektiviteten til hash-implementasjonen, uansett hvilken metode man velger for kollisjonshåndteringen.

Første avsnitt beskriver datastrukturen til de inverterte indeksene, mens neste avsnitt beskriver og analyserer substitusjonsalgoritmen (som er felles for de to hash-funksjonene). Deretter vil de to etterfølgende avsnittene beskrive hver sin versjon av hash-funksjonen, mens det siste avsnittet inneholder en sammenligning av dem.

11.1.1 Datastrukturen

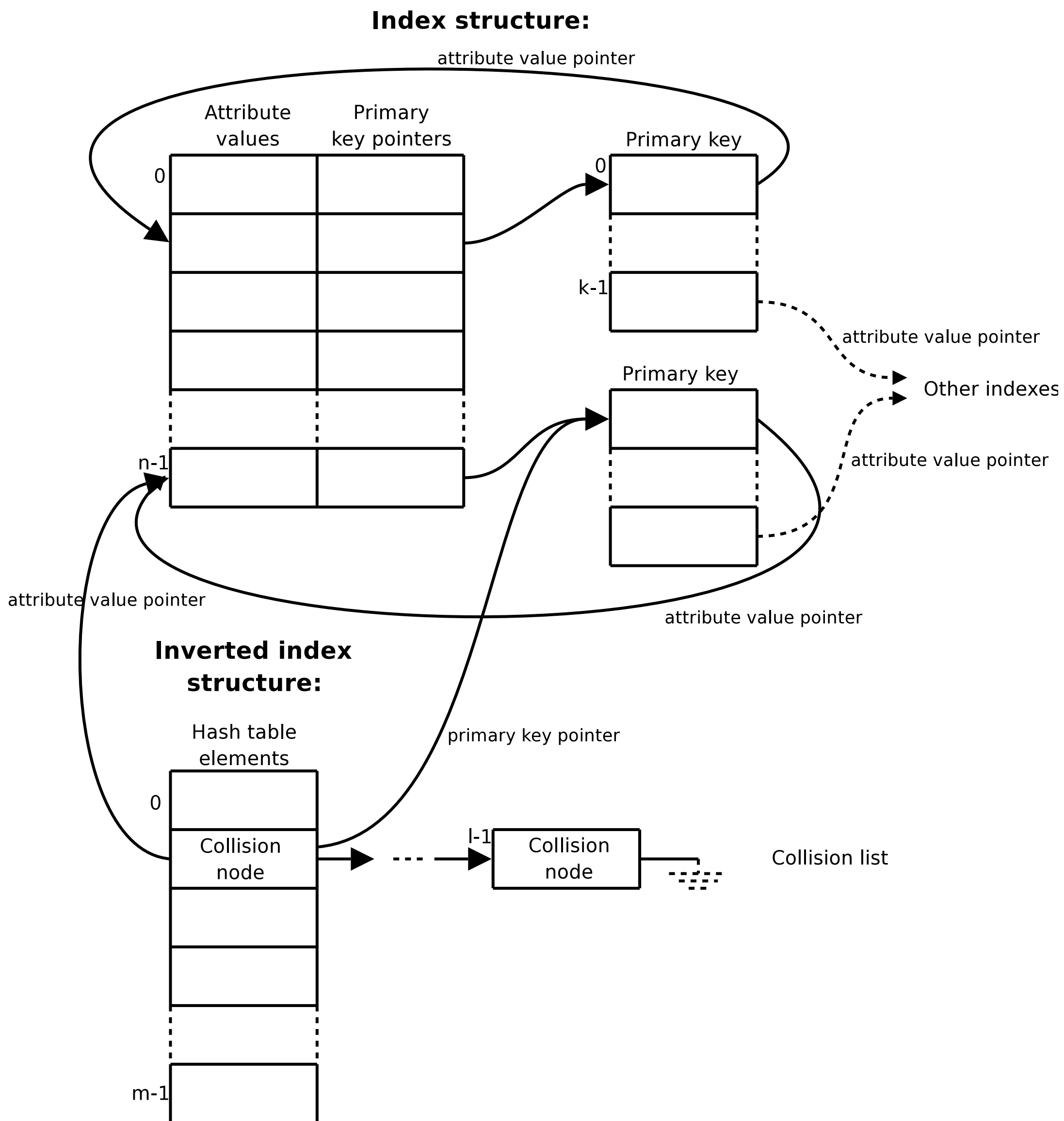
For hvert attributt som Rindex indekserer[3], er det også nødvendig å tilordne en invertert indeks. Figur 11.1 på neste side viser hvordan indeksene, de inverterte indeksene og deres tilhørende datastrukturer er bygd opp, samt relasjonene mellom dem.[4] I denne figuren (og i senere figurer) er lengdene på de ulike datastrukturene markert med variabelnavn. For eksempel betegner n lengden på arrayen med attributtverdiene. Disse variablene blir senere brukt i kompleksitetsanalysen.

Hver indeks består av en array (av lengde n) med attributtverdiene, og en like stor array med pekere til de tilhørende primærnøklene. Siden en primærnøkkel kan bestå av flere attributter, er hver primærnøkkel-struktur satt sammen av en mengde (av lengde k) attributt-pekere, også kalt elementer av

[2] En kollisjon oppstår dersom to ulike input-verdier får samme hash-verdi.

[3] Se avsnitt 4.3 på side 17 for hvilke attributter dette gjelder.

[4] Dette er en noe forenklet figur, hvor enkelte mindre implementasjonsdetaljer er utelatt. I tillegg er bare et representativt utvalg av pekerene illustrert.



Figur 11.1: Datastruktur for indekser og inverterte indekser for substitusjons-algoritme 1

primærnøkkelen videre i rapporten. Disse peker på attributtet i den tilsvarende indeksen, det vil si enten samme indeks eller andre indekser i samme relasjon.

Figuren viser også de inverterte indeksene, representert ved en hash-tabell (av lengde m). Hver slik tabell består av kollisjonslister, med gjennomsnittlig lengde l, som er implementert ved en enveis pekerkjede bestående av kollisjonsnoder. Det er ikke benyttet et eget hode for hver pekerkjede, men hver kollisjonsnode har en peker til en eventuell neste kollisjonsnode. I tillegg har hver node også pekere til primærnøkkelen og den tilsvarende attributtverdien den representerer. Primærnøkkelen er nødvendig å ha i noden for å kunne sjekke om den er lik den ønskede nøkkelen, siden alle noder i en kollisjonsliste vil ha samme hash-verdi.

I versjon 0.1 av Rindex valgte vi å la hver invertert indeks implementere en egen hash-tabell, slik figuren viser, og ikke la indeksene dele én felles stor tabell. Dette er det flere grunner for. En av dem er at tabellene da er uavhengige og kan derfor bedre og enklere tilpasses det tilhørende attributtet:

- Tabellstørrelsen kan tilpasses behovet til hvert attributt. Mindre tabellstørrelser betyr at det er større sannsynlighet for å finne store nok sammenhengende minneblokker ved allokering, spesielt når det ledige minneområdet er fragmentert.

- Hash-funksjonen kan tilpasses attributttypen slik at den distribuerer verdiene best mulig (det vil si mest mulig uniformt). En slik spesialisert hash-funksjon kan ofte være bedre enn en generell. Dette vil igjen kunne føre til at lengdene på kollisjonslistene reduseres. Denne muligheten er ikke studert videre her.

En annen grunn er at operasjonen for å hente ut elementer fra hash-tabellen blir enklere, fordi det blant annet ikke er nødvendig å teste at attributtet tilhører den ønskede indeksen. Dette vil også effektivisere substitusjonsalgoritmen noe.

11.1.2 Algoritmen

Substitusjonsalgoritmen kan beskrives med pseudo-koden[5] som er vist i algoritmelisting 1 på neste side. Den ytre løkken itererer over alle n attributtverdier som skal substitueres ut i indeksen (eller mellomlagringen). Den tilhørende primærnøkkelen brukes deretter for å beregne hash-verdien.[6] Denne verdien brukes til å slå opp i hash-tabellen, og resultatet av dette er en kollisjonsliste. I neste løkke gjennomløpes kollisjonslisten, en node om gangen,

[5] I pseudo-koden brukes $\leftarrow$ for tilordning og $\rightarrow$ for å referere til et medlem i en datastruktur. Koden antar normal eksekveringsrekkefølge og presendens. Dette betyr at høyresiden i en tilordning beregnes først.

[6] Hash-funksjonen $hash\,sum()$ forklares i de neste avsnittene, 11.1.3 og 11.1.4.

Substitusjonsalgoritme 1

```
foreach attribute value in index do
    hash value ← hash sum(primary key)
    collision list ← hash table[hash value]
    foreach node in collision list do
        foreach key in primary key do
            if same relation(key, toindex) then
                foreach key2 in node → primary key do
                    if memory location(key) = memory location(key2)
                    then
                        break
                    else
                        continue with next node
                    end
                end
            end
        end
        attribute value ← node → attribute value
        break
    end
end
```

inntil riktig node er funnet. En node er riktig dersom den inneholder de samme elementene i primærnøkkelen som originalattributtet. For å utføre denne sjekken, må man gjennomløpe og sammenligne elementene *key* i primærnøkkelen som tilhører den gjeldende attributtverdien, med elementene key_2 i primærnøkkelen noden peker på. Denne sammenligningen innebærer først en test på om *key* tilhører samme relasjon som indeksen det nye attributtet skal ligge i (*toindex*). Funksjonen *same relation*() er brukt til dette. Grunnen til at en slik test er nødvendig, er fordi primærnøkkelstrukturen, etter en joinoperasjon, vil inneholde elementer fra ulike relasjoner.[7] Deretter utføres selve sammenligningen ved å sjekke om *key* og key_2 peker på det samme minneområdet. Det er ikke nødvendig å teste på likhet på selve attributtverdiene for de to elementene (attributtpekerne), siden pekerne nødvendigvis må peke på samme minneområde for å inneholde samme attributtverdi (se figur 11.1). Testen på pekerlikhet er valgt fordi den er like effektiv som tester på verdilikhet for heltall, og langt mer effektiv enn tester på verdilikhet for andre datatyper, spesielt for tekststrenger. Dersom testen på pekerlikhet slår til, kan man hoppe ut av den indre løkken og fortsette med å teste neste *key*-element.[8] Hvis testen ikke slår til, innebærer dette at elementene

[7] Det er ikke nødvendig å utføre tilsvarende test for key_2, fordi denne hentes fra den gjeldende kollisjonsnoden, og denne vet vi er fra samme relasjon som indeksen (og den påvirkes heller ikke av joinoperasjoner).

[8] Vi antar her at *key*-elementene er ordnet slik at elementene fra samme indeks ligger sekvensielt og med samme rekkefølge som i de inverterte indeksene. Denne antagelsen er

i attributtverdiens tilhørende primærnøkkel, ikke finnes blant elementene i primærnøkkelen den gjeldende noden peker på. Derfor må man forsette med neste node i kollisjonslisten.

Til slutt vil man komme i en situasjon der alle *key*-elementene har et tilsvarende key_2-element, og dette betyr at algoritmen har funnet riktig node. Dermed kan attributtet som noden peker på, erstatte det gjeldende originalattributtet i mellomlagringen. Deretter forsetter algoritmen med neste attributtverdi.

Implementasjonen av algoritmen er en optimalisert versjon av pseudokoden, og den er utvidet slik at den er robust mot ulik rekkefølge på elementene i primærnøkkelen.

Tidskompleksitet

Algoritmen har en tidskompleksitet på $T(n) = \mathcal{C}(n * (h + l * k * k_2)) = \mathcal{C}(n * h + n * l * k * k_2)$, der n er antall attributtverdier i indeksen, l den gjennomsnittlige lengden på kollisjonslisten, h kompleksiteten til hash-funksjonen (som i verste tilfellet er tilnærmet lik k, se de to avsnittene som beskriver hash-funksjonene) og k og k_2 lengdene på primærnøklene for henholdsvis indeksen og kollisjonsnoden. Lengden k er markert i figur 11.1, mens k_2 ikke er illustrert i figuren fordi primærnøkkelstrukturen den representerer nødvendigvis ligger i en annen indeks (ellers hadde ikke substitusjonen vært nødvendig). Størrelsene k og k_2 er som regel små, avhengig av definisjonen av relasjonene og antall joinledd i spørringen. Den avgjørende faktoren i effektiviteten til algoritmen er den gjennomsnittlige lengden (l) på kollisjonslisten. De andre størrelsene er bare avhengige av definisjonene av relasjonene i databasen, og de må derfor uansett gjennomløpes ved denne implementasjonen. Dette betyr at det er viktig for hash-funksjonen å distribuere nøklene mest mulig uniformt over hash-tabellen, slik at kollisjonslisten blir kortest mulig (det vil si en lavest mulig l-verdi).

Siden indeksene i versjon 0.1 av Rindex er statiske og alle nøkler derfor er kjent på forhånd, ville det være mulig å konstruere en perfekt hash-funksjon [8, avsnitt 6.4 (s. 513)]. Med en slik funksjon ville det vært mulig å garantere at kollisjoner aldri finner sted og dermed også en tidskompleksitet på $\mathcal{O}(1)$ for oppslaget.[9] En mulighet for å konstruere en slik funksjon er å bruke `gperf`[10], som er et verktøy for å generere perfekte hash-funksjoner. Denne løsningen er ikke vurdert videre i rapporten, siden kravet om at indeksene skal være statiske ikke vil gjelde for fremtidige versjoner av Rindex.[11]

---

en forenkling, men gjelder ikke for implementasjonen av algoritmen.

[9] Genereringen av en slik perfekt hash-funksjon krever $\mathcal{O}(n)$ operasjoner.

[10] `http://www.gnu.org/software/gperf`

[11] Det er ikke noe absolutt krav til at de må være statiske for å kunne benytte en perfekt hash-funksjon, men da må denne funksjonen og alle hash-verdiene beregnes på nytt for hver gang det skjer en endring i indeksene, og for de fleste DBMSer vil dette være uakseptabelt.

Plasskompleksitet

Selve substitusjonsalgoritmen bruker ikke noe ekstra minne. Derimot vil substitusjonsoperatoren som bruker algoritmen, opprette en ny mellomlagring med de nye attributtverdiene som algoritmen finner. Denne mellomlagringen kan enten erstatte en annen mellomlagring, eller den kan legges til. Dette er vist i avsnitt 7.3 på side 37 og det tilhørende eksempelet 7.3. Det er derfor interessant å se på plassbruken til datastrukturen beskrevet tidligere i avsnittet. Hver indeks har en plasskompleksitet på $\mathcal{C}(n + n * k)$, der n er antall attributtverdier i indeksen og k antall elementer i hver primærnøkkel. For hver indeks blir det også generert en invertert indeks. Denne har en plasskompleksitet på $\mathcal{C}(p * l + m - p)$, der m er størrelsen på hash-tabellen, l den gjennomsnittlige lengden på kollisjonslistene og p antall posisjoner i hash-tabellen som inneholder minst én kollisjonsnode. Vi ser at $\mathcal{C}(p * l) = \mathcal{C}(n)$. Dersom vi for enkelhets skyld ikke tar hensyn til at eventuelle tomme posisjoner i hash-tabellen vil bruke plass i minnet, ser vi at $\mathcal{C}(n+m-p) = \mathcal{C}(n)$. Til sammen gir dette $S(n) = \mathcal{C}(n + n * k) + \mathcal{C}(n) = \mathcal{C}(2n + n * k)$ for indeksene som genereres ved oppstart av databasen. Mellomlagringer trenger ikke å opprette en ny invertert indeks og for disse er minnebruken derfor $S(n) = \mathcal{C}(n + n * k)$.

11.1.3 Hash-funksjonen, versjon 1

Denne funksjonen er brukt i versjon 0.1 av Rindex. Den er mer detaljert beskrevet i avsnitt 4.2 på side 15. Funksjonen kan også uttrykkes med pseudokode på følgende måte:

function *hash sum*(*primary key*)
  hash value ← 0
  foreach *key* **in** *primary key* **do**
    if *same relation*(*key*, *toindex*) **then**
      hash value ← *hash value* + *memory location*(*key*)
    end
  end
  return *hash value* mod *table size*
end function

Funksjonen mottar argumentet (*primary key*) med strukturen til primærnøkkelen som hash-funksjonen skal operere på. Hvert nøkkelelement gjennomløpes, og dersom det tilhører samme relasjon som indeksen det nye attributtet skal ligge i (*toindex*)[12], legges elementets minneadresse til den eksisterende hash-verdien. Til slutt brukes modulo-operasjonen for å sikre at hash-verdien

[12]Testen, ved kallet på *same relation*()-funksjonen, må utføres av samme grunn som beskrevet for substitusjonsalgoritme 1.

er innenfor lengden (*table size*, det vil si m) av hash-tabellen. Dette betyr at hash-funksjonen fordeler hash-nøklene uavhengig av datatypene på input-verdiene, siden den kun ser på minneadressene. Den egner seg derfor spesielt godt dersom disse minneadressene ligger sekvensielt i minnet.

Siden hver minneadresse bare blir lagt sammen og hver adresse tar 4 byte (på x86 arkitekturer), vil algoritmen hoppe over 4 tall i output-domenet for hver av verdiene som blir lagt sammen. En variant av denne funksjonen vil derfor dividere hver minneadresse på 4 for å benytte alle verdier i domenet. Denne varianten er ikke studert videre i rapporten. Grunnen er at den vil gi lik distribusjon av nøklene som hash-funksjon 1, når tabellstørrelsen blir satt dynamisk (se under).

Det viste seg tidlig at de inverterte indeksene forårsaket flaskehalsen i substitusjonsalgoritmen. Årsaken til dette var at hash-funksjonen distribuerte nøklene dårlig over tabellen.[13] Hovedårsaken til denne dårlige distribusjonen, var at tabellstørrelsen var satt statisk ved kompilering av programmet. En slik fast størrelse på tabellen vil ikke passe like godt for alle relasjonsstørrelser. Dette betyr at det for noen relasjoner allokeres større tabeller enn nødvendig, slik at deler av tabellen blir stående ubrukt, mens det for andre relasjoner blir mange kollisjoner i tabellen fordi den er for liten. For å forbedre distribusjonen, er det derfor nødvendig å sette tabellstørrelsen dynamisk, men med en bestemt maksstørrelse[14]. Dette kan gjøres på minst to måter:

1 Tabellstørrelsen settes lik relasjonsstørrelsen.

2 Tabellstørrelsen settes lik et primtall (for eksempel et i nærheten av, men helst større enn relasjonsstørrelsen).

Den første løsningen er enkel å implementere, men den er ikke optimal siden tabellstørrelsen brukes i modulo-operasjonen i hash-funksjonen, for at hash-verdien som er regnet ut skal passe inn i tabellen. Spesielt dårlige valg for tabellstørrelsen er partall, multipler av 3 og potenser av maskinens basis-tall (radix). Knuth [8, avsnitt 6.4 (s. 516)] sier mer generelt om tabellstørrelsen M:

> In general, we want to avoid values of M that divide $r^k \pm a$, where k and a are small numbers and r is the radix of the alphabetic character set (usually $r = 64$, 256 or 100), since a remainder modulo such a value of M tends to be largely a simple superposition of the key digits.

og videre

[13] Avhengig av tabellstørrelsen og relasjonene som ble brukt, lå fyllingsgraden på mellom 5% og 25%.

[14] Maksstørrelsen er satt til 100 000 i testene og måledataene som presenteres i denne rapporten.

> Such considerations suggest that we *choose M to be a prime number* such that $r^k \not\equiv \pm a$ (modulo M) for small k and a. This choice has been found to be quite satisfactory in most cases.

Dette taler for at alternativ 2 er det beste valget for å få en mest mulig uniform distribusjon av nøklene. Tabellen i avsnitt 11.1.5 på neste side viser hvordan de ulike hash-funksjonene som omtales i denne rapporten påvirkes av tabellstørrelsen, og hvor bra de utnytter tabellen. Som forventet er det stor forskjell i distribusjon av nøklene ved bruk av relasjonsstørrelsen eller et passende primtall som tabellstørrelse.

Et problem med alternativ 2 er å bestemme primtallet effektivt, selv om det for små tall ikke vil være et stort problem, dersom det skal være dynamisk satt i forhold til relasjonsstørrelsen. Det å teste om et tall er et primtall eller ikke, ble først i 2002 vist å være et P-problem, da det ble presentert en algoritme som bruker polynomisk tid for problemet.[15] Det eksisterer også effektive algoritmer som bygger på sannsynlighetsregning. Slike algoritmer benytter sofistikerte teknikker som *nesten* alltid returnerer et svar, men ikke med en absolutt matematisk sikkerhet. En kjent, meget effektiv og mye brukt sannsynlighetsalgoritme er "Rabin-Miller strong pseudoprime test", som blant annet brukes i Mathematica[16]. En annen mulighet er å ha en tabell med en del forhåndsgenererte primtall, og bruke denne for å finne et passende primtall. Denne løsningen kan ha en ulempe dersom det er mange primtall i tabellen og denne skal gjennomløpes ofte for å finne et primtall. I Rindex vil ikke dette være et problem, siden en tabellstørrelse ikke kan være større enn maksstørrelsen og fordi tabellstørrelsen bare blir bestemt én gang, ved oppstart av databasen. På bakgrunn av dette er denne løsningen valgt.

Det å endre tabellstørrelsen fra å være et statisk tall til et dynamisk primtall vil antageligvis bety mye kortere kollisjonslister og dermed også en lavere l-verdi i kompleksiteten til substitusjonsalgoritmen.

Originalversjonen av denne substitusjonsalgoritmen bruker altså hashtabeller med en forhåndsbestemt, statisk størrelse (satt til primtallet 1009), og denne algoritmen vil senere bli referert til som substitusjonsalgoritme 0. Den har samme tids- og plasskompleksitet som substitusjonsalgoritme 1, men vil som regel ha en mye større l-verdi.

11.1.4 Hash-funksjonen, versjon 2

Det vil også være interessant å sammenligne den egenproduserte hash-funksjonen fra forrige avsnitt med andre som er mye brukt, gjerne i eksisterende DBMSer. Et eksempel på en slik funksjon er beskrevet av Robert John

[15] Dette ble gjort av Agrawal, Kayal og Saxena [1] og kompleksiteten til denne algoritmen er $\mathcal{O}(ln^{12}n)$.

[16] `http://www.wolfram.com/products/mathematica`

Jenkins Jr. [6], og den er blant annet brukt i DBMSen PostgreSQL[17] og i deler av Linux-kjernen[18]

Funksjonen tar et input med variabel lengde, i form av en array av bytes, og returnerer et 32-bits heltall som hash-verdien. I funksjonen vil ethvert bit i input-verdien påvirke alle bits i hash-verdien. Ifølge dokumentasjonen skal funksjonen ha en tidskompleksitet på $36 + 6m$ instruksjoner, der m er antall bytes i input-verdien.

Denne funksjonen fungerer best når tabellstørrelsene er på formen 2^x, altså ikke et primtall. Grunnen til dette er at den ikke krever bruk av modulo-operasjonen for at hash-verdien skal ligge innenfor lengden på hash-tabellen. I stedet kan man bruke `and`-operatoren på hash-verdien og en bit-maske for å hente ut det antall bits man trenger fra hash-verdien. Dette er ikke gjort her, fordi det ville innebære en omstrukturering både med tanke på hash-funksjonen og tabellstørrelsene. I tillegg ville ikke en slik endring ha betydning for distribusjonen til funksjonen, bare på hastigheten (fordi modulo-instruksjonen er relativt treg).

Andre velkjente og mye brukte hash-funksjoner inkluderer blant annet "Fowler/Noll/Vo (FNV) hash" [3] og "Daniel J. Bernstein's djb2 hash" [2], men disse er ikke studert videre her.

11.1.5 Sammenligning av hash-funksjonene

Tabellen under viser hvor bra de ulike hash-funksjonene utnytter hash-tabellen (fyllingsgrad) og forventet antall kollisjoner i tabellen. Målingene er gjort mot filmdatabasen [12] (databasen er mer beskrevet i avsnittet om testing, 11.5.2).

	Relasjonsstørrelse		Nærmeste primtall	
	Fyllingsgrad	Kollisjoner	Fyllingsgrad	Kollisjoner
Hash-funksjon 1	0.241955	4.29917	0.99947	0.47312
Hash-funksjon 2	0.711296	0.80257	0.74142	0.98584

Tabell 11.1: Fyllingsgrad og forventet antall kollisjoner for de ulike hash-funksjonene

Tabellen viser at fyllingsgraden blir veldig høy for hash-funksjon 1 når tabellstørrelsen settes til et passende primtall. Gjennomsnittet av forventet antall kollisjoner dras kraftig opp av de relasjonene som er større enn den forhåndsdefinerte maksgrensen for tabellstørrelsen. Dersom man ser bort fra disse relasjonene, vil forventet antall kollisjoner være 0, det vil si at ingen av tabellene vil ha kollisjoner. Også i de relasjonene som er større enn maksgrensen vil nøklene bli tilnærmet uniformt fordelt. Dette viser at hash-funksjon 1

[17] `http://www.postgresql.org`

[18] Fra versjon 2.6.9-rc3, se `http://kernel.org/pub/linux/kernel/v2.6/testing/ChangeLog-2.6.9-rc3`.

fungerer svært godt for denne databasen. Siden funksjonen ikke tar hensyn til attributtverdiene, men bare minneadressene, kan vi forvente at funksjonen vil fungere like godt for andre databaser.

Hash-funksjon 2 gir ikke like høy fyllingsgrad og har en noe høyere forventet lengde på kollisjonslistene når tabellstørrelsen er et primtall. Den gjør det derimot bedre enn hash-funksjon 1 når relasjonsstørrelsen brukes som tabellstørrelsen. Funksjonen er også mer generell enn funksjon 1, i det den sistnevnte utnytter at datastrukturen og bruken av funksjonen er kjent.

En eventuell forskjell i tidskompleksitet mellom algoritmene som benytter disse hash-funksjonene, vil først og fremst ligge i den forventede lengden av kollisjonslisten l. Derfor er distribusjonen til hash-funksjonen de benytter essensiell. Som tabellen viser, er det hash-funksjon 1 som distribuerer best og som vi dermed kan forvente at vil gi den beste ytelsen. Derfor er hash-funksjon 1 brukt i tester og målinger videre i rapporten.

11.2 Substitusjonsalgoritme 2: Inverterte indekser i hver primærnøkkel

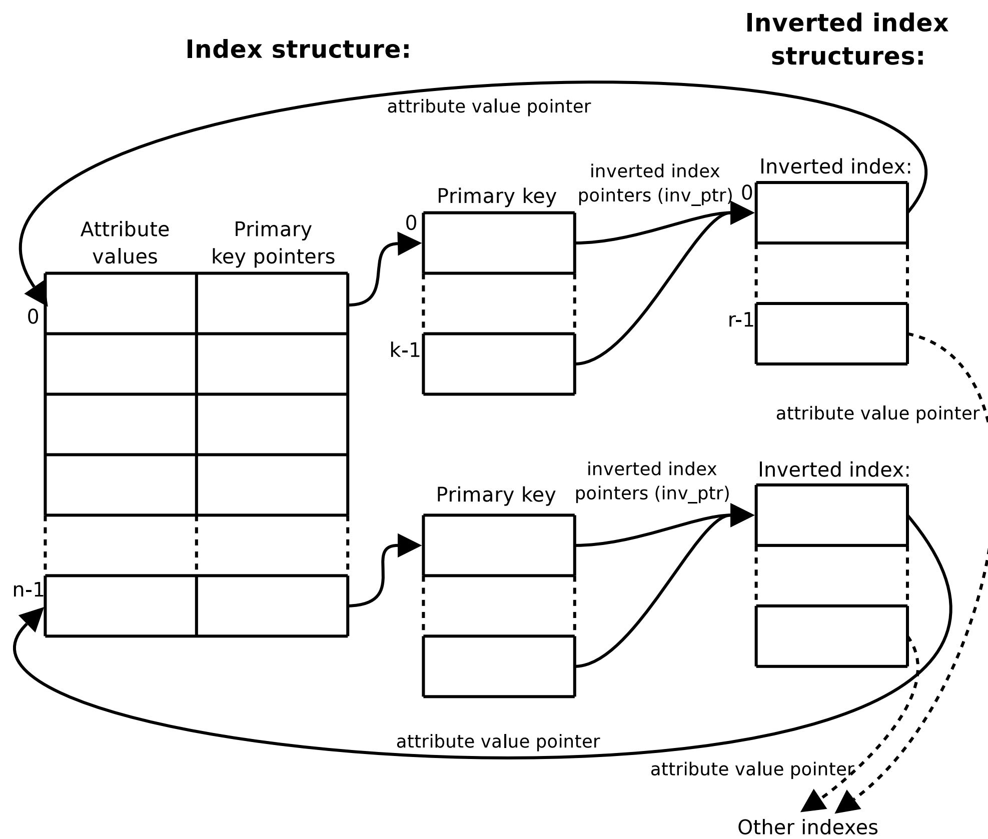
De resterende forslagene til substitusjonsalgoritmer, presentert i dette og de neste avsnittene, benytter andre datastrukturer enn hash-tabeller for de inverterte indeksene. Bakgrunnen for bruken av hash-tabeller var ønsket om å, utfra en primærnøkkel, kunne utføre et direkte oppslag for å finne den attributtverdien som skal substitueres inn. Etter hvert ble det klart at slike direkte oppslag kunne gjøres mer effektivt ved hjelp av pekerstrukturer, det vil si datastrukturer bestående av bare pekere. Hovedidéen som de neste avsnittene vil basere seg på, er at hver attributtverdi i en indeks har en peker til én bestemt posisjon i datastrukturen til de inverterte indeksene. I denne strukturen vil det deretter være mulig å følge nye pekere for å finne den ønskede attributtverdien.

Her beskrives første versjon av en slik pekerstruktur, og deretter den nye substitusjonsalgoritmen som følger av denne.

11.2.1 Datastrukturen

De inverterte indeksene kan implementeres ved å splitte dem i mindre deler, og la hver primærnøkkel i indeksen ha en peker til én slik del. Ved å la hvert attributt ha én bestemt posisjon innenfor relasjonen, er det mulig å implementere hver av disse små inverterte indeksene som en array med like mange elementer som antall attributter (indekser) r i den tilhørende relasjonen.[19] Figur 11.2 på neste side viser en slik datastruktur. For hver posisjon i arrayen, er det en peker til den attributtverdien som har samme primærnøkkel

[19] De attributtene som ikke har blitt indeksert av Rindex, er ikke med i dette antallet.



Figur 11.2: Datastruktur for indekser og inverterte indekser for substitusjonsalgoritme 2

og som ligger i indeksen med den tilsvarende attributtposisjonen i samme relasjon.

Som figuren også viser, inneholder hvert element i primærnøkkelen en peker til samme struktur (invertert indeks). Dette er unødvendig, men er gjort for å slippe å endre på strukturen og implementasjonen av primærnøklene da dette er en ganske omfattende oppgave, siden de fleste andre operasjoner i indekshåndteringen berøres av dette. Se avsnitt 11.4 på side 66 for forslag til en slik løsning.

11.2.2 Algoritmen

Substitusjonsalgoritmen som følger ut fra den nye datastrukturen for de inverterte indeksene, er vist i algoritmelisting 2. Algoritmen er betraktelig enklere enn substitusjonsalgoritme 1 på side 55, og dette medfører at vi kan forvente en forbedring i effektiviteten. Den ytre løkken er alltid nødvendig, fordi den

Substitusjonsalgoritme 2

```
foreach attribute value in index do
    foreach key in primary key do
        if same relation(key, toindex) then
            attribute value ←
                key → inv_ptr[toindex → attr_pos] → attribute value
        end
    end
end
```

gjennomløper hvert element som skal byttes ut. Den indre løkken, derimot, er bare nødvendig i det tilfellet der substitusjonen utføres på en mellomlagring som er resultatet av en tidligere joinoperasjon. En slik joinoperasjon fører til at primærnøkkelen blir satt sammen av elementene fra to datasett (mellomlagringer eller permanente indekser). Algoritmen må derfor lete etter første element som tilhører den riktige relasjonen (derav if-testen og kallet på *same relation*() i algoritmen).

Den nye attributtverdien hentes ut fra den inverterte indeksen som elementet i primærnøkkelen peker på ($key \rightarrow inv_ptr$). Oppslaget i denne arrayen kan gjøres direkte. Dette er mulig fordi vi vet hvilken indeks attributtet skal ligge i (*toindex*), og denne indeksen inneholder en variabel (*attr_pos*) med posisjonen den har i relasjonen. Dermed kan denne posisjonen brukes for direkte oppslag i datastrukturen som representerer den inverterte indeksen.

Tidskompleksitet

Kompleksiteten til denne algoritmen er $T(n) = \mathcal{C}(n * k)$, der n er antall attributtverdier i indeksen og k antall elementer (det vil si antall attributter) i primærnøkkelen. Dersom det ikke forekommer en join i spørringen, vil den

indre løkken stoppe etter første gjennomløp av elementene i primærnøkkelen og kompleksiteten blir dermed $\mathcal{C}(n)$. Algoritmen er i dette tilfellet optimal med hensyn på tidskompleksiteten.

Plasskompleksitet

Selve algoritmen bruker ikke noe ekstra minne, men den nye attributtverdien blir lagt inn i en ny mellomlagring, allokert av substitusjonsoperatoren. De endringene som her er gjort for de inverterte indeksene, vil også påvirke indeksstrukturen siden hver primærnøkkel vil ha en egen invertert indeks. Derfor vil hver indeks ha en total plasskompleksitet på $S(n) = \mathcal{C}(n+n*k*r)$, der r, som vist i figur 11.2, er lengden på hver av de små inverterte indeksene (det vil si antall indekserte attributter i den tilhørende relasjonen).

11.3 Substitusjonsalgoritme 3: Inverterte indekser for hver relasjonen

Denne substitusjonsalgoritmen er nært beslektet med algoritme 2 fra forrige avsnitt. Forskjellen mellom dem er bare datastrukturen til de inverterte indeksene de baserer seg på.

11.3.1 Datastrukturen

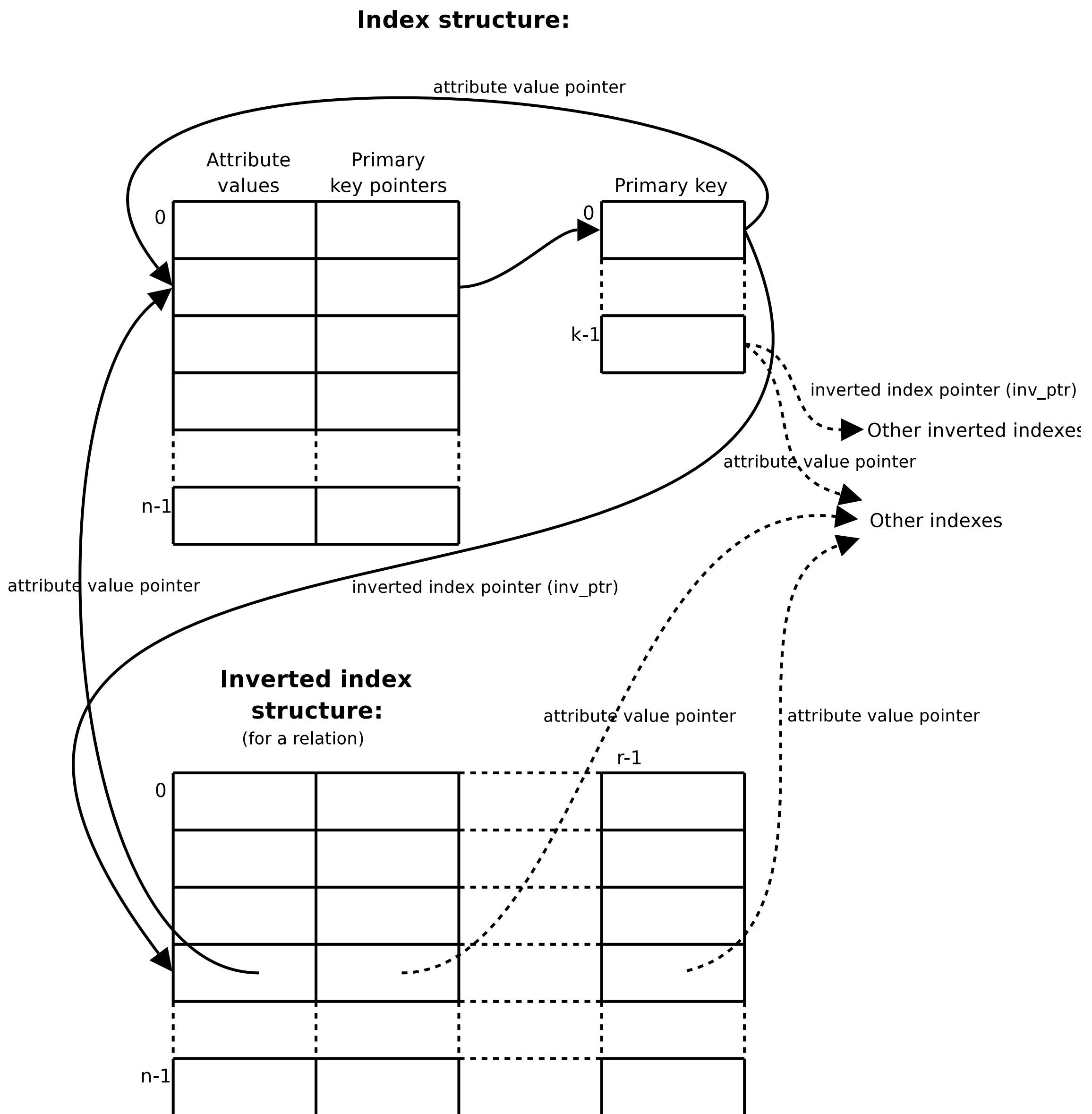
Datastrukturen for algoritme 2 lider, som nevnt, av en del dobbeltlagring av data. Hver indeks som tilhører samme relasjon, oppretter de samme inverterte indeksene. Dette er det mulig å unngå ved å la hver *relasjon*, i motsetning til hvert *attributt*, ha en datastruktur for de inverterte indeksene. Figur 11.3 på neste side viser denne nye datastrukturen, og hvordan elementetene i primærnøklene nå peker inn i denne. Figuren viser at datastrukturen kan implementeres ved hjelp av en todimensjonal array, der den ene dimensjonen får lengden n (antall attributtverdier), mens den andre dimensjonen får lengden r (antall indekserte attributter i relasjonen). Hver *rad* i denne nye datastrukturen, representerer samme datastruktur som den brukt for de små inverterte indeksene i algoritme 2 (se figur 11.2).

11.3.2 Algoritmen

Substitusjonsalgoritme 3 blir lik algoritme 2. Forskjellen er at i denne algoritmen, vil *inv_ptr* peke på en rad i den nye datastrukturen, mens den pekte på en egen array for hver indeks i den forrige substitusjonsalgoritmen.

Tidskompleksitet

Tidskompleksiteten blir som forventet den samme som for algoritme 2, det vil si $T(n) = \mathcal{C}(n * k)$, der n er antall attributtverdier i indeksen og k antall



Figur 11.3: Datastruktur for indekser og inverterte indekser for substitusjons-algoritme 3

Substitusjonsalgoritme 3

```
foreach attribute value in index do
    foreach key in primary key do
        if same relation(key, toindex) then
            attribute value ←
                key → inv_ptr[toindex → attr_pos] → attribute value
        end
    end
end
```

elementer i primærnøkkelen.

Plasskompleksitet

Minnebruken for hver indeks vil bli noe redusert i forhold til forrige algoritme, siden de inverterte indeksene her er felles for alle indeksene i samme relasjon og noe dobbeltlagring dermed unngås. Dette gir kompleksiteten $\mathcal{C}(n + n * k)$ for indeksene og $\mathcal{C}((n * r)/r) = \mathcal{C}(n)$ for de inverterte indeksene, dersom vi antar en lik fordeling av plassbruken mellom indeksene i relasjonen. Her er r antall indekserte attributter i den tilhørende relasjonen (se figur 11.3). Totalt vil plasskompleksiteten bli $S(n) = \mathcal{C}(n + n * k) + \mathcal{C}(n) = \mathcal{C}(2n + n * k)$.

11.4 Substitusjonsalgoritme 4 og 5: Omstrukturering av primærnøklene

Det eksisterer, som nevnt tidligere, fremdeles minst to områder der algoritmen ennå kan forbedres:

1 Effektivitet: Det kan være mulig å forbedre eller unngå den indre løkken.

2 Minnebruk: Siden alle indekser i samme relasjon har de samme primærnøklene, betyr dette at disse nøklene dupliseres i hver indeks innen samme relasjon.

Det vil bli presentert forslag på to mulige løsninger for begge disse områdene, i form av substitusjonsalgoritme 4 og 5. De har samme datastruktur, men med en viktig forskjell i implementasjonen av datastrukturen. En konsekvens av å ta i bruk disse løsningene, er at de kan ha positiv eller negativ innvirkning på alle andre operasjoner som bruker indekser og mellomlagringer i `IndexManager`.

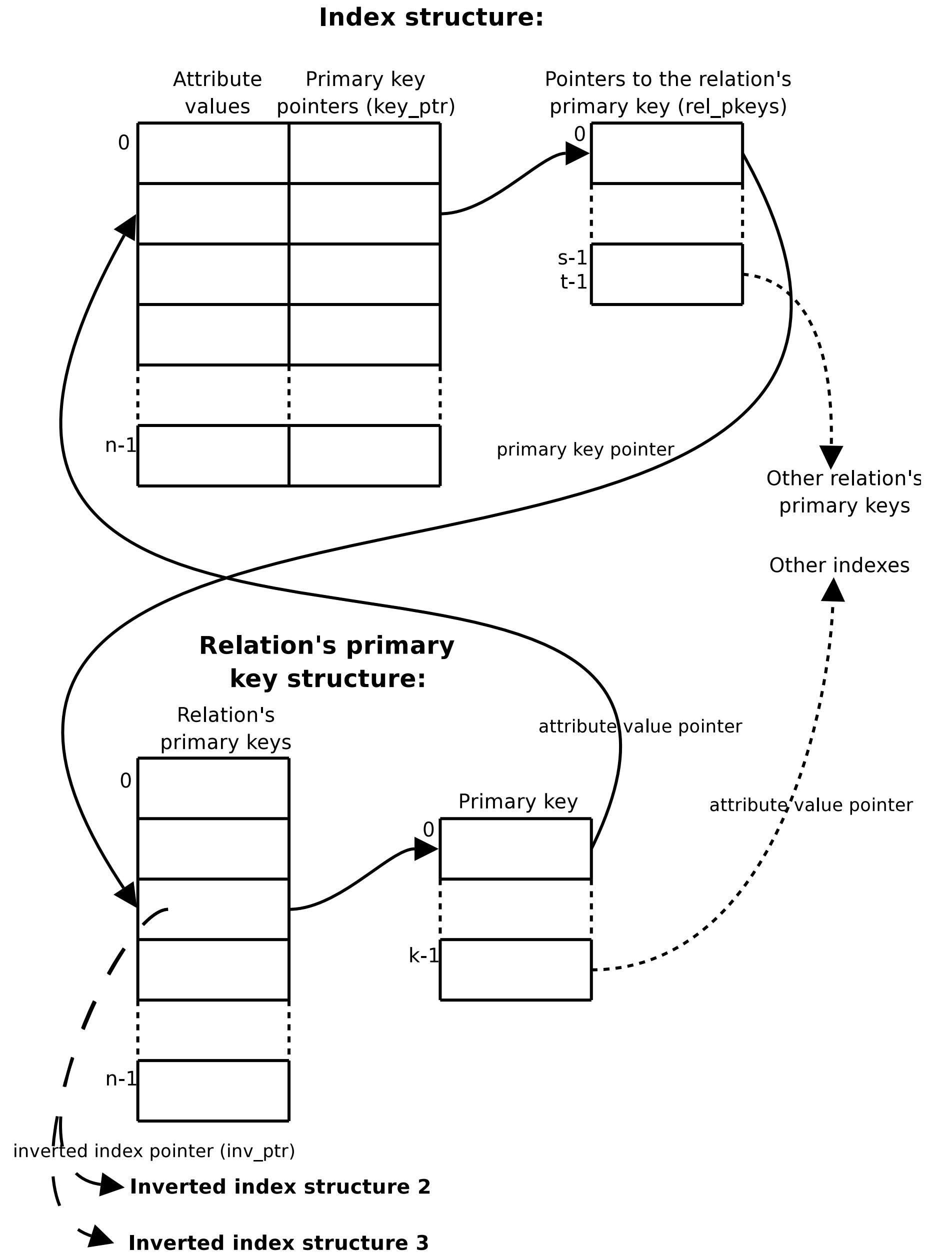
11.4.1 Datastrukturene

Hvert av forbedringspunktene som er beskrevet over, vil innebære en omstrukturering av primærnøklene. Vi observerer at eneste forskjell mellom primærnøklene i indeksene som tilhører samme relasjon, er rekkefølgen. Dette betyr at arrayen med pekere til primærnøklene må beholdes i hver indeks, mens selve nøklene kan lagres felles for hver relasjon. Figur 11.4 på neste side viser et forslag for en slik omorganisering. Indeksene inneholder pekere (*key_ptr*) til arrayer (*rel_pkeys*) som består av pekere til relasjonens primærnøkler. Denne arrayen kan implementeres på minst to ulike måter, og dette utgjør forskjellen mellom datastrukturene til algoritme 4 og 5:

For substitusjonsalgoritme 4: Arrayen (*rel_pkeys*) er av lengde én for alle indekser og for alle mellomlagringer som ikke inneholder resultatet av en tidligere joinoperasjon. I figuren er denne lengden markert med s. Siden arrayen inneholder pekere til relasjonens primærnøkler, er lengden avhengig av antall relasjoner som har vært involvert i spørringen på det tidspunktet mellomlagringen opprettes. Den vil derfor ofte bruke mindre plass enn den tilsvarende arrayen (med elementene i primærnøkkelen) i de tidligere løsningene, der strukturen er av lengde k. Forholdet mellom k og s er alltid $s \leq k$ og $k = \sum_{i=1}^{s} k_i$, der k_i er antall elementer i primærnøkkelen til relasjon i (for de relasjoner som deltar i spørringen). Selv om arrayen med pekere til relasjonens primærnøkler kommer i tillegg, vil antageligvis denne plassbruken oppveies av plassbesparelsen ved å la primærnøklene være felles for alle indeksene i relasjonen. Reduksjonen i størrelsen av s i forhold til k vil medføre at den indre løkken i substitusjonsalgoritmen ikke trenger like mange gjennomløp som tidligere (dersom spørringen inneholder minst én joinklausul), og vil generelt føre til en ytelsesforbedring. Denne endringen vil i tillegg ha innvirkning på, og effektivisere, de fleste andre algoritmer for eksekvering av spørreplaner i Rindex, siden en slik gjennomløping også er vanlig i disse.

For substitusjonsalgoritme 5: Et alternativ til løsningen beskrevet over, er å la arrayen med pekere til relasjonens primærnøkler (*rel_pkeys*) ha en fast størrelse lik antall relasjoner i databasen (markert med t i figuren). På denne måten vil hver peker til en relasjons primærnøkler ha sin faste posisjon i arrayen. Posisjonen (senere omtalt som *rel_pos*) blir den samme som i den interne ordningen av relasjonene i Rindex.[20] Dermed er det mulig å bruke direkte oppslag for å aksessere de ønskede nøklene, og den indre løkken fra substitusjonsalgoritme 2, 3 og 4, kan fjernes. Denne endringen kan imidlertid ha et par negative konsekvenser. For det første vil t ofte være en del større enn de tilsvarende

[20] Dette er tidligere beskrevet i avsnitt 5.2 på side 22.



Figur 11.4: Datastruktur for indekser og inverterte indekser for substitusjonsalgoritmene 4 og 5

størrelsene (k og s) i de forrige løsningene, ikke minst i forhold til substitusjonsalgoritme 4. Dette betyr økt minnebruk i hver indeks og hver mellomlagring. For det andre er det ikke sikkert at denne løsningen vil forbedre ytelsen merkbart. Grunnen er at i den forrige løsningen vil aldri den indre løkken iterere særlig mange ganger (avhengig av antall joinoperasjoner og hvilken substitusjonsoperasjon som skal utføres). I tillegg vil økningen av denne arrayens størrelse påvirke algoritmene for andre operasjoner i indekshåndteringen negativt, siden datastrukturen gjennomløpes ofte.

Videre har hver relasjon i disse datastrukturene en array med primærnøkler som er felles for indeksene i relasjonen (se figur 11.4). Hver slik primærnøkkel inneholder en peker til en array med elementene i primærnøkkelen, som er pekere til attributtverdier i den samme eller andre indekser. Hver primærnøkkel inneholder i tillegg en peker (*inv_ptr*) til den inverterte indeksen.

Når det gjelder de inverterte indeksene, kan strukturen i enten algoritme 2 eller i algoritme 3 brukes (markert med henholdsvis `Inverted Index structure 2` og `Inverted Index structure 3` i figuren). De vil bruke omtrent like mye plass i minnet og vil i teorien være like effektive.[21]

11.4.2 Algoritmene

For substitusjonsalgoritme 4, vil algoritmen bli den samme som for substitusjonsalgoritmene 2 og 3 (se for eksempel algoritmelisting 3 på side 66). Substitusjonsalgoritme 5 er vist under. Legg merke til at den indre løkken

Substitusjonsalgoritme 5
foreach *attribute value* **in** *index* **do**
attribute value ← *key_ptr* → *rel_pkeys*[*toindex* → *rel_pos*] → *inv_ptr*[*toindex* → *attr_pos*] → *attribute value*
end

er borte her. I denne algoritmen er *key_ptr* pekeren fra indeksen til arrayen (*rel_pkeys*) med pekere til relasjonens primærnøkler. For oppslaget i denne arrayen, brukes altså den interne posisjonen *rel_pos* som indeksen det nye attributtet skal ligge i (*toindex*), har i Rindex. Dette oppslaget resulterer blant annet i en peker til den inverterte indeksen *inv_ptr* (enten `Inverted index structure 2` eller `Inverted index structure 3` i figuren). I den inverterte indeksen brukes attributtposisjonen *attr_pos* som indeksen *toindex* har i den tilhørende relasjonen, for direkte oppslag.

[21] Det er mulig en av dem er raskere enn den andre for eksempel på grunn av utnyttelsen av cachen på maskinen som følge av ulik datastruktur.

Tidskompleksitet

Tidskompleksiteten for substitusjonsalgoritme 4 vil bli $T(n) = \mathcal{C}(n*s)$, der n er antall attributtverdier i indeksen og s antall relasjoner som indeksen inneholder primærnøkler av. Som nevnt tidligere i avsnittet, vil denne s-verdien ofte være mindre enn tilsvarede verdi (det vil si k-verdien) i substitusjonsalgoritmene 2 og 3, spesielt i spørringer som inneholder en eller flere joinklausuler. Dette muliggjør en ytelsesforbedring. Dersom spørringen ikke inneholder joinklausuler, vil kompleksiteten være $\mathcal{C}(n)$.

For substitusjonsalgoritme 5 trenger algoritmen bare den ytterste løkken for å utføre operasjonen, og algoritmen er derfor optimal med kompleksiteten $T(n) = \mathcal{C}(n)$. Effektiviteten i andre algoritmer i indekshåndteringen kan, som tidligere nevnt, påvirkes negativt av denne endringen i primærnøklene.

Plasskompleksitet

Substitusjonsalgoritme 4 vil antageligvis føre til en merkbar reduksjon i minnebruken til programmet, spesielt etter genereringen av de permanente indeksene ved oppstart. Grunnen er at primærnøklene er felles for alle attributter i samme relasjon, og at s-verdien ofte er mindre enn den tilsvarende k-verdien. Hver indeks vil her ha en plasskompleksitet på $\mathcal{C}(n + n * s)$. For de permanente indeksene vil $s = 1$ og plasskompleksiteten dermed $\mathcal{C}(2n)$. Primærnøklene vil i tillegg bruke $\mathcal{C}(n * k)$ for hver *relasjon*, der k som før er antall elementer i primærnøkkelen.

Substitusjonsalgoritme 5 vil kreve en del mer plass i minnet enn den forrige algoritmen, og kompleksiteten for denne er $\mathcal{C}(n + n * t)$, der t er antall relasjoner i databasen. Grunnen til forventningen om økt minnebruk for denne algoritmen, er at denne t-verdien med stor sannsynlighet vil være en del større enn s-verdien i forrige algoritme, og k-verdiene i de tidligere algoritmene. Primærnøklene vil også her bruke $\mathcal{C}(n * k)$.

For de inverterte indeksene er det som nevnt også mulig å velge to ulike datastrukturer (se figur 11.4). Disse to alternativene vil gi lik effektivitet, men strukturen for algoritme 3 er her valgt fordi den har et forventet lavere minnebruk enn algoritme 2. Plasskompleksiteten blir dermed $\mathcal{C}(n)$ for de inverterte indeksene i hver relasjon.

For substitusjonsalgoritme 4 vil hver indeks få en total plasskompleksitet på $S(n) = \mathcal{C}(n + n * s) + \mathcal{C}((n * k)/r) + \mathcal{C}(n) = \mathcal{C}(2n + n * s + (n * k)/r)$, mens den for substitusjonsalgoritme 5 vil være $S(n) = \mathcal{C}(2n + n * t + (n * k)/r)$, der n er antall attributtverdier, k antall elementer i primærnøkkelen, r antall indekserte attributter i relasjonen og s og t som beskrevet over.

11.5 Sammenligning

Det har blitt presentert fem ulike måter å effektivisere den originale substitusjonsoperatoren (algoritme 0) på. Hver av dem har en bestemt datastruktur for de inverterte indeksene og primærnøklene, og dermed også en bestemt substitusjonsalgoritme. Her følger en sammenligning av løsningene.

11.5.1 Teoretisk sammenligning

Som en oppsummering viser tabellen under en sammenligning med hensyn på tidskompleksiteten til de ulike løsningene, samt plasskompleksiteten til datastrukturene[22] de benytter.

	Tidskompleksitet	Plasskompleksitet
Algoritme 0	$\mathcal{C}(n * h + n * l * k * k_2)$	$\mathcal{C}(2n + n * k)$
Algoritme 1	$\mathcal{C}(n * h + n * l * k * k_2)$	$\mathcal{C}(2n + n * k)$
Algoritme 2	$\mathcal{C}(n * k)$	$\mathcal{C}(n + n * k * r)$
Algoritme 3	$\mathcal{C}(n * k)$	$\mathcal{C}(2n + n * k)$
Algoritme 4	$\mathcal{C}(n * s)$	$\mathcal{C}(2n + n * s + (n * k)/r)$
Algoritme 5	$\mathcal{C}(n)$	$\mathcal{C}(2n + n * t + (n * k)/r)$

Tabell 11.2: Sammenligning av substitusjonsalgoritmene

I tabellen er n antall attributtverdier, h kompleksiteten til hash-funksjonen (som i verste tilfellet er tilnærmet lik k), k og k_2 antall elementer i primærnøklene for henholdsvis indeksen og kollisjonsnoden, l gjennomsnittslengden av kollisjonslisten, r antall indekserte attributter i den tilhørende relasjonen, s antall relasjoner indeksen/mellomlagringen inneholder primærnøkler av og t totalt antall relasjoner i databasen.

Som et utgangspunkt for videre diskusjoner og resonneringer, har jeg valgt følgende ordning av variablene: $l \leq s \leq h \leq k \leq r \leq t \leq n \leq m$. Dette vil ha innvirkning på hvilke forventninger vi kan ha til algoritmene. Denne ordningen vil sannsynligvis passe ganske godt i en normalt stor og omfattende database (med hensyn på relasjonsstørrelser, antall attributter i hver primærnøkkel og antall kollisjoner i hash-tabellen).

Her følger en kort drøfting av kompleksitetsforskjellene i mellom de ulike algoritmene, utfra tabellen over:

Algoritme 0 og 1: Som tidligere nevnt har disse to algoritme i teorien samme tids- og plasskompleksitet, men i praksis vil det nok vise seg forskjeller i effektiviteten. Årsaken er forskjellen i implementasjonene av hash-tabellene, og utfra dette kan vi forvente en mindre gjennomsnittslengde l av kollisjonslistene, og dermed også en bedre effektivitet, for

[22] Vi ser her bare på datastrukturene til indekser, siden mellomlagringer ikke genererer inverterte indekser.

algoritme 1. En eventuell forskjell i minnebruken mellom de to algoritmene, kommer av forenklingen vi gjorde ved å anta at tomme posisjoner i hash-tabellen ikke bruker plass i minnet, siden dette ikke gjelder i praksis.

Algoritme 1, 2 og 3: Siden tidskompleksiteten til algoritme 2 er betraktelig mindre enn kompleksiteten til algoritme 1, kan vi forvente en merkbar forbedring i effektiviteten fra algoritme 1 til 2. Det er nok her vi kan forvente den største endringen i effektiviteten. Algoritmene 2 og 3, derimot, har samme tidskompleksitet og effektiviteten bør derfor bli omtrent lik.

Når det gjelder minnebruken, ser vi at substitusjonsalgoritme 1 og 3 har samme plasskompleksitet dersom vi antar den forenklede kompleksiteten for algoritme 1. Algoritmene 2 og 3 har litt ulik datastruktur, og vi bør derfor kunne forvente noe ulik minnebruk. Siden datastrukturen for algoritme 3 ble konstruert for å fjerne en del unødig dobbeltlagring i datastrukturen til algoritme 2, kan vi forvente at algoritme 3 vil bruke noe mindre plass i minnet enn algoritme 2.

Algoritme 3 og 4: En eventuell forskjell i effektivitet mellom disse algoritmene vil ikke vise seg før substitusjonsoperatoren brukes på en mellomlagring som er resultatet av en tidligere joinoperasjon. Dette er fordi algoritmene ikke trenger å gjennomløpe den indre løkken dersom primærnøklene i mellomlagringen ikke er satt sammen av primærnøkler fra ulike relasjoner. Dersom det for eksempel har blitt utført én joinoperasjon, vil $s = 2$, mens k er lik summen av antall elementer i de to primærnøklene som joines. Dette kan gi utslag på effektiviteten.

Det er ikke helt trivielt å bestemme forskjellen i minnebruk for algoritmene 3 og 4 ut fra kompleksitetene i tabellen. Det er derimot enklere og mer intuitivt å resonnere ut fra beskrivelsen av datastrukturene, gitt i de foregående avsnittene. En del av bakgrunnen for at algoritme 4 (og 5) ble utforsket, var observasjonen av at de tidligere datastrukturene brukte unødig plass i minnet. Årsaken til dette er, som sagt, at primærnøklene i disse strukturene blir duplisert for hver indeks innen samme relasjon. Datastrukturen til algoritme 4 (og dermed også algoritme 5) ble konstruert for å eliminere denne dupliseringen av data, og vi kan derfor forvente at minnebruken for algoritme 4 blir vesentlig mindre enn for algoritme 3 (og tidligere algoritmer).

Algoritmene 4 og 5: Forskjellen i effektivitet mellom disse to algoritmene er nok ikke like stor i praksis som det kan virke utfra tidskompleksiteten. Dette kommer av at s-verdien som regel vil være liten. Det er også mulig at andre faktorer (for eksempel utnyttelse av cachen) kan være mer utslagsgivende.

For minnebruken vil det nok eksistere en forskjell, selv om de benytter samme datastruktur. Grunnen er at forskjellen på s- og t-verdiene kan bli forholdsvis stor, siden s er 1, eller eventuelt like stor som antall relasjoner som har vært involvert i spørringen, mens t er antall relasjoner i databasen.[23]

11.5.2 Empirisk sammenligning

De ulike forslagene er implementert, og en del ytelsestester utført, for å undersøke om resultatene og forholdene blir som forventet. Originalalgoritmen (tidligere omtalt som substitusjonsalgoritme 0) i versjon 0.1 av Rindex er brukt som referansemåling for å kunne måle innvirkningen av de foreslåtte algoritmene i forhold til utgangspunktet og til hverandre.

Først og fremst er det tidsbruken som er interessant å måle og sammenligne, siden dette er bakgrunnen for problemstillingen. Utfra den teoretiske analysen, kan man forvente at de siste fire algoritmene vil gi ganske lik ytelse. Derfor kan det også være interessant å se på minnebruken til de ulike datastrukturene som er foreslått brukt for hver enkelt algoritme.

Testmaskinen

Maskinen som er brukt til de praktiske testene og målingene, er en dedikert maskin (`blot.ifi.uio.no`) som selv kjører en Oracle-database (versjon 9i, release2). Maskinen har 2GB RAM (internt hurtigminne) og to Intel® Xeon™ prosessorer, hver med en klokkehastighet på 2.4GHz, støtte for HyperThreading og 512KB level 2 cache.

Testmaskinen har ingen andre oppgaver, og bare noen få brukere har tilgang til systemet. Dette er gjort for å redusere antall mulige feilkilder i testene.

Testdatabasene

Tidsbruk: For å teste effektiviteten til de ulike løsningene, blir det brukt en database der alle attributtene er av heltallstypen (int), og attributtverdiene er positive heltall. Derfor vil den bli referert til som nummerdatabasen. Databasen består av relasjonene $X(\overline{x_1}, x_2, x_3, x_4, x_5)$, $Y(\overline{y_1, y_2, y_3, y_4}, y_5)$ og $Z(\overline{z_1, z_2, z_3, z_4}, z_5)$, hvor $y_1, z_1 \subseteq x_1$.[24] En fordel med denne databasen er at det er enkelt å automatisk generere relasjoner med bestemte størrelser, utfra hvor store datasett man ønsker å teste på. Relasjonsstørrelsene blir satt slik

[23] I store databaser kan antall relasjoner t komme opp i flere hundre, mens det er 11 relasjoner i filmdatabasen.

[24] Denne notasjonen betyr at for eksempel relasjon Y består av de fem attributtene y_1 til y_5, der attributtene y_1 til y_4 utgjør primærnøkkelen. Tilsvarende gjelder for de andre relasjonene. Det siste uttrykket betyr at attributtene y_1 og z_1 er fremmednøkler til x_1 og attributtverdiene i disse to er derfor en delmengde av verdiene i x_1 (som kalles domenet).

at $|X| \leq |Y| \leq |Z|$. Primærnøklene i Y og Z består av mange attributter for at det skal være mulig å konstruere spørringer der k- og s-verdiene er forholdsvis ulike. Dette er nødvendig for å kunne måle forskjeller i effektivitet mellom for eksempel algoritme 3 og 4 (se tabell 11.2).

Attributtverdiene i databasen er tilnærmet uniformt distribuert, siden det er brukt en random-funksjon (uten seed) for å generere verdiene som ligger i et bestemt intervall[25]. Alle attributtverdiene blir generert på denne måten, bortsett fra fremmednøklene (her y_1 og z_1). Attributtverdiene for disse trekkes på samme tilfeldige måte. Dette domenet er ikke det samme intervallet, men i stedet de verdiene som attributtet de refererer til (x_1) allerede har blitt tildelt.

Testene blir utført på relasjoner som gir bestemte størrelser på de mellomlagringene som substitusjonsoperatorene skal arbeide på. Mellomlagringene spørringene kjøres mot er valgt å være av størrelsesorden 10 000, 50 000, 100 000, 200 000 og 300 000.

Minnebruk: For å måle minnebruk, er filmdatabasen [12] valgt fordi den er relativt stor og komplisert (i forhold til databasen beskrevet over), og fordi den har større variasjon av attributtyper og relasjonsstørrelser. Disse egenskapene er nok til å gi målbare utslag med hensyn på plassbruk i minnet. Databasen har totalt 28 attributter fordelt på 11 relasjoner av ulike størrelser, mellom 5 og 232 598 tupler. I gjennomsnitt er det omtrent 49 414 tupler i hver relasjon. Alle attributtene i databasen blir indeksert av Rindex. Minnebruken blir målt etter oppstart og generering av alle indekser og inverterte indekser i Rindex.

Testprogrammet

Hver av de ulike substitusjonsalgoritmene er implementert i Rindex og de er kompilert med optimalisering på 2. nivå (`-O2`) av `gcc`-kompilatoren[26]. Hver av de ulike implementasjonene er lagt ut på hjemmesiden til prosjektet.

For å utføre testene, er skriptet `benchmark_subst.py` brukt. Skriptet utfører en rekke kjøringer. En kjøring består i å teste en substitusjonsalgoritme med 100 repetisjoner av en spørring mot datasett av en bestemt størrelse. Dette betyr at vi trenger å utføre én kjøring for hver kombinasjon av substitusjonsalgoritme, SQL-spørring og størrelse på datasettet. Siden vi har seks algoritmer, fire spørringer og fem ulike størrelser på datasettet, vil det si totalt 120 kjøringer, hver av dem med 100 repetisjoner av spørringen. I hver kjøring blir gjennomsnittstiden for den aktuelle implementasjonen av substitusjonsalgoritmen beregnet. Siden hver av disse operatorene blir utført på like store datasett, beregnes gjennomsnittet på enkleste måte ved å dividere

[25] I denne databasen er intervallet satt til $[0, N]$, der N er størrelsen på X-relasjonen.

[26] `gcc` står for GNU Compiler Collection, og er en sentral del av GNU-prosjektet, se `http://gcc.gnu.org`.

akkumulert eksekveringstid fra hver substitusjonsoperator med totalt antall slike operatorer i kjøringen.

Målingene av eksekveringstiden skjer internt i programmet, det vil si bare mens programmet faktisk bruker CPUen.[27] Derfor skal målingene ikke påvirkes av varierende belastning på maskinen. Unntaket er målingene som involverer den underliggende databasen (her Oracle), som for eksempel tiden det tar å generere indeksene ved oppstart. For å minimere innvirkningen av mulige feilkilder, er testene utført på den dedikerte maskinen, og på tidspunkter da maskinen har minimal belastning.

Tester har vist at kjøringer med et mindre antall repetisjoner bare gir små utslag i tidsbruken. Dette tyder på at en kjøring med for eksempel 20 repetisjoner av spørringen ville gitt ganske likt resultat som å teste med 100 repetisjoner, slik det ble gjort her.

Spørringene

For å utføre testene, er det benyttet ulike SQL-spørringer som oppfyller tre krav:

1 Alle substitusjonsoperatorene i en spørring må behandle et datasett av én bestemt størrelse. Dette kravet gjør det mulig og fornuftig å beregne og sammenligne gjennomsnittstiden for de ulike implementasjonene av substitusjonsalgoritmen.

2 Spørringene må inneholde et ulikt antall substitusjonsoperatorer[28], fordi vi forventer at antallet ikke skal ha noen innvirkning på gjennomsnittstiden.

3 For det tredje bør det være en forskjell på k- og s-verdiene i en spørring, for å kunne måle den eventuelle innvirkningen dette har på effektiviteten.

Hver SQL-spørring kjøres mot ulike genererte databaser med forskjellige størrelser på datasettet. Dette blir gjort for å teste om datamengden substitusjonsoperatorene arbeider på, har innvirkning på forholdet i effektivitet mellom de ulike substitusjonsalgoritmene. Størrelsen på datasettet blir bestemt av størrelsene på relasjonene og eventuelle joinklausuler gitt i spørringen. For å oppfylle kravet til konstant størrelse på datasettet, må de andre operatorene som substitusjonsoperatorene får input fra (for eksempel seleksjon og union), ikke føre til endring i størrelsen av resultatsettet. For enkelhets skyld blir det derfor bare brukt seleksjoner, og ikke unioner, i spørringene. Dersom

[27] Biblioteksfunksjonen *clock*(3) er brukt for å måle eksekveringstiden. Funksjonen er definert i ISO C-standarden.

[28] Antall nødvendige substitusjonsoperatorer og plasseringen av dem i spørretreet er mer detaljert beskrevet i avsnitt 7.3 på side 37.

en SQL-spørring bare krever én substitusjonsoperator, vil kravet automatisk bli oppfylt.

Siden Rindex ikke gjør noen optimaliseringer av spørretreet, vet vi at operatorene blir eksekvert i samme rekkefølge som spørringen spesifiserer. Operatorene i `where`-klausulen blir eksekvert fra venstre mot høyre.

Følgende fire SQL-spørringer er valgt ut fra kravene over. Hver spørring er også illustrert i figur 11.5 på neste side.

Spørring 1:

```
select z1 from Z where z2>=0 and z3>=0
```

Dette er en av de enkleste spørringene som oppfyller kravene over. Den fremtvinger bruk av én substitusjonsoperator, og denne vil bli lagt inn mellom seleksjonene på attributtene `z2` og `z3`, som vist i figur 11.5(a). For å unngå at størrelsen på datasettet endres, må ikke betingelsene til seleksjonene redusere størrelsen på resultatsettet. Det er for eksempel mulig å sette `z2>=0`, siden alle attributtverdier er positive. Denne teknikken er brukt for alle seleksjonsoperatorene i denne og i de andre spørringene. I denne spørringen er $k = 4$, siden `Z`-relasjonen inneholder fire attributter i primærnøkkelen. Videre er $s = 1$ fordi det bare er én relasjon involvert i spørringen.

Spørring 2:

```
select x1 from (Y inner join Z on y1=z1) where z3>=0
```

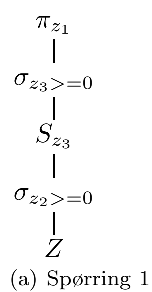
Denne spørringen involverer én joinoperasjon og fremtvinger én substitusjonsoperator mellom joinoperatoren og seleksjonsoperatoren, fordi seleksjonen skal utføres på et annet attributt (`z2`) enn joinen resulterer i (`y1`). Spørringen er vist i figur 11.5(b). Her er $k = 8$ fordi joinen medfører at alle attributtene i primærnøklene til Y og Z, er med i joinresultatet. Siden det her er to relasjoner involvererert, blir $s = 2$.

Spørring 3:

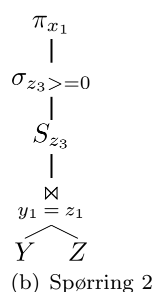
```
select x1 from ((X inner join Y on x1=y1)
 inner join Z on x1=z1) where y2>=0 and z3>=0
```

I denne spørringen joines alle tre relasjonene, og derfor er $k = 9$ og $s = 3$. I tillegg er det benyttet to seleksjoner. Her vil det være nødvendig med to substitusjonsoperatorer: en etter den ytterste joinoperatoren og før den første seleksjonsoperatoren, og den siste mellom de to seleksjonsoperatorene (på henholdsvis `y2` og `z3`). Dette er vist i figur 11.5(c). Legg merke til at SQL-spørringen er konstruert på en slik måte at det ikke er nødvendig med en substitusjonsoperator mellom de to joinoperatorene, siden det er `x1` som vil ligge i mellomlagringen fra den første joinen (mellom `X` og `Y`) og dette er ett av attributtene som den siste joinoperatoren forventer som input.[29] På denne måten unngår vi

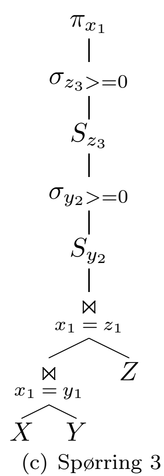
[29]Se også avsnitt 7.3 på side 37.



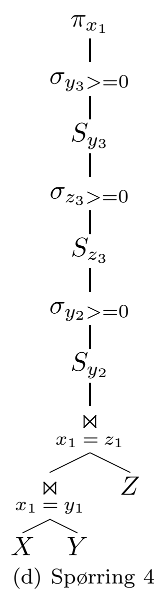
(a) Spørring 1



(b) Spørring 2



(c) Spørring 3



(d) Spørring 4

Figur 11.5: De fire testspørringene

at en slik substitusjon vil påvirke gjennomsnittstiden, siden denne ville arbeidet på et mindre datasett enn de andre.

Spørring 4:

```
select x1 from ((X inner join Y on x1=y1)
 inner join Z on x1=z1) where y2>=0 and z3>=0 and y3>=0
```

Den siste spørringen introduserer tre substitusjonsoperatorer: to av dem på samme måte som i spørring 3, og én mellom de to siste seleksjonsoperatorene (`z3` og `y3`). Figur 11.5(d) viser dette. Av samme grunn som for forrige spørring, blir det ikke lagt inn en substitusjonsoperator mellom de to joinoperatorene. I tillegg blir k- og s-verdiene de samme som for forrige spørring, siden det er de samme relasjonene som joines.

Det første kravet oppfylles av hver spørring, ved at betingelsene i seleksjonsoperatorene ikke fjerner noen tupler i resultatsettet. Det andre kravet oppfylles fordi antall nødvendige substitusjonsoperatorer varierer mellom spørringene. Til slutt innfris det siste kravet ved at k- og s-verdiene er forholdsvis ulike i spørringene.

Resultatene

Dette avsnittet presenterer resultatene for tidsbruk og minnebruk for de ulike substitusjonsoperatorene, samt tidsbruk ved genereringen av datastrukturene for indeksene og de inverterte indeksene. Til slutt trekkes det fram noen mulige feilkilder.

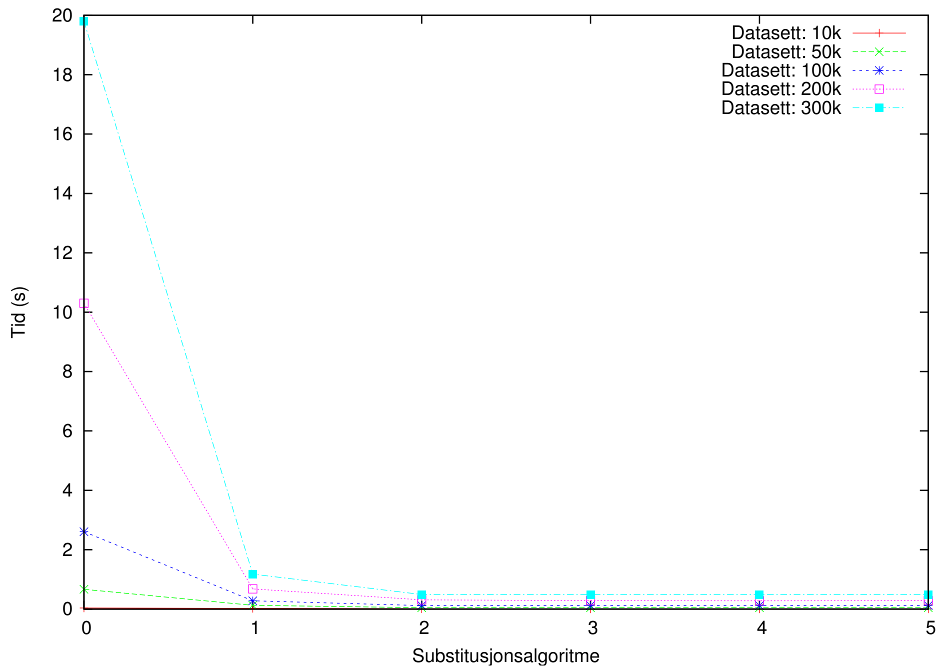
Tidsbruk: For hver SQL-spørring vil, som sagt, hver kjøring skrive måledataene (gjennomsnittstiden) til en fil. Disse dataene kan dermed brukes for å sammenligne de ulike substitusjonsalgoritmene. Det er benyttet `gnuplot`[30] for å generere et plott for hver SQL-spørring. I tillegg er det også generert et plott med en logaritmisk skala på y-aksen for hver spørring. Plottene for spørring 1 er vist i figurene 11.6 og 11.7, for spørring 2 i figurene 11.8 og 11.9, mens figurene 11.10 og 11.11 er plottene for spørring 3 og 11.12 og 11.13 for spørring 4. I hvert av plottene er gjennomsnittstiden (målt i sekunder) lagt på y-aksen, mens x-aksen representerer de ulike substitusjonsalgoritmene. Hver graf i plottet representerer kjøringer mot én bestemt størrelse på datasettet (det vil si en mellomlagring eller en indeks).[31]

Her følger noen kommentarer til resultatene, ut fra forventningene diskutert i avsnitt 11.5.1 på side 71:

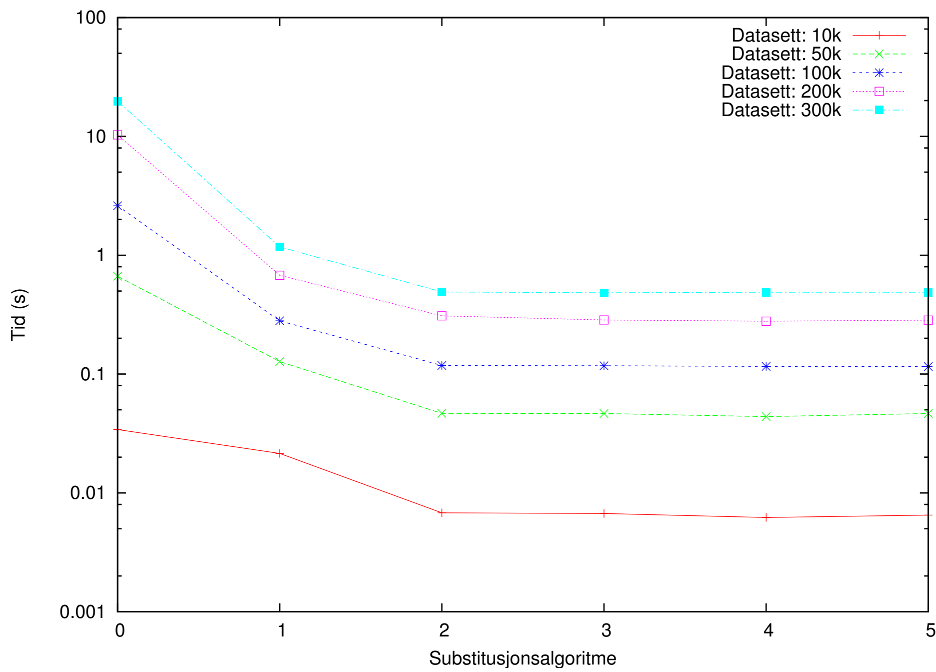
- Det er, som forventet, størst endring i ytelse mellom substitusjonsalgoritmene 0, 1 og 2, mens de fire siste er mer jevne.

[30] `gnuplot`, `http://www.gnuplot.info`, er et kommandolinjebasert program for å lage 2D og 3D plott.

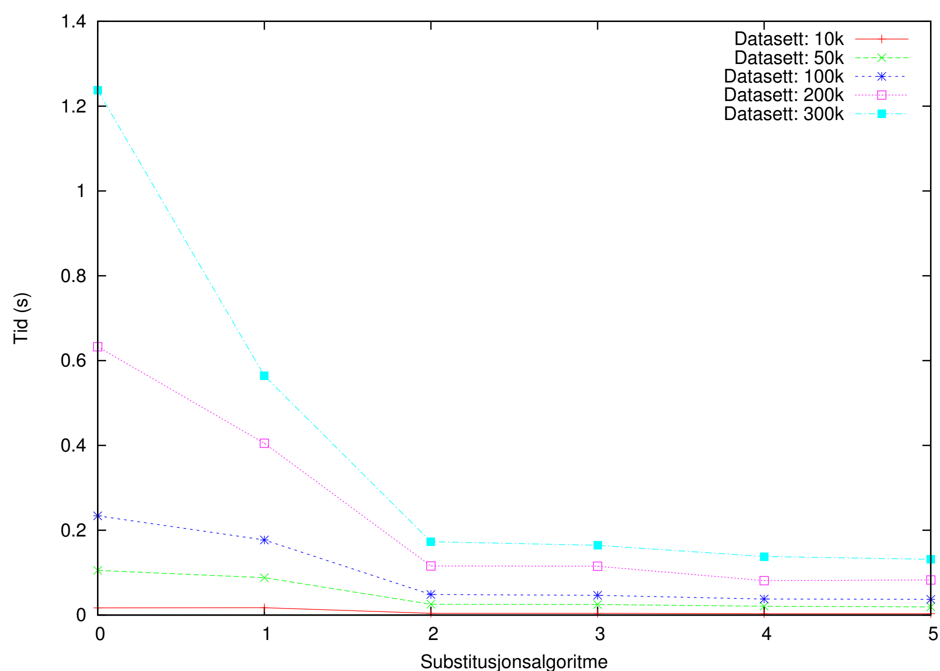
[31] Størrelsene er gitt med endelsen k for kilo, det vil si tusen.



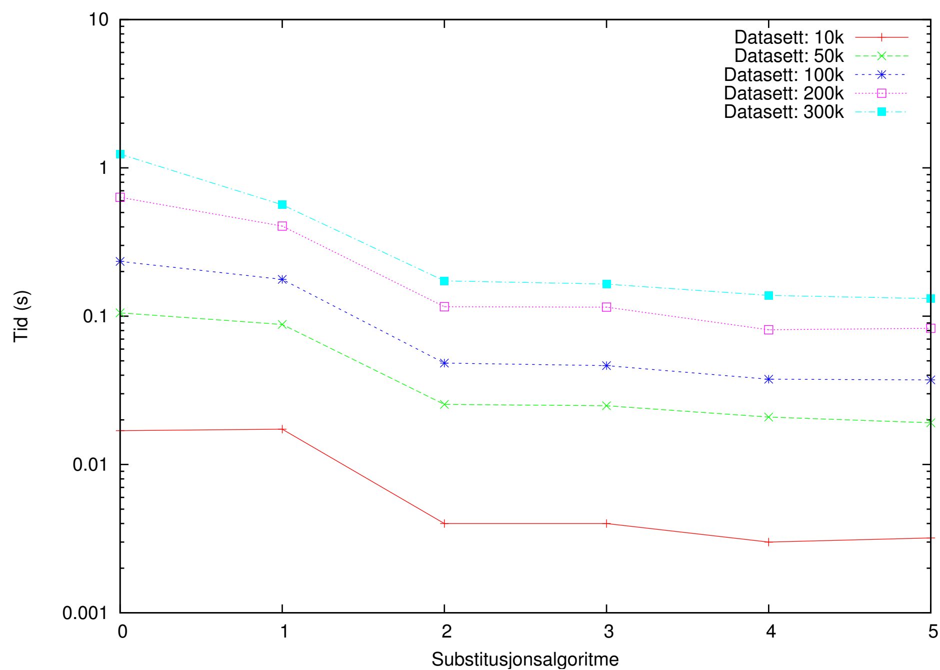
Figur 11.6: Plott for spørring 1



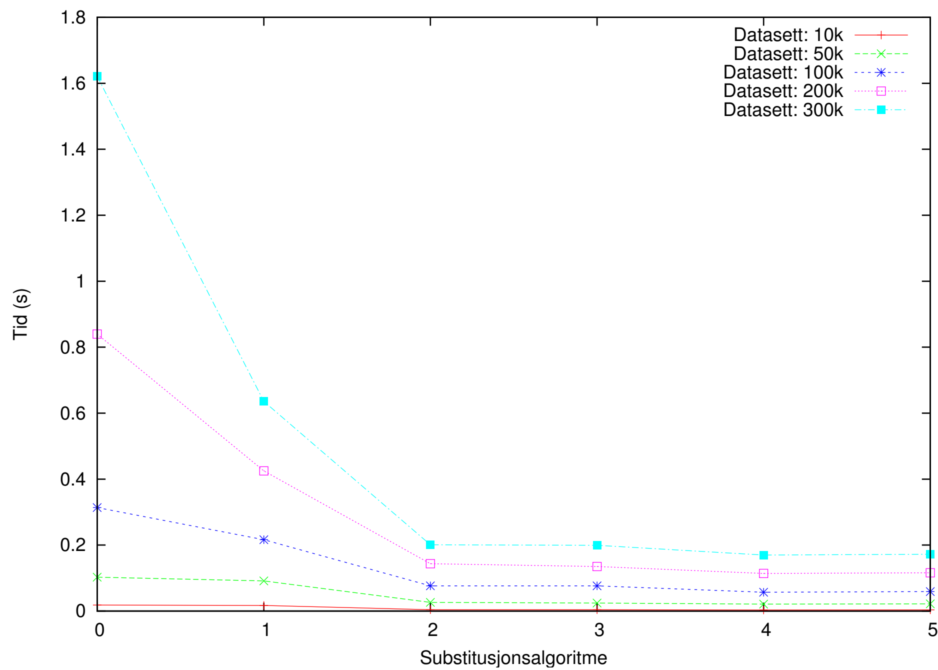
Figur 11.7: Plott med logaritmisk skala for spørring 1



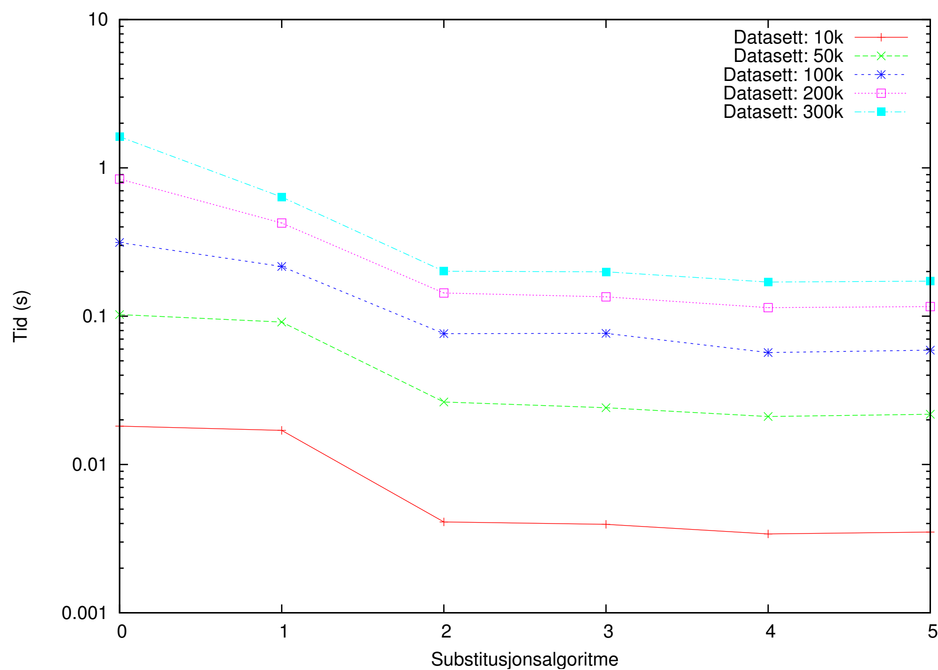
Figur 11.8: Plott for spørring 2



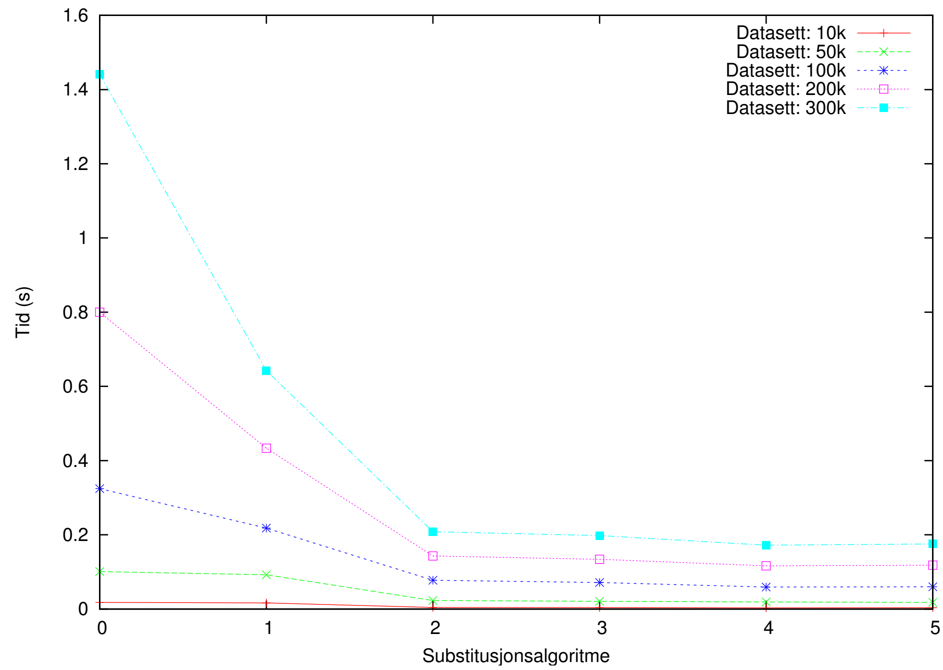
Figur 11.9: Plott med logaritmisk skala for spørring 2



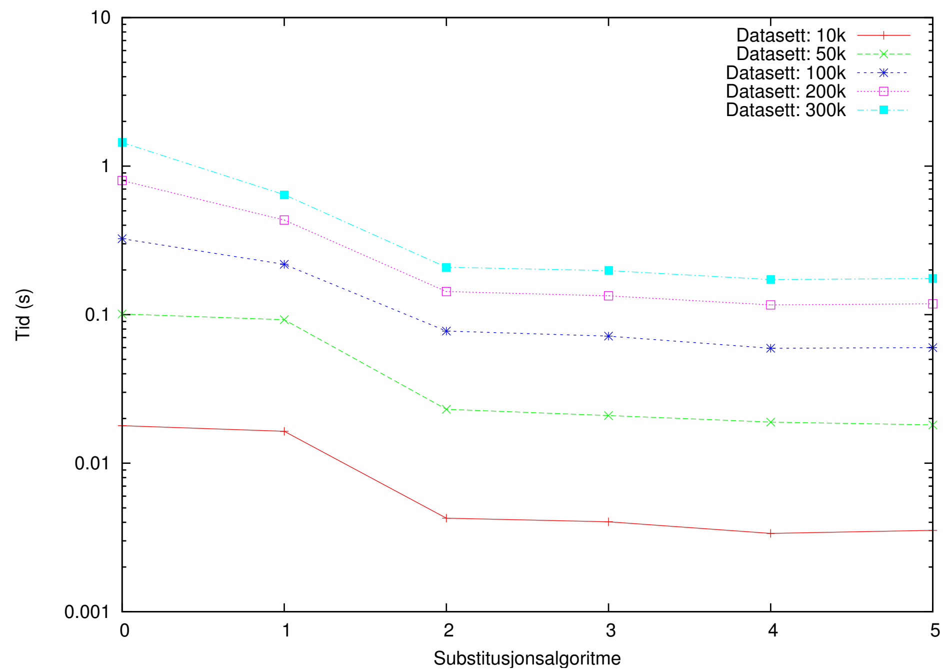
Figur 11.10: Plott for spørring 3



Figur 11.11: Plott med logaritmisk skala for spørring 3



Figur 11.12: Plott for spørring 4



Figur 11.13: Plott med logaritmisk skala for spørring 4

- Spørring 1 resulterer i den største forskjellen mellom de tre første algoritmene. Hovedgrunnen er at `Y`-relasjonen her er mye større enn relasjonene i de andre testene, siden denne spørringen ikke inneholder noen join. Dette betyr at kollisjonslistene blir større, spesielt for algoritmene 0 som har en fast tabellstørrelse. De fire siste algoritmene kjører alle i $\mathcal{C}(n)$ på grunn av mangelen på en joinklausul.

- Videre ser vi fra målingene at algoritmene 2 og 3 er omtrent like. Dette er som forventet fordi algoritme 2 og 3 har samme tidskompleksitet $\mathcal{C}(n * k)$. Det er også klart at algoritme 4 (med tidskompleksiteten $\mathcal{C}(n * s)$) er minst like effektiv som de foregående algoritmene. Dette er også som forventet og henger sammen med differansen mellom k- og s-verdiene.

- Det som er mest overraskende med resultatene, er at algoritme 5 i enkelte tilfeller gjorde det litt dårligere enn algoritme 4, til tross for at den skulle ha den beste effektiviteten med $\mathcal{C}(n)$.

  En mulig årsak til dette kan være at alle implementasjonene av substitusjonsalgoritmene krever at datastrukturene med lengdene k, s, og t traverseres for å kopiere elementene. I testene er $1 \leq k \leq 9$, $1 \leq s \leq 3$ og $t = 3$. Dette kan forklare noe av fordelen med algoritme 4 i de spørringene hvor s er den minste av disse størrelsene. I den siste spørringen, der $s = t$, er algoritme 4 og 5 nesten like effektive.

- Forskjellen mellom algoritme 0 og 1 viser hvor stor effekt endringen av tabellstørrelse, fra en statisk størrelse til et dynamisk satt primtall, har. Hovedårsaken til effektivitetsforbedringen er derfor kortere kollisjonslister.

- Den merkbare reduksjonen i tidsbruk fra algoritme 1 til 2 viser også at pekerstrukturer av den typen de siste algoritmene baseres på, og ikke hash-tabeller, er den beste løsningen. Årsaken til denne forbedringen ligger hovedsaklig i forenklingen av substitusjonsalgoritmen, ved at vi kunne fjerne flere løkker og spesielt nøstede løkker. Dette gjelder løkkene rundt kollisjonslisten og elementene i primærnøkkelen som noden peker på, samt beregningen av hash-verdien.

- Plottene med logaritmisk skala indikerer at trenden stort sett er den samme for alle relasjonsstørrelser. Vi kan derfor anta at størrelsen på datasettet ikke har innvirkning på forholdet mellom algoritmene.

- Siden forholdet mellom algoritmene er som forventet også på tvers av de ulike spørringene, kan vi anta at antall substitusjonsoperatorer ikke har innvirkning på gjennomsnittstiden til substitusjonsalgoritmene.

Minnebruk: Målingene av minnebruken er gjort etter genereringen av indeksene ved oppstart av Rindex, se figur 11.14 og utsnittet i figur 11.15.[32] Figurene viser minnebruken for indeksene (inkludert primærnøklene) og de inverterte indeksene, og hvordan disse utgjør total minnebruk for substitusjonsalgoritmene.

Ut fra forventningene om minnebruk diskutert i avsnitt 11.5.1 på side 71, ser vi følgende:

- Minnebruken går, som forventet, noe opp når størrelsen på hash-tabellen endres fra å være statisk til dynamisk satt. Grunnen er at en slik dynamisk størrelse settes noe større enn den faste størrelsen, og derfor bruker den litt mer plass i minnet.

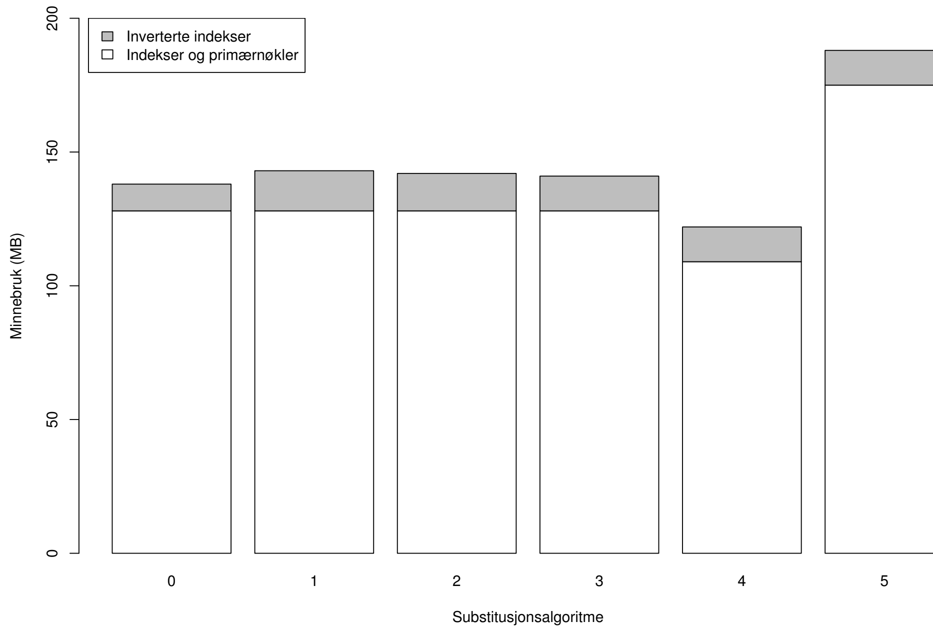
- Forskjellen mellom algoritme 2 og 3 er minimal, men den sistnevnte bruker litt mindre plass siden de inverterte indeksene er felles for relasjonen. Disse resultatene stemmer også bra med forventningene.

- Utfra den teoretiske analysen kunne vi vente oss at minnebruken til substitusjonsalgoritme 1 ville være på nivå med algoritme 3, men resultatene viser at dette ikke stemmer i praksis. Grunnen til dette avviket er at vi forenklet plasskompleksiteten til algoritme 1 ved å anta at tomme posisjoner i hash-tabellen ikke påvirket minnebruken.

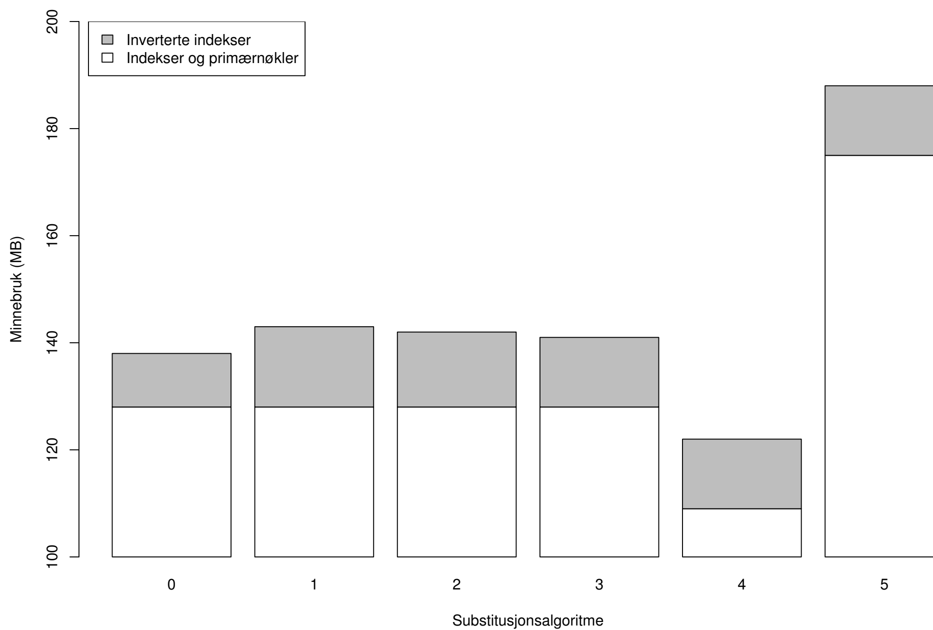
- Målingene viser i tillegg at de fire første algoritmene har samme minnebruk for indeksene og primærnøklene, siden de bruker samme datastruktur. Det er altså bare minnebruken til de inverterte indeksene som skiller dem.

- Videre ser vi at de inverterte indeksene til algoritme 4 og 5 har samme minnebruk som for algoritme 3, siden de bruker den samme datastrukturen. Derimot har omstruktureringen av primærnøklene som foreslått i algoritme 4 og 5, meget stor betydning for den totale minnebruken. Datastrukturen for algoritme 4 førte til den laveste minnebruken, mens den for algoritme 5 forårsaket den høyeste. Forskjellen er hele 66MB, noe som utgjør en økning på over 54% av den totale minnebruken til algoritme 4. Grunnen til dette er hovedsaklig forskjellen på lengdene (s og t) for strukturen som inneholder pekere til relasjonens primærnøkler, markert *rel_pkeys* i figur 11.4 på side 68. For algoritme 4 vil denne, som tidligere nevnt, alltid ha lengde $s = 1$ for de permanente indeksene, og for mellomlagringer vil størrelsen være lik antall relasjoner som inntil da har vært involvert i eksekveringen av spørringen. For algoritme 5 vil lengden t alltid være like stor som antall relasjoner i databasen (også for mellomlagringene). Databasen som er brukt som grunnlag for disse testene, har som sagt 11 relasjoner, det

[32] Her er statistikkpakken `R`, `http://www.r-project.org`, brukt for å generere plottene.



Figur 11.14: Minnebruken



Figur 11.15: Utsnitt av minnebruken

vil si $t = 11$. Denne forskjellen i størrelse har stor innvirkning fordi strukturen opprettes for hver eneste attributtverdi i hver enkelt indeks. Dette medfører at substitusjonsalgoritme 5 i nåværende form, ikke er spesielt aktuell i praksis.

Generering av indeksene: I tillegg til å måle effektiviteten og minnebruken til substitusjonsalgoritmene, ble også tiden for byggingen av indeksene målt. Dette er interessant fordi både datastrukturen til primærnøklene og til de inverterte indeksene kan ha noe betydning for denne tiden. Målingene ble gjort ved generering av filmdatabasen [12] (altså samtidig med måling av minnebruken), og resultatet er vist i figur 11.16, samt utsnittet av dette i figur 11.17.

Målingene viser at tiden det tar å generere indeksene, som forventet, er avhengig av datastrukturen, og er kortest for strukturene med minst minneforbruk. På en annen side viser målingene også at forskjellene er små nok til at det i praksis ikke vil ha noen betyding, siden denne genereringen bare gjøres under oppstart av databasen.

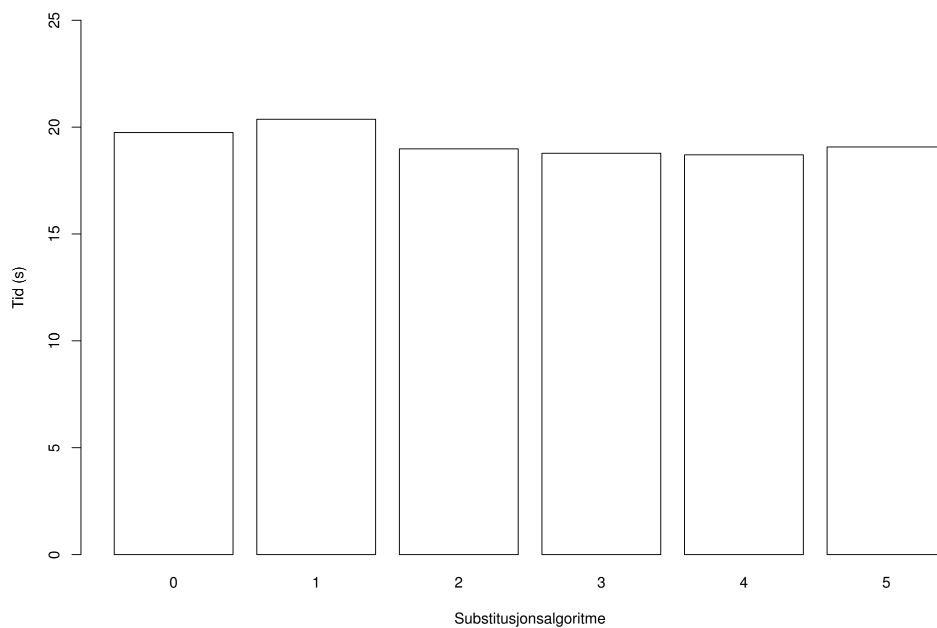
Som tidligere nevnt, er det bare kjøretiden til Rindex som måles. Dette inkluderer tiden Rindex bruker på å få overført dataene fra Oracle, men ikke tiden Oracle selv bruker (som disk I/O).[33] Derfor kan blant annet ulik belastning på maskinen påvirke tiden det tar å overføre dataene til Rindex, og dette kan med andre ord være en mulig feilkilde. Siden Rindex og databasen kjører på samme dedikerte maskin, vil ikke nettverkstrafikk påvirke målingene slik det gjorde i målingen omtalt i avsnitt 4.3 på side 17.

Andre feilkilder

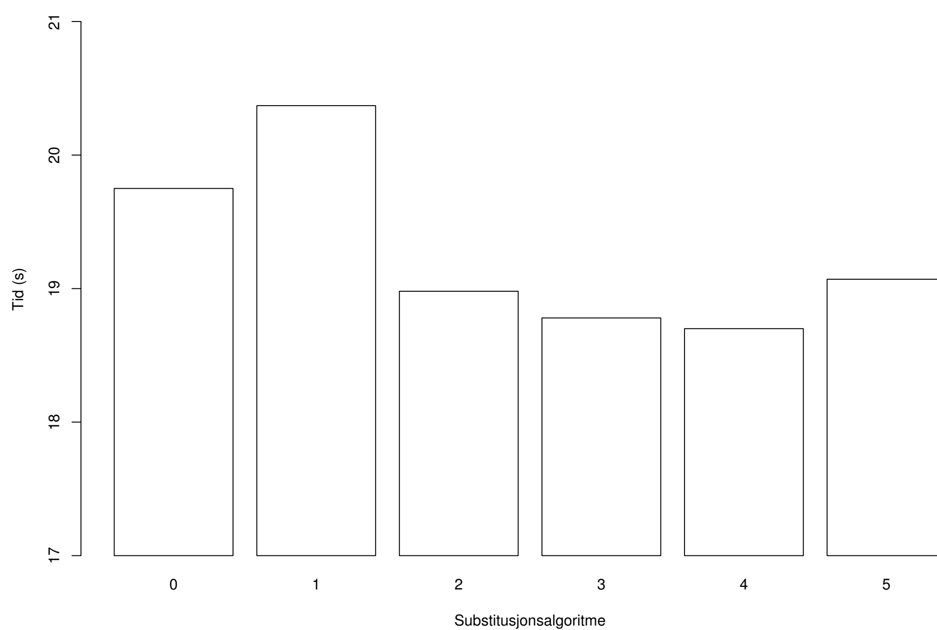
Måleresultatene kan være påvirket av ulike feilkilder, selv om disse er forsøkt minimalisert (for eksempel ved bruk av en dedikert maskin). Noen mulige feilkilder kan være:

- Hardware-spesifikke områder som caching, det vil si ulikt antall "cache miss" og "cache hit".
- Ulik minnehåndtering fra operativsystemet.
- Ulik kvalitet på implementasjonen og designen av substitusjonsalgoritmene og de tilhørende datastrukturene.
- Ulik optimaliseringsstrategi i kompilatoren som følge av forskjellene mellom datastrukturene og algoritmene.
- Ugunstig design av databasen brukt i ytelsestestene.

[33] Kjell-Magne Øierud har i sin oppgave [19] gjort tester der tiden Oracle bruker er med i, og en viktig del av, målingene.



Figur 11.16: Tidsforbruk ved generering av indeksene



Figur 11.17: Utsnitt av tidsforbruket ved generering av indeksene

- Ulik belastning på maskinen under testene.

En stor fordel med å gjøre alt i minnet, er at dette lageret er "random access" og derfor skal gi lik tid på oppslag (hvis man ser bort fra caching o.l.). Man slipper dermed enkelte andre, potensielt store, feilkilder. Et slikt eksempel er disk I/O, der aksesstidene kan variere mye mer.[34] Et annet eksempel er nettverkskommunikasjon, noe som også er ekskludert her.[35] Det at disse er utelukket som feilkilder, er nok hovedgrunnen til at målingene blir meget stabile, både med få og med mange repetisjoner av spørringen i hver kjøring. Denne stabiliteten taler også for at de nevnte feilkildene sannsynligvis ikke har hatt noen særlig innvirkning på målingene.

Rindex er grundig sjekket for minnelekkasjer (eneste kjente lekkasje er i SQLAPI-biblioteket [17] Rindex benytter).[36] Derfor var det også helt uproblematisk og stabilt å kjøre Rindex med en stor mengde sekvensielle spørringer slik det ble gjort i disse testene.

11.6 Konklusjon

Utfra analysen av substitusjonsalgoritmene og resultatene fra testkjøringene, framtrer substitusjonsalgoritme 4 som den beste og mest lovende løsningen. Dette er det flere grunner til, blant annet:

- Substitusjonsalgoritmen er minst like effektiv som de andre algoritmene.
- Algoritmen er sannsynligvis også den mest effektive ved genereringen av indeksene og de inverterte indeksene ved oppstart av databasen. Dette indikeres av målingene gjort her.
- Den tilhørende datastrukturen har det desidert laveste minneforbruket.
- Med tanke på skalering av databasen, vil nok denne datastrukturen være den best egnede. Grunnen er at lengden s (lengden på datastrukturen med pekere til relasjonens primærnøkler) bare er avhengig av spørringen og ikke størrelsen og kompleksiteten til databasen. Derimot påvirkes k, r og ikke minst t i større grad av definisjonen av databasen.[37]
- Siden datastrukturen som innholder pekere til relasjonens primærnøkler (*rel_pkeys* i figur 11.4 på side 68) er redusert til det minimale, vil

[34] Genereringen av indeksene påvirkes heller ikke av disk I/O, som forklart i forrige avsnitt.

[35] Bakgrunnen for dette er beskrevet i kapittel 1.1.

[36] `valgrind`, `http://valgrind.kde.org`, ble brukt for å sjekke minnelekkasjer og ugyldig bruk av uinitialiserte minneområder.

[37] Betydingen av disse variablene er blant annet beskrevet i avsnitt 11.5.1 på side 71.

dette også ha innvirkning på, og sannsynligvis merkbart effektivisere, de andre operatorene. Ikke minst gjelder dette projeksjonsoperatoren som aktivt bruker de inverterte indeksene for å generere resultatet. Virkningene av dette ble det for tidkrevende å teste her.

- Omstruktureringen av primærnøklene fører også til enklere, mer forståelig og intuitiv kode, og derfor sannsynligvis også mer vedlikeholdbar og utvidbar kode.

Kapittel 12

Plassering av substitusjonsoperatoren

Selv etter forbedringene av substitusjonsalgoritmen i det foregående kapittelet, er fremdeles substitusjonsoperatoren kostbar[1] i forhold til de fleste andre operatorene i Rindex (med unntak av unionsoperatoren). Derfor kan det være interessant å studere *plasseringen* av substitusjonsoperatorene, for å minimere antall nødvendige slike operatorer. Algoritmen som skal plassere substitusjonsoperatorer i spørretreet under genereringen av eksekveringsplanen vil heretter bli referert til som *plasseringsalgoritmen*.

12.1 Eksisterende algoritme

Avsnitt 7.3 på side 37 beskriver plasseringsalgoritmen som er brukt i versjon 0.1 av Rindex. Denne algoritmen tar bare hensyn til lokale forhold når den skal avgjøre om en substitusjonsoperator er nødvendig eller ikke. Den forsøker ikke å unngå unødvendige substitusjoner ved å bedre plasseringen av slike operatorer. En substitusjonsoperator er nødvendig dersom output-attributtet fra en operator ikke er det samme som det forventede input-attributtet i neste operator.[2]

Rindex har både unære og binære operatorer.[3] De unære tar kun ett input, mens de binære tar to. Hvert input representerer et attributt i form av en indeks eller en mellomlagring. Ettersom disse har lik datastruktur, er det mulig å benytte samme algoritme for begge typer input. Alle operatorene, både de unære og de binære, resulterer i én mellomlagring. I Rindex er konvensjonen at de binære operatorene alltid velger attributtet i den venstre

[1]Eksekveringstiden til en substitusjonsoperator med substitusjonsalgoritme 4, utgjør nå 40-50% av den totale kjøretiden.

[2]Med neste operator menes her den operatoren som ligger over i spørretreet, og som derfor vil utføres etter den gjeldende operatoren.

[3]Disse er beskrevet i avsnitt 4.4 på side 17.

grenen som output. Dette er gjort fordi konstruksjonen av denne plasseringsalgoritmen, samt fraværet av en optimaliseringsalgoritme, gjør at man hittil ikke har hatt noe grunnlag for å avgjøre hvilken gren som bør velges.

12.2 Forbedret algoritme

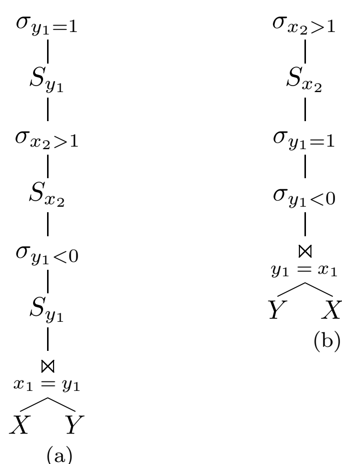
På grunn av svakhetene i den eksisterende plasseringsalgoritmen, vil vi forsøke å konstruere en algoritme som kan forbedre plasseringen av substitusjonsoperatorene. Målet med dette er å minimere antallet slike operatorer. Dette kan oppnås på minst to måter:

1 Ved å bytte om på rekkefølgen av to operatorer i spørretreet.

2 Ved å bytte venstre og høyre gren i binære operatorer.

Et viktig krav til bruken av disse reglene, er at det modifiserte spørretreet alltid må uttrykke det samme som det opprinnelige. Videre kan en binær operator bare bytte om på grenene dersom den er kommutativ. For ombytting av to unære operatorer, gjelder reglene som beskrevet i [5, avsnitt 16.2].

Slike ombyttinger kan gi ønsket effekt, en reduksjon i antallet substitusjonsoperatorer, dersom de fører til at output-attributtet fra en operator, er det samme som det input-attributtet neste operator i treet forventer. For å illustrere dette, kan vi se på et enkelt eksempel som vist i figur 12.1. Figuren



Figur 12.1: Effekt ved bruk av ombyttingsreglene

viser et spørretre som består av tre seleksjonsoperatorer og én joinoperator. To av seleksjonene skal gjøre en operasjon på y_1-attributtet, mens den tredje forventer et input med verdier fra x_2-attributtet. Joinoperatoren skal gjøre en equi-join på attributtene x_1 og y_1, som tilhører henholdsvis relasjon X

og Y.[4] I figur 12.1(a) ser vi hvordan disse operatorene kan være ordnet i Rindex, og hvordan dette fremtvinger bruk av tre substitusjonsoperatorer. Figur 12.1(b) viser hvordan en enkel ombytting av de to øverste seleksjonsoperatorene, samt det at de to grenene i joinoperatoren bytter plass, fører til at bare én substitusjonsoperator er nødvendig.

12.2.1 Beskrivelse av algoritmen

Algoritmen som skal plassere substitusjonsoperatorene i spørretreet, kan implementeres på minst to måter:

1 Som en del av optimaliseringsalgoritmen.

2 I form av en selvstendig komponent, som utføres etter optimaliseringsprosessen og på det optimaliserte spørretreet.

Det er både fordeler og ulemper med begge disse. Det første alternativet har den fordelen at ombyttingsreglene som er nevnt over, passer naturlig inn i en optimaliseringsalgoritme. Grunnen er at optimaliseringsalgoritmen blant annet bygger på ulike algebraiske regler, jf. [5, avsnitt 16.2]. På den andre siden er det også en ulempe at dette alternativet krever implementasjon av og god kunnskap om optimaliseringsalgoritmen. Plasseringsalgoritmen blir mer komplisert og kan ikke operere selvstendig. Det er naturlig at plasseringsalgoritmen, i større eller mindre grad, tar hensyn til de valg som optimaliseringsalgoritmen har gjort med hensyn på ordningen av operatorene. Tradisjonelt sett baseres optimaliseringsstrategien hovedsaklig på estimater for hvor kostbar en operator er forventet å være. Disse estimatene vil også ha betydning for den forventede effektiviteten til en substitusjonsoperator. Kostnaden til en operator er vanligvis sterkt avhengig av forventet antall tupler i inputet (enten en indeks eller en mellomlagring), og ikke minst forventet antall diskaksesser som er nødvendig for å utføre operasjonen. For å beregne disse estimatene, baserer optimaliseringsalgoritmen seg på tilgjengelig statistikk om relasjonene, samt ulike regler og heuristikker, jf. [5, avsnittene 16.4-16.5]. For Rindex er ikke antall diskaksesser interessant, ettersom alle operasjoner utføres i minnet. Derimot vil estimatet på antall tupler være en viktig faktor for denne algoritmen i Rindex, siden dette påvirker CPU- og minnebruken. Det er derfor ikke sikkert at alle valg en tradisjonell optimaliseringsalgoritme tar, er like fornuftige for Rindex. Denne problemstillingen overlates til en eventuell senere oppgave å utforske.

Det andre alternativet har naturlig nok den fordelen at plasseringsalgoritmen kan designes og implementeres uavhengig av, og dermed også før, optimaliseringsalgoritmen. En ulempe med dette alternativet, er at denne mangelen på integrasjon mellom de to algoritmene kan innebære at nyttig

[4]Relasjonene og attributtene er de samme som i nummerdatabasen, se avsnitt 11.5.2 på side 73.

informasjon fra optimaliseringsalgoritmen ikke er tilgjengelig for den aktuelle plasseringsalgoritmen.

I versjon 0.1 av Rindex gjøres det ingen optimaliseringer av spørretreet, jf. avsnitt 7.2 på side 34. Derimot er rammeverket for fremtidige optimaliseringer lagt, og dette vil være en naturlig og viktig videreutvikling av den nåværende versjonen av Rindex. På grunn av denne mangelen på optimaliseringsalgoritme, samt omfanget av å studere ulike sider ved en slik algoritme, er det første alternativet ikke studert her. Det vil derfor være alternativ 2 som er mest realistisk å utforske videre.

Den ordningen av operatorene som en optimaliseringsalgoritme velger, fører ofte til en reduksjon av antall nødvendige substitusjonsoperatorer, selv om algoritmen ikke har dette som et eksplisitt mål. Årsaken er at den grupperer operatorene, og spesielt seleksjonsoperatorene, nærmest mulig relasjonen de kommer fra. På denne måten vil de operatorene som arbeider på det samme attributtet, stort sett være samlet i ett område av treet. På bakgrunn av dette, virker det naturlig å splitte spørretreet opp i slike områder, og la plasseringsalgoritmen arbeide isolert innenfor hvert slikt område. Det vil sannsynligvis ikke lønne seg å flytte en unær operator oppover og forbi en binær operator i spørretreet, siden dette som regel vil føre til at begge operatorene må arbeide på større mellomlagringer. Et eksempel på dette er en seleksjonsoperator som kan utføres før en joinoperator i treet. En mulighet er derfor å la de binære operatorene, som join og union, danne grensene mellom de ulike områdene i treet. Et område vil derfor bestå av én binær operator og to mengder med unære operatorer som representerer hver sin gren i den binære operatoren. Hvert område vil derfor ha ett output-attributt (resultatet av den binære operatoren) og to input-attributter[5]. Dersom spørringen ikke inneholder noen binære operatorer, vil det ikke eksistere noe slikt område. I stedet kan vi bruke plasseringsalgoritmen på hele treet.

Konflikt

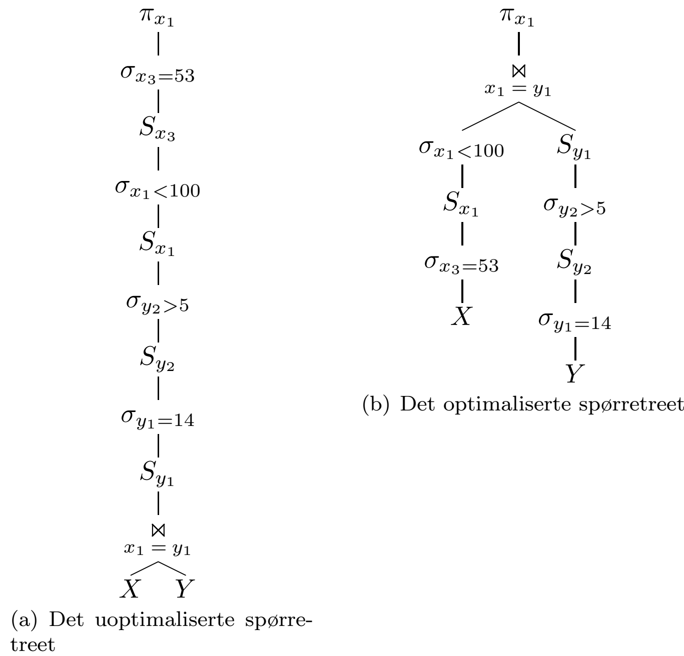
En viktig observasjon er at plasseringsalgoritmen og optimaliseringsalgoritmen ofte kan komme i *konflikt* når det gjelder valg av ordning for operatorene i treet. Optimaliseringsalgoritmen vil velge en ordning, for eksempel på seleksjonsoperatorene, som fører til mellomlagringer med lavest mulig estimat for antall attributtverdier. Dette kan føre til at det trengs flere substitusjonsoperatorer enn det ville gjort dersom rekkefølgen hadde vært en annen, men dette er ikke nødvendigvis et dårligere valg enn å minimere antall substitusjoner. Grunnen er at optimaliseringsalgoritmens ordning kan føre til at hver substitusjonsoperator kan utføres mer effektivt, siden mellomlagringene de opererer på, sannsynligvis er mindre enn de ville vært med en annen ordning. Følgende eksempel kan illustrere en slik mulig konflikt:

[5] Et input-attributt kan enten være outputet fra ett av to forrige områdene, eller en relasjon. Det er også mulig at de to input-attributtene egentlig er de samme.

Eksempel 12.1 Vi benytter også her attributter og relasjoner fra nummerdatabasen som beskrevet i avsnitt 11.5.2 på side 73. Vi ser på spørringen

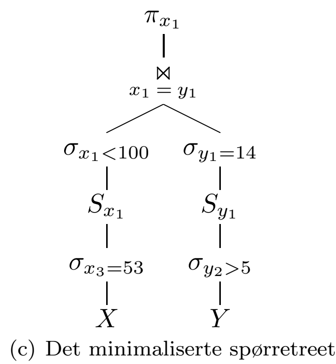
```
select x1 from (X inner join Y on x1=y1)
 where y1=14 and y2>5 and x1<100 and x3=53
```

Figur 12.2 viser tre ulike spørretrær som alle representerer spørringen over. Det uoptimaliserte spørretreet, slik det vil være i versjon 0.1 av Rindex, vi-



(a) Det uoptimaliserte spørretreet

(b) Det optimaliserte spørretreet



(c) Det minimaliserte spørretreet

Figur 12.2: Eksempelspørringen

ses i figur 12.2(a). Her vil det være behov for fire substitusjonsoperatorer.[6]

[6] Legg merke til at det aldri er nødvendig med en substitusjonsoperatorer rett før projeksjonen, siden denne bruker de inverterte indeksene og derfor er uavhengig av inputattributtet.

Dersom man følger helt enkle optimaliseringsregler, jf. [5, avsnitt 16.2], vil man ende opp med det optimaliserte treet i figur 12.2(b). Dette treet fører også til unødvendig bruk av substitusjonsoperatorer, selv om optimaliseringsalgoritmen har fjernet behovet for én av dem ved å gruppere de operatorene som opererer på samme attributt, i én gren av joinoperatoren. Derfor vil plasseringsalgoritmen forsøke å bli kvitt de unødvendige substitusjonene. I dette tilfellet ser vi at substitusjonen i den venstre grenen til joinoperatoren ikke kan fjernes. Derimot kan man i den høyre grenen bli kvitt én av de to substitusjonsoperatorene, dersom man bytter om rekkefølgen på seleksjonsoperatorene. Dette er vist i figur 12.2(c). Denne ombyttingen danner altså grunnlaget for en konflikt mellom optimaliseringsalgoritmen og plasseringsalgoritmen.

Vi ser at optimaliseringsalgoritmen har valgt å utføre seleksjonen på y_1-attributtet først, fordi en seleksjon på likhet kan forventes å fjerne flere tupler i resultatsettet enn en seleksjon på "større enn" ($>$). Dette betyr at situasjonen som er vist i det siste treet, sannsynligvis fører til at både substitusjonen og den siste seleksjon før joinen, må utføres på større mellomlagringer enn dersom rekkefølgen hadde vært slik den er i figur 12.2(b).

For å studere denne konflikten og fullføre eksemplet, kan vi gjøre noen enkle estimater for operatorene og mellomlagringene som brukes på den høyre grenen av joinoperatoren. De eneste aktuelle operatorene er derfor substitusjonsoperatoren og seleksjonsoperatoren. Vi benytter samme notasjon som i [5, avsnitt 16.4], der $T(R)$ er antall tupler i relasjon R, og $V(R, a)$ er antall *distinkte* verdier for attributt a i relasjon R. For seleksjonsoperatoren estimerer vi, jf. [5, avsnitt 16.4], at en seleksjon $\sigma_{a=c}$, der a er et attributt i R og c er en konstant, vil gi en mellomlagring med $T(R)/V(R, a)$ attributtverdier. Videre estimeres en seleksjon på "større enn" ($>$) til å resultere i en mellomlagring med $T(R)/3$ verdier.[7] Siden Rindex versjon 0.1 benytter et binært søk i seleksjonsalgoritmen, kan vi for enkelhetsskyld anta at tidskompleksiteten til denne seleksjonsoperatoren er $\mathcal{C}(n) = \log_2(n) + M$, der n er antall attributtverdier i inputet og M er antall attributtverdier vi har estimert at operatoren skal resultere i. Grunnen til at vi må legge M til kompleksiteten for selve det binære søket, er at algoritmen må løpe gjennom og kopiere disse M verdiene til outputet. I tillegg vet vi at tidskompleksiteten til substitusjonsoperatoren er $\mathcal{C}(n * s) = \mathcal{C}(n)$, fordi vi benytter substitusjonsalgoritme 4 på mellomlagringer som ikke inneholder resultatet av en tidligere joinoperator.

Dersom vi antar at $T(Y) = a$ og $V(Y, y_1) = b$, vil situasjonen vist i figur 12.2(b) medføre følgende operasjoner:

[7] [5] begrunner valget av dette estimatet, og ikke $T(R)/2$, med at slike seleksjoner ofte er konstruert for å hente ut en mindre del av verdiene. Ettersom seleksjoner på "mindre enn" ($<$) eller ulikhet ikke opptrer i dette eksemplet, tar vi ikke med estimatene for disse.

Operasjon	Input-størrelse	Output-størrelse	Tidsforbruk
$\sigma_{y_1=14}$	a	$\frac{a}{b}$	$\log_2 a + \frac{a}{b}$
S_{y_2}	$\frac{a}{b}$	$\frac{a}{b}$	$\frac{a}{b}$
$\sigma_{y_2>5}$	$\frac{a}{b}$	$\frac{a}{3b}$	$\log_2(\frac{a}{b}) + \frac{a}{3b}$
S_{y_1}	$\frac{a}{3b}$	$\frac{a}{3b}$	$\frac{a}{3b}$

Totalt vil dette resultere i en estimert tidsbruk

$$\log_2 a + \log_2 \frac{a}{b} + \frac{8a}{3b} \tag{12.1}$$

På samme måte ser vi at for figur 12.2(c) blir operasjonene:

Operasjon	Input-størrelse	Output-størrelse	Tidsforbruk
$\sigma_{y_2>5}$	a	$\frac{a}{3}$	$\log_2 a + \frac{a}{3}$
S_{y_1}	$\frac{a}{3}$	$\frac{a}{3}$	$\frac{a}{3}$
$\sigma_{y_1=14}$	$\frac{a}{3}$	$\frac{a}{3b}$	$\log_2(\frac{a}{3}) + \frac{a}{3b}$

Her vil den totale tidsbruken estimeres til

$$\log_2 a + \log_2 \frac{a}{3} + \frac{a(2b+1)}{3b} \tag{12.2}$$

Vi legger også merke til at størrelsen på mellomlagringen etter alle at operasjonene er utført, som forventet er den samme ($a/3b$) i begge tabellene.

Siden verdiene i nummerdatabasen er tilnærmet uniformt fordelt, kan vi forvente at $b = T(Y)/D = a/D$, der D er antall mulige verdier i domenet. I tillegg vet vi at $1 \leq b \leq a$ er et krav.[8] Det er altså forholdet mellom antall tupler a i relasjon Y og antall distinkte attributtverdier b for attributtet y_1 i samme relasjon, som er avgjørende for hvilken av de alternative eksekveringsrekkefølgene som er mest effektiv.

For å studere dette forholdet nærmere og forsøke å finne hvilket alternativ som gir den laveste forventede eksekveringstiden, kan vi observere at følgende er ekvivalent for tidsestimatene (12.2) og (12.1), og heltallige a og b:

$$\begin{aligned}
\log_2 a + \log_2 \frac{a}{3} + \frac{a(2b+1)}{3b} &\geq \log_2 a + \log_2 \frac{a}{b} + \frac{8a}{3b} \\
\frac{a(2b+1)}{3b} - \frac{8a}{3b} &\geq \log_2 \frac{a}{b} - \log_2 \frac{a}{3} \\
a(2b-7) &\geq 3b \log_2 \frac{3}{b} \\
a &\geq \frac{3b \log_2 \frac{3}{b}}{2b-7} \qquad \text{for } b \geq 4 \\
a &\leq \frac{3b \log_2 \frac{b}{3}}{2b-7} \qquad \text{for } b \leq 3
\end{aligned}$$

[8] Vi ser for enkelhetsskyld ikke på tomme relasjoner eller attributtverdier som ikke har noen forekomster.

Vi ser at ulikheten bare er oppfylt for $b \geq 4$. Siden vi krever at $1 \leq b \leq a$, vil aldri det siste uttrykket gi mening. Det vil alltid vil gi $a < b$ for $1 \leq b \leq 3$. Dette betyr at estimat (12.1) er det laveste for alle $b \geq 4$, mens (12.2) må være det laveste av estimatene for $b \leq 3$. I dette eksemplet blir det altså illustrert en situasjon der man kan forvente at ordningen som plasseringsalgoritmen velger (figur 12.2(c)) er best i noen b-verdier, det vil si når $b \leq 3$. For alle andre lovlige b-verdier, er optimaliseringsalgoritmens ordning av operatorene (se figur 12.2(b)) mest effektiv.

$\triangle$

For at plasseringsalgoritmen skal kunne avgjøre om to operatorer innen samme område skal byttes, virker det altså nødvendig å ha tilgang til den samme informasjonen som optimaliseringsalgoritmen har brukt for å bestemme rekkefølgen. Dette taler for at det første alternativet er en bedre løsning for implementasjonen av plasseringsalgoritmen, men omfanget av å studere en fullstendig optimaliseringsalgoritme ble for stort for denne oppgaven. Neste avsnitt vil derfor bare gi et kort forslag for en mulig plasseringsalgoritme. Det overlates til andre å analysere og realisere denne eller andre mulige algoritmer.

Algoritmeskisse

Her følger et forslag til skisse for en algoritme som plasserer substitusjonsoperatorer i spørretreet.

Det første punktet i algoritmen er å dele spørretreet opp i mindre områder ved å splitte på de binære operatorene (som join og union). På denne måten kan hvert område behandles isolert. Det kan være en fordel å begynne med de nederste områdene i treet (det vil si de nærmest relasjonene), og fra venstre mot høyre, siden outputet fra disse områdene bestemmer inputet til områdene over i treet. Dette taler for en rekursiv postfiks traversering av treet.

Det sentrale punktet i algoritmen blir å finne den antatt beste ordningen for hver mengde med unære operatorer innen et bestemt område. Dette innebærer altså å vurdere målet om færrest mulig substitusjonsoperatorer opp mot ordningen som optimaliseringsalgoritmen har valgt. Viktige faktorer her er antall substitusjonsoperatorer det er mulig å fjerne, og de ulike estimatene optimaliseringsalgoritmen har basert valgene sine på. Det vil også være nødvendig å bestemme tidskompleksiteten til de andre operatorene i Rindex. På bakgrunn av denne informasjonen, samt estimatene på størrelsene av mellomlagringene som operatorene arbeider på, vil det være mulig å vurdere kompleksitetsforholdet mellom substitusjonsoperatorene og andre de operatorene i det aktuelle området. Dette ble vist i forrige eksempel, men der ble forholdet begregnet eksakt utfra estimatene som ble gjort. I praksis er det sannsynligvis nødvendig med andre tilnærmingsformer, for eksempel heuris-

tikker eller en effektiv numerisk løsningsmetode.

Når en ordning er valgt, kan algoritmen sette inn de nødvendige substitusjonsoperatorene. Legg merke til at det også kan være nødvendig med en substitusjonsoperator mellom input-attributtet og den første av de unære operatorene, eller mellom den siste av de unære operatorene og den binære operatoren. Dette vil også være en viktig faktor i valget av ordning for algoritmen. I tillegg kan man gjøre en vurdering når det gjelder valget av output-attributt i binære operatorer, siden de gir mulighet til å velge blant de to input-attributtene.

Til slutt vil man ende opp med et spørretre som man forventer totalt sett vil gi den beste ytelsen.

Kapittel 13

Avslutning

13.1 Oppsummering

I den siste delen av rapporten har vi studert ulike sider ved substitusjonsoperatoren i form av to problemstillinger som har blitt drøftet i hvert sitt kapittel. Bakgrunnen for valget av dette problemområdet, var at denne operatoren viste seg å være en essensiell flaskehals i Rindex.

13.1.1 Problemstilling 1

I den første problemstillingen stilte vi følgende spørsmål: *Hvordan kan selve substitusjonsalgoritmen forbedres?*

Vi behandlet dette spørsmålet i kapittel 11. Det ble presentert fem ulike forslag til mulige forbedringer av den originale algoritmen. Det første forslaget er en forbedring i utnyttelsen av hash-tabellene i forhold til den originale algoritmen. De fire siste bruker andre datastrukturer, i stedenfor hash-tabeller, for de inverterte indeksene. Hovedidéen her er å basere seg på pekere mellom de ulike strukturene for å kunne gjøre direkte oppslag, og dette viste seg å være gunstig. I tillegg observerte vi en unødig minnebruk og kompleksitet i datastrukturene for primærnøklene i en indeks. For å forbedre dette, ble substitusjonsalgoritme 4 og 5 foreslått. Algoritme 5 ble vist å være optimal med hensyn på tidskompleksiteten. Til slutt ble de ulike forslagene sammenlignet teoretisk og empirisk. Testene ble utført ved å kjøre et utvalg SQL-spørringer mot hver substitusjonsalgoritme, og med ulike størrelser på datasettet. Det viste seg at målingene fra disse testene stemte godt overens med forventningene fra analysen.

For problemstilling 1 konkluderte vi med at substitusjonsalgoritme 4 var det beste alternativet. Denne vurderingen bygger på analysen og testene som er gjort her. Det er sannsynlig at også andre faktorer kan spille inn på valget av algoritme.Et slikt eksempel er den innvirkningen valget av algoritme og datastruktur har på de andre operatorene, med hensyn både på effektiviteten og minnebruken.

13.1.2 Problemstilling 2

I kapittel 12 behandlet vi problemstilling 2: *Hvordan kan antall substitusjonsoperatorer i en spørring reduseres?*

Målet her var å konstruere en algoritme for å minimere antall nødvendige substitusjonsoperatorer i en spørring. Dette omfatter hovedsaklig bruken og plasseringen av substitusjonsoperatorene i spørretreet. Det viste seg imidlertid å være mer komplisert enn forventet å konstruere en slik algoritme, fordi den er avhengig av tilgang til den samme informasjon som optimaliseringsalgoritmen baserer sine valg på. Det ble også vist at en slik algoritme ofte kan komme i konflikt med enkelte av operatorordningene som optimaliseringsalgoritmen bruker, og at en reduksjon i antall substitusjonsoperatorer ikke nødvendigvis vil føre til en mer effektiv eksekvering, siden operatorene dermed må arbeide på større mellomlagringer. Det ble vist et eksempel som bekrefter denne konflikten. Som også vist i forrige kapittel, vil optimaliseringsalgoritmen ofte redusere antall nødvendige slike operatorer, og derfor blir det mindre mulighet for å redusere dette antallet ytterligere.

På grunn av forholdet mellom plasseringsalgoritmen og optimaliseringsalgoritmen, ble det for omfattende å utlede algoritmen fullstendig i denne rapporten.

13.1.3 Styrker og svakheter

Som nevnt i innledningen (avsnitt 1.1.1), valgte vi å begrense oppgaven på en rekke områder. For problemstilling 1 er den viktigste begrensingen at vurderingene bare er gjort med hensyn på de statiske datastrukturene som er brukt i versjon 0.1 av Rindex. Problemstilling 2 påvirkes ikke av dette, siden den bare omhandler *bruken* av substitusjonsoperatoren. I kapittel 11 er det fokusert spesielt på datastrukturen for de inverterte indeksene, og den er designet med samme statiske utgangspunkt som indeksene i Rindex. Dette betyr at de inverterte indeksene vil egne seg like dårlig for oppdateringer som indeksene. Det er altså ikke tatt nok hensyn til senere forbedringer av plattformen når det gjelder vurderingene som er gjort for de inverterte indeksene.

Det er verdt å legge merke til at for mellomlagringer vil den foreslåtte datastrukturen egne seg like godt for både statiske og dynamiske indekser. Årsaken er at dataene i en mellomlagring ikke endres mens strukturen eksisterer, og de kan derfor alltid betraktes som statiske. Datastrukturen for mellomlagringene er ikke nødvendigvis optimal, men det er valgt å bruke samme struktur på indekser og mellomlagringer for at algoritmene ikke skulle kompliseres unødvendig.

Den positive siden er at substitusjonsoperatoren har blitt markant effektivisert ved bruk av de foreslåtte forbedringene. Dette har gjort det mulig og interessant å sammenligne Rindex med eksisterende DBMSer, slik det

er gjort i Kjell-Magne Øieruds masteroppgave [19]. I tillegg er det antydet muligheter for videre forbedringer.

13.2 Videre arbeid

Det er allerede nevnt en del viktige punkter for videre arbeid med Rindex-plattformen i kapittel 9. I tillegg kan det nevnes noen andre områder som er knyttet til den siste delen av rapporten.

Siden dynamiske indekser er en nødvendighet for å støtte oppdateringer effektivt, må valgene og vurderingene som er gjort i denne rapporten revurderes når de dynamiske datastrukturene skal designes. Det er forsket en del på dynamiske indeksstrukturer for bruk i minnet, og de tradisjonelle B+-trærne har vist seg være godt egnet også når hele strukturen ligger i minnet, jf. [18]. I den senere tid er det er blitt presentert noen nye datastrukturer for dette formålet, for eksempel CSS- og CSB+-trær. CSS-trær ("Cache Sensitive Search Trees") [13] gir raskere oppslag enn B+-trær og T-trær, hovedsaklig på grunn av utnyttelsen av cachen, men er designet for mer statiske data. CSB+-trær ("Cache Sensitive B+-Trees") [14] er inspirert av CSS-designen til å utnytte cachen bedre enn tradisjonelle B+-trær. CSB+-trær fremstår derfor som et godt alternativ for en dynamisk indeksstruktur.

Selv om en del ulike forslag, inkludert en optimal algoritme, er presentert for substitusjonsalgoritmen, kan det være mulig å komme opp med nye innfallsvinkler og idéer for implementasjonen av substitusjonsoperatoren og datastrukturene til de inverterte indeksene, spesielt når det gjelder design av dynamiske strukturer.

Det kan også være interessant å vurdere hvordan endringene i datastrukturene for de inverterte indeksene og primærnøklene, påvirker de andre operatorene og algoritmene i indekshåndteringen i Rindex. Dette kan, som tidligere nevnt, ha innvirkning på hvilken substitusjonsalgoritme man foretrekker å bruke. I tillegg vil det være nødvendig å studere hvor effektivt cachen utnyttes i de ulike algoritmene.

Man bør forsøke å fullføre analysen, designen og implementasjonen av algoritmen for å plassere substitusjonsoperatorene i spørretreet, siden dette kan ha positiv innvirkning på effektiviteten for eksekveringen av en spørring. En del av dette kan være å undersøke reglene, heuristikkene og estimatene som en tradisjonell optimaliseringsalgoritme bruker, og hvilke av dem som er fornuftige for Rindex. I tillegg kan det være nyttig å utlede tidskompleksiteten til de andre operatorene i Rindex, for å kunne sammenligne dem med substitusjonsoperatoren (slik det ble vist i eksempel 12.1).

Bibliografi

[1] Manindra Agrawal, Neeraj Kayal, and Nitin Saxena. *PRIMES is in P*. Technical report, Department of Computer Science & Engineering, Indian Institute of Technology Kanpur, 2002.

[2] Daniel J. Bernstein. *djb2 hash*. Internett: `http://www.cs.yorku.ca/~oz/hash.html`.

[3] Glenn Fowler, Landon Curt Noll, and Phong Vo. *FNV hash*. Internett: `http://www.isthe.com/chongo/tech/comp/fnv/`.

[4] Free Software Foundation. *Bison 1.35*. Internett: `http://www.gnu.org/software/bison/manual/`.

[5] Hector Garcia-Molina, Jeff Ullman, and Jennifer Widom. *Database Systems: The Complete Book*. Prentice Hall, 2001.

[6] Robert John Jenkins Jr. *Hash Functions for Hash Table Lookup*. Internett: `http://burtleburtle.net/bob/hash/evahash.html`.

[7] Brian W. Kernighan and Rob Pike. *The Practice of Programming*. Addison-Wesley, 1999.

[8] Donald Erwin Knuth. *The Art of Computer Programming*, volume 3: Sorting and Searching. Addison-Wesley, 2nd edition, 1998.

[9] J. R. Levine, T. Mason, and D. Brown. *lex & yacc*. O'Reilly & Associates, Inc., 2nd edition, 1992.

[10] Kenneth C. Louden. *Compiler construction: principles and practice*. PWS Publishing Company, Boston, Massachusetts, USA, 1997.

[11] Vern Paxson. *Flex, A fast scanner generator*. Free Software Foundation. Internett: `http://www.gnu.org/software/flex/manual/`.

[12] Igor V. Rafienko. *Filmdatabasen*. Technical report, Institute of Informatics, University of Oslo, 2002.

[13] Jun Rao and Kenneth A. Ross. *Cache Conscious Indexing for Decision-Support in Main Memory.* In *VLDB '99: Proceedings of the 25th International Conference on Very Large Data Bases*, pages 78–89. Morgan Kaufmann Publishers Inc., 1999.

[14] Jun Rao and Kenneth A. Ross. *Making B+- trees cache conscious in main memory.* In *SIGMOD '00: Proceedings of the 2000 ACM SIGMOD international conference on Management of data*, pages 475–486. ACM Press, 2000.

[15] Eric S. Raymond. *The Art of UNIX Programming.* Addison-Wesley, 2004.

[16] Marc J. Rochkind. *Advanced UNIX Programming.* Addison-Wesley, 2nd edition, 2004.

[17] SQLAPI++. *SQLAPI++ Library.* Internett: `http://www.sqlapi.com`.

[18] Rune Storløpa. *Indekser i RAM-databaser.* Master's thesis, Institute of Informatics, University of Oslo, 2000.

[19] Kjell-Magne Øierud. *Rindex, Ytelsestest av plattform for indekshåndtering i RAM.* Master's thesis, Institute of Informatics, University of Oslo, 2005.