UNIVERSITETET I OSLO
Institutt for Informatikk

Fritt søk med FAST søkemotor integrert i PostgreSQL relasjonsdatabase

Hovedoppgave

Åsmund Kveim Lie
(asmundkl@ifi.uio.no)

November 2004

ABSTRACT

Focus of this thesis is the relationship between databases and information retrieval systems. As a background, the first part consists of a general presentation of databases and information retrieval systems and some examples of already existing efforts to combine the two. While these examples typically have expanded either a database system or an IRS to obtain multi-functionality, we have made an effort of bridging the two systems.

Our prototype integrates FDS (Fast Data Search) into the PostgreSQL database management system as a new index access method. FDS is a powerful and scalable commercial enterprise search platform using a typical search engine query language. PostgreSQL, being open source and a general basis for research, lends itself well to customization. The new index access method provides the database with powerful free text capabilities while retaining the power of the relational model for structured data. Preliminary results including a simple performance test verify the feasibility of the integration, and demonstrate the scalability of the prototype. Storage, indexing, updating and search functions are implemented, but ACID properties could not be guaranteed, because the external indexing system has no such guarantee.

I also present a prototype for automatic extraction of related structured data in the relational database to XML. Combining these two prototypes by allowing the extracted information to be searched using the full text index, makes it possible to search the database without knowledge of the underlying database scheme.

Finally I discuss potential expansions of our implementation by indexing other data than text, multicolumn-indexing and moving complex evaluation from PostgreSQL to FDS, and suggest how this could be done.

The thesis is written in Norwegian.

INNHOLDSFORTEGNELSE

KAPITTEL 1: INNLEDNING

Attempt the end and never stand to doubt;
Nothing's so hard, but search will find it out.
- Robert Herrick, Seeke and Finde (1648) (hentet fra [1])

Da min far begynte på skolen i gruvebyen Sulitjelma i 1945 var det en telefon til familien på Østlandet per år. Denne måtte bestilles om morgenen og kl 17 var alle linjene koblet sammen og en kort samtale kunne finne sted. Da jeg etter avsluttet skolegang reiste som ryggsekkturist i Kambodsja i 2000, hadde jeg nesten daglig e-postkontakt med familie og venner hjemme i Norge.

Internett har forenkelt kommunikasjon mellom mennesker og tilgjengeliggjort enorme mengder informasjon, i det siste også opplysninger om privatpersoner - "Jeg googles, altså er jeg"[2]. Jo mer data som lagres, jo større blir kravene til dokumenthåndtering. Data som ikke enkelt kan gjenfinnes har liten verdi. Informasjonsgjenfinningssystemer, laget for å søke i tekst, fikk et utvidet bruksområde og ble i stigende grad et nødvendig hjelpemiddel med internetts voldsomme vekst. Google har i skrivende stund indeksert 4 285 199 774 internettsider[3].

Informasjonsgjenfinning og databaseteknologi er "gamle" grener innen informatikken. Databasesystemer ble utviklet for å håndtere strukturerte data med stor oppdateringsrate og samtidighet. Klassiske eksempler er forretningsbanker og helseregistre. Informasjonsgjenfinningssystemer er utviklet for å kunne søke i ustrukturert data – hovedsaklig tekst. Det typiske er få oppdateringer og mange søk med rangering av resultatet. Klassisk eksempel er artikkelsamlinger. I nyere tid ser en økende behov for å benytte seg av alle disse funksjonene i et integrert system.

For eksempel er informasjonsmengden som lagres internt i bedrifter stadig voksende. I dag inneholder ofte en produktdatabase også ikke-tradisjonell databaseinformasjon som bilder, brosjyrer og utfyllende beskrivelser. "Det papirløse samfunn" medfører behov for arkivsystemer ikke bare for brev, men også for e-post og en flom av interne notater. Oracles nestkommanderende, Chuck Phillips, hevdet nylig at datamengden i bedrifter vil minst fordobles hvert år fremover [4]. Dette er eksempler som har medført behov for både mer avansert søkefunksjonalitet internt i databaser og mulighet for å søke i en database som en del av et omfattende søkesystem. Denne problemstillingen ønsket jeg å se nærmere på i min hovedfagsoppgave og presenterer en prototyp som tilbyr både mulighet for fritt søk i tekst i en database og for fritt søk i databasen generelt uten hensyn til relasjonsstrukturen.

Å integrere informasjonsgjenfinningsteknologi og databaser har lenge vært et mål for å kunne tilby transaksjonsstøtte til dokumenter [5]. Det er også gjort en del forskning for å tilby integrert funksjonalitet bl. a. [6-8]. Standardiseringsorganet ISO/IEC har definert hvordan fulltekstsøk bør integreres i databaser med SQL/MM [9;10]. I kommersielle applikasjoner ser en at

leverandører tilstreber å gjøre dette fra hver sin kant. Oracle og Microsoft SQL Server tilbyr begge fulltekstsøk i sine databaser [11;12]. *FDS* (FAST Data Search) [13;14] tilbyr en utvidbar søkemotor som nå også overtar søk i tradisjonelle databaseapplikasjoner som eksempelvis søk i Gule Sider[15].

I eksemplene ovenfor har man utvidet et eksisterende system for å oppnå multifunksjonalitet. Jeg ønsket isteden å bygge en "bro" mellom to ledende systemer innen hver sin kategori. Jeg tok utgangspunkt i PostgreSQL[16;17] som relasjonsdatabase og FDS som informasjonsgjenfinningssystem.

I kapittel 2 og 3 gjennomgår jeg generelle egenskaper ved henholdsvis relasjonsdatabaser og informasjonsgjenfinningssystemer og presenterer PostgreSQL og FDS. Dernest, i kapittel 4 gir jeg en innføring i forskjellige tilnærminger til kombinert funksjon, der jeg presenterer så vel kommersielle løsninger som standarder og forskningsarbeid. I kapittel 5 beskriver jeg vårt arbeide for å oppnå en integrasjon mellom disse systemene ved å implementere en ny indeksaksessmetode i PostgreSQL, der selve indeksen ligger i FDS. Her går jeg inn på arbeidet som ble gjort både mot FDS og PostgreSQL. Med dette kunne vi tilby fritekstsøk i PostgreSQL. Videre, i kapittel 6, vurderer jeg hvorvidt vi har oppnådd en fullstendig integrasjon med prototypen, og presenter resultatet av en enkel ytelsestest og viser at integrasjonen ikke bare tilfører ny funksjonalitet, men også skalerbar ytelse. I kapittel 7 trekker jeg dette arbeidet videre gjennom en prototyp for analyse av systemtabeller og automatisk uttrekking av strukturert data til en semistrukturert XML-fil. Denne kan direkte indekseres av indeksaksessmetoden som dermed tilbyr fritt søk uavhengig av strukturell plassering av data i databasen. Videre utreder jeg muligheten for utvidelser av prototypen for å tilby en mer fullstendig integrering og avslutter med en konklusjon i kapittel 8.

Arbeidet med å utvide PostgreSQL med en ny indeksaksessmetode for å tilby fulltekstsøk og utføring av ytelsestest av prototypen, ble gjort som et samarbeid med Håkon Clausen [18]. Det var nesten en forutsetning å være to for å få tilstrekkelig oversikt til å utføre arbeidet, fordi vi måtte sette oss inn i store mengder nytt stoff, både om informasjonsgjenfinningssytemer generelt og detaljkunnskaper om FDS som er et meget omfattende og komplisert system. Dessuten måtte vi sette oss inn i hvordan PostgreSQL-kildekoden er bygget opp og kan videreutvikles. Bare kildekoden til PostgreSQL består av over en halv million linjer. Vi arbeidet veldig godt sammen. Jeg vil takke Håkon for et spennende og produktivt samarbeid og godt vennskap.

Jeg vil også takke min veileder Knut Omang for inspirasjon og gode råd, og tålmodig oppfølging underveis. Til slutt en varm takk til familie og venner for støtte og oppmuntring og til Nosyko for en fleksibel arbeidssituasjon.

KAPITTEL 2: DATABASER

I dette kapittelet gir jeg en kort oversikt over hovedegenskapene til en database, med vekt på de deler som har betydning for vår oppgave og litt om PostgreSQL som er databasen vi har valgt å bruke i oppgaven vår. Siden det forutsettes en del forhåndskunnskap om databaser, vil dette kapittelet ikke være så detaljert.

Hva er en database? I boken *Fundamentals of database system* [19] defineres en database som en samling av relaterte data, og det listes opp følgende implisitte egenskaper :

- Den representerer et subset av den virkelige verden, ofte kalt *UoD* (Universe of Discourse). Endringer i virkeligheten blir reflektert i databasen.
- Det er en logisk koherent samling av data med en form for naturlig sammenheng/mening.
- En database er designet, laget og fylt med data for et bestemt formål.

Med andre ord kan en si at en database har en kilde i den virkelige verden hvor det hentes data fra, en eller annen form for interaksjon med hendelser i den virkelige verdenen og et sett med brukere som er aktivt interessert i innholdet.

2.1 Databasesystemer, DBMS

En av fordelene ved å benytte en database er at mange års utvikling har ført til stabil og kraftig programvare for å manipulere dem, kalt *database managment system*, DBMS. Det er endog hevdet at en database i vanlig språkbruk simpelthen kan defineres som en samling data som kan håndteres av et DBMS [20]. Hovedfunksjonene et DBMS skal tilby er:

- *Persistent lagring* av store mengder data som eksisterer uavhengig av andre prosesser som måtte behandle dem og datastrukturer for effektiv tilgang til dem.
- *Pragrammeringsgrensesnitt*. Et DBMS tilbyr brukere og applikasjoner et kraftig spørrespråk for å tilgjengeliggjøre og modifisere data.
- *Transaksjonsstøtte*. Et DBMS tilbyr samtidig tilgang til data fra flere prosesser av gangen. For å unngå feil ved samtidig bruk må den støtte transaksjoner ved å tilby ACID egenskapene (atomisitet, konsistens, isolasjon og varighet). ACID omtales nedenfor (2.3).

De første DBMS kom allerede på slutten av 60 tallet og var systemer som var videreutviklet fra filsystemet. Disse var ikke fullstendige DBMS'er slik de er beskrevet ovenfor, men tilbød en måte å strukturere store mengder data. Videre var brukeren avhengig av å vite hvordan data var lagret og en benyttet seg hovedsaklig av to datamodeller; hierarkiske- (tre baserte) og nettverks- (graf baserte) databaser. Problemet med begge disse var at de ikke støttet et høynivåspørrespråk. Eksempelvis støttet en i CODASYL's (Committee on Data Systems and Languages) grafdatabase kun spørrespråk for gå fra en node til en annen basert på pekere.

2.2 Relasjonsdatabaser, RDBMS og spørrespråket SQL

I 1970 kom en berømt artikkel fra E. F. Codd [21] som forandret denne tankemåten. Codd foreslo at brukeren kun skulle forholde seg til at data var organisert i tabeller som han kalte *relasjoner*. Bak kulissene kunne det være avanserte datastrukturer for både lagre og hente frem data, men det behøvde ikke brukeren å forholde seg til. Med andre ord ønsket han å skille mellom hvordan data er fysisk og logisk lagret. Dette resulterte i at brukeren kunne uttrykke spørringer i et høynivåspråk. Codd introduserte også *relasjonsalgebraen* som et slikt spørre- og manipulasjonsspråk. Relasjonsalgebraen inneholder et sett med operasjoner for å manipulere relasjonsmodellen. Resultatet av en slik spørring er en ny relasjon, som en igjen kan utføre operasjoner på.
Relasjonsalgebraoperasjoner deles hovedsaklig inn i to kategorier: En gruppe, arvet fra matematikken, går på mengdeoperasjoner som snitt og union, mens en annen gruppe er utviklet spesielt for relasjoner med operasjoner som seleksjon, projeksjon og join. Satt sammen i vilkårlig komplekse uttrykk viste det seg at en kan løse de aller fleste oppgaver, noe som sammen med at det lot seg implementere effektivt gjorde at dette fikk en veldig oppslutning.

Databasesystemer som baserer seg på denne tankegangen blir referert til som relasjonsdatabaser og *RDBMS* (Relational Database Management System). Videre i oppgaven vil jeg følge innarbeidet praksis og referere til relasjonsdatabaser og databaser om hverandre i betydningen relasjonsdatabaser med mindre eksplisitt angitt.

Det mest benyttede spørrespråket i relasjonsdatabaser i dag er *SQL* (Standard Query Language)[1]. SQL understøtter de fleste operasjoner som kan uttrykkes i relasjonsalgebra, men har i tillegg også muligheter for å manipulere data og databaseskjemaer. SQL finnes i en rekke versjoner og dialekter fra den første ANSI SQL, en oppdatert ny versjon i 1992 kalt SQL2 og SQL-92 til den siste som kom i 2000 kalt SQL-99. SQL-99 utvider SQL-92 med objektsrelasjonelle egenskaper. F.eks. kan datatyper i SQL-99 tilordnes metoder og attributter ala klasser i objektverdenen. De fleste kommersielle aktørene på markedet i dag har implementert mesteparten av SQL-92 samt deler av SQL-99 og noen egne utvidelser. Så selv om de fleste databaseleverandører hevder at de følger standarden er det tvilsomt at det å overføre en applikasjon fra en databaseleverandør til en annet vil kunne utføres uten å måtte gjøre endringer.

2.3 Transaksjoner

Som nevnt i starten av kapittelet er transaksjonsstøtte en av de viktigste egenskapene som kreves av en moderne RDBMS. Jeg vil her gi en kort innføring i transaksjoner siden det blir diskutert hvordan vår implementasjon overholder transaksjoner og hvilke utvidelser som må til for å støtte det i kapittel 6.1.2.

En transaksjon er en logisk enhet bestående av et sett av operasjoner utført mot databasen[20]. Mer konkret en gruppering av SQL-kommandoer hvis resultat blir synlig for resten av systemet som en enhet når transaksjonen fullfører [commits] -- eller ikke[22]. I et RDBMS er det vanlig å si at en transaksjon må overholder

[1]Ofte også uttalt som sequel og referert til som ”the universal query language”

de såkalte ACID egenskapene. De har sitt navn fra en sammentrekning av den engelske betegnelsen for fire egenskaper:

Atomisitet (Atomisity) betyr at i en transaksjon skal alt eller ingenting utføres. Eksempelvis kan en transaksjon bestå av at en overfører penger fra en konto til en annen. Det skal da ikke være mulig at en kun tar ut på den ene kontoen, men ikke setter inn på den andre og omvendt.

Konsistens (Consistency) betyr at en database alltid skal gå fra en ”korrekt” tilstand til en annen. Med andre ord skal det ikke være lov at en transaksjon fører til at data i databasen ikke overholder skjema og regel definisjoner. Eksempelvis kan en regel si at det ikke kan være minus beløp på en konto. Ingen transaksjon har derfor lov til å avslutte hvis det medfører brudd på denne reglen. Enkelte operasjoner innen en transaksjon kan imidlertid bryte disse så sant databasen er i en konsistent tilstand når transaksjonen fullfører.

Isolasjon (Isolation) betyr at enhver transaksjon skal utføres slik at den ikke får noe annet resultat enn om den kjørte alene på databasen (ingen innvirkning fra andre transaksjoner). Dette betyr at ingen operasjoner utenfor en pågående transaksjon kan se mellomresultatene fra den.

Varighet (Durabililty) betyr at endringer påført av en ferdig transaksjon aldri skal gå tapt selv om databasesystemet feiler/går ned.

For å oppnå varighet av data og atomisitet av transaksjoner må enhver forandring i databasen logges separat til disk med informasjon om hvilken transaksjon som utførte den. Det finnes flere forskjellige fremgangsmåter for å utføre logging. Alle medfører at uansett når en systemfeil inntreffer så vil en gjenopprettingsalgoritme kunne lese loggen for å sette databasen til en konsistent tilstand. Dette skjer blant annet ved kun å utføre de endringer som ble utført av en fullført transaksjon. Loggen kan da brukes for å ”spole tilbake” databasen (såkalt undo log) eller så kan en gjøre de operasjonene som faktisk ble vellykkede (redo log). Begge teknikker har fordeler og ulemper, men jeg vil ikke gå inn på de her.

Siden flere transaksjoner kan utføres samtidig kan dette medføre at isolasjon ikke er oppfylt. Et eksempel er der en transaksjon har oppdatert en verdi som en annen leser før den første er ferdig (kalt skitten les). Hvis så den andre baserer seg på disse data og den første transaksjonen blir avbrutt slik at endringen må omgjøres vil resultatet bli feil. Et annet problem er når to transaksjoner skal oppdatere samme felt basert på verdien som var der. Hvis de begge baserer seg på den samme verdien kan den oppdateringen som utføres først bli tapt (kalt tapt oppdatering). En streng grad av isolasjon kalles for *serialiserbarhet*. Det betyr at selv om transaksjoner overlapper i tid skal de ha samme effekt som om de ble utført etter hverandre, *serielt*, i tid. En ønsker jo nemlig at flere transaksjoner skal kunne utføres samtidig siden det vil tilhøre unntakene at det oppstår konflikter. Likevel må en unngå at potensielle feil oppstår. Det finnes også flere andre, mindre strenge isolasjonsnivåer som benyttes for å kunne øke samtidighet og forekle implementasjon. Eksempelvis benytter PostgreSQL seg av et annet, kalt

read commited, som betyr at en transaksjon kan se resultatet av andre fullførte transaksjoner selv om dette kan bryte med serialiseringsprinsippet da den som fullførte kan ha startet etter en pågående. For å løse dette må også databasen tilby samtidighetskontroll. Den mest brukte måten å løse dette på er å benytte seg av låser på deler av databasen. En transaksjon vil da ta en lås på den delen av databasen den skal operere på slik at andre transaksjoner ikke kan interferere med den første. Den vanligste måten å garantere serialiserbarhet ved hjelp av låser er at en transaksjon som først har tatt en lås på en del av databasen ikke vil slippe den før den er ferdig med transaksjonen, men også andre teknikker som benytter seg av tidsstempling av transaksjoner, gjerne kombinert med flere versjoner av data i databasen kan benyttes. Fordelen med de siste variantene er at en kan øke samtidigheten da transaksjonene ikke holder på låser lenger enn strengt tatt nødvendig noe som bl.a. PostgreSQL benytter seg av.

Enhver samtidig applikasjon som benytter seg av låser kan risikere vranglåssituasjoner. Enkleste eksemplet er to transaksjoner som hver har tatt en lås, men som gjensidig venter på at en av hverandres låser skal bli frigitt. Et DBMS må derfor også ha mulighet for å unngå at slike situasjoner oppstår, eventuelt ha mulighet til å oppdage det når det har skjedd for så å kunne håndterere det. Det er sjelden et DBMS har mulighet for å forhindre vranglåssituasjoner. Det er fordi teknikkene brukt for å løse det enten har urealistiske antagelser, som at en transaksjon på forhånd vet akkurat hvilke låser det vil være behov for, eller de potensielt medfører veldig mye ekstraarbeid og slik kan minske samtidigheten. Det normale er derfor at en har mulighet for å oppdage vranglåser. En teknikk er bruk av ”venter på” grafer der hver transaksjon som venter på en annen blir en node i grafen med en kant til den transaksjonen den venter på. Denne grafen må så jevnlig analyseres og er det en syklus i denne er det en vranglås. En må da ha algoritmer for å avgjøre hvilken av transaksjonene som skal avbrytes. En annen enklere metode benytter tidsavbrudd og denne velges ofte på grunn av det lille ekstraarbeidet som må til. Bruker en transaksjon lenger tid enn en forhåndsdefinert frist kan en anta at den er i en vranglåssituasjon og den vil bli avbrutt uavhengig om det er tilfellet eller ikke. Endelig må en unngå at transaksjoner blir ”utsultet” ved bruk av låser. Det kan skje hvis algoritmene for fordeling av låser ikke er rettferdige og en transaksjon alltid kommer bakerst i køene slik at den ikke får de låser en trenger og ikke får fullført.

2.4 Indeksaksessmetoder

I en database kan det finnes milliarder av tupler og en er ofte kun ute etter ett eller en mindre delmengde som har en ønsket egenskap. Gitt at en skulle finne kontobeholdningen til en bestemt person. Det ville da vært ganske ineffektivt å lese alle kontoene for så å stoppe når en har funnet den som passet (kalt sekvensielt søk). I stedet for kunne en benyttet seg av en eller annen form for datastruktur som kunne raskt fortelle hvor på disken kontobeholdningen til personen lå. Dermed trengte databasesystemet kun å lese den blokken der de aktuelle data lå. For å løse denne typen oppgaver benytter en seg av indeksaksessmetoder.

Enkelt definert er en indeksaksessmetode en eller annen form for datastruktur som tar en søkenøkkel som parameter og som kan finne de tupler som tilfredstiller nøkkelen ”fort”. Fort vil i de aller fleste tilfeller bety færrest mulig diskoperasjoner, antall blokker som må leses opp. En kan derfor forsvare at en slik indeksaksessmetode også må lese fra disk og utføre komplekse beregninger så lenge antallet diskoperasjoner minskes. Tradisjonelle indeksaksessmetoder vil ofte være såpass små at de kan få plass i internminnet og dermed minske søketiden betraktelig. Enkelte indeksaksessmetoder kan også benyttes for å overholde unikhet, eksempelvis for primærnøkler. Data i en indeks kan også være sortert slik at den kan optimalisere ”større enn” og ”mindre enn” søk.

De tradisjonelle databaseindeksaksessmetodene har stort sett kun hatt enkle nøkler som tall eller enkeltord og har bestått av nøkkelen samt en peker til hvor på disken tuppelet måtte befinne seg. Eksempler på indeksaksessmetoder som er mye brukt i databaser i dag er B-trær (primært B+-trær) og hashindekser. Med andre ord er nøkkelen hele innholdet i et attributt, eksempelvis kontonummer, og pekeren vil peke på de tupler som har eksakt det kontonummeret, og alle vil være like relevante. I oppgaven ser jeg jo på integrasjon av et IR-system inn i en relasjonsdatabase og da ved å bruke søkemotoren som en indeksaksessmetode. De datastrukturer som tradisjonelt benyttes i et IR-system er ganske forskjellig. For det første er det ikke lenger snakk om eksakte treff, men om ”handler om” og alle treff vil heller ikke lenger være like interessante. Videre vil slike indekser være betydelig større enn datamengden som indekseres, men likevel vil et slik system svare på spørringer mye hurtigere enn et sekvensielt søk hvis det i det hele tatt hadde vært mulig å besvare uten å være indeksert.

Spørreplanleggeren vil, ved søk, evaluere hvilken måte den best kan svare på en spørring basert på statistikk bygget opp under kjøring. Det er ikke gitt at det alltid vil lønne seg å bruke en indeks f.eks. hvis begrensningen i spørringen resulterer i at en likevel skal hente opp de fleste tuplene. I et slikt tilfelle vil det lønne seg å lese opp hele relasjonen da denne ofte ligger etter hverandre på disken (på den måten får du opp flere tupler per blokk). Alternativet, bruk av en indeks, blir at en må gjøre en diskleseoperasjon per tuppel en skal ha opp. Oppdateringer i databasen vil også ta lenger tid når en benytter seg av indekser da disse også må oppdateres. Derfor må en, avhengig av bruken av databasen, vurdere hvor mange indekser en ønsker seg. Så lenge det er snakk om tette (dense) indekser kan også andre typer spørringer løses ved at en kun ser på data i indeksen uten å trenge å hente opp tuppelet. Eksempler på slike spørringer er EXISTS-spørringer, og aggregeringer.

2.5 PostgreSQL

PostgreSQL[23] er den mest avanserte relasjonsdatabasen basert på åpen kildekode i dag. Den er en videreutvikling av POSTGRES [24;25]. POSTGRES ble utviklet som et forskningsprosjekt på universitetet i California på Berkeley Computer Science Department som igjen en oppfølger av deres forskningsdatabase INGRES (POST inGRES). Prosjektet som var ledet av Michael Stonebraker var sponset av bl.a. DARPA (Defense Advanced Research Projects Agency) og NSF (National Science Foundation) og banet vei for mange

av de objektsrelasjonelle konseptene vi nå ser i kommersielle databaser, bl.a. brukerdefinerte datatyper.

POSTGRES gikk gjennom flere versjoner og ble etter hvert også brukt i en del eksterne prosjekter. Illustra Information Technologies, senere del av Informix brukte koden og kommersialiserte den. Senere ble kildekoden videreutviklet som et åpenkildekodeprosjekt, først som Postgres95 som gjorde om spørrespråket fra PostQUEL til SQL og senere skiftet navn til PostgreSQL.

PostgreSQL er i stadig utvikling. Programmet består for tiden av 500.000+ linjer kode og 200.000+ har kommet bare de siste to årene. PostgreSQL brukes fremdeles mye i forskningsøyemed fordi en kan forandre i kildekoden og fordi den baserer seg mye på systemtabeller.

PostgreSQL oppfyller alle ACID-kravene og det er spesielt to teknikker som benyttes for at disse blir ivaretatt [22;26] *Write Ahead Log* (WAL) og *Multi Version Concurrency Control* (MVCC). WAL er en form for redo-log teknikk som sikrer at en fullført transaksjon alltid vil være bevart ved at logg over transaksjonen vil bli skrevet til disk før den erklæres ferdig. Går systemet ned kan det bli gjenopprettet ved at ferdige transaksjoner som enda ikke er skrevet til disk blir skrevet til disk, mens de som ikke var fullført vil bli ignorert. Dette vil ivareta kravene om atomisitet og varighet. MVCC er en samtidighetskontrollteknikk som medfører at databasen vil ta vare på flere eldre versjoner av et tuppel i tillegg til den gjeldende. Vanligvis må en transaksjon T2 som leser en verdi som T1 har skrevet vente til T1 har avsluttet og eventuelt avbryte med denne for å unngå skitten les, men i PostgreSQL løses dette ved at i stedet for leser T2 den ”gamle” verdien som var før T1 skrev til den. Denne teknikken ivaretar isolasjon samtidig som den åpner for et høyere nivå av samtidighet ved at den minsker antall transaksjoner som må avbrytes. Konsistens ivaretas ved at transaksjoner som forsøker å fullføre etter at den har gjort endringer som ikke er i tråd med databasebetingelsene vil bli avbrutt.

De aller fleste RDBMS bruker systemtabeller for å lagre metadata. Dette blir på en måte en minidatabase om databasen. Her lagres informasjon om en relasjons attributter, views osv. Informasjon om indekser lagres også, men ofte kun for å bli benyttet av spørreplanleggeren slik at den kan ta et godt valg. Her kan det finnes informasjon om at det finnes indekser og statistikker som viser hvordan data er fordelt. PostgreSQL skiller seg ofte litt ut fra kommersielle RDBMS’er fordi den har et veldig høyt bruk av systemtabeller noe som gir store muligheter for utvidelser/eksperimentering uten at en nødvendigvis må endre på selve kildekoden. I PostgreSQL har en også mulighet for å dynamisk laste opp funksjoner skrevet i C kompilert inn i en objektfil. Disse egenskapene er noe vi benyttet oss av i vår prototyp da PostgreSQL ble benyttet som database i vårt integrerte system.

KAPITTEL 3: INFORMASJONSGJENFINNING, IR

Informasjonsgjenfinning (Information Retrieval), IR er en gammel gren innen informatikk, sett med dataøyne, og har vært fokus for forskning siden 60-tallet [27]. I litteraturen brukes søkemotorer og informasjonsgjenfinning om hverandre. Jeg kommer utover i oppgaven til å referere til IR, informasjonsgjenfinning, som en vitenskap med en rekke velutviklede teknikker og teorier og en søkemotor som en applikasjon som benytter seg av dette. Søkemotorer for web er nyere, og dukket ikke opp før på slutten av 90 tallet. I begynnelsen hadde ikke disse søkemotorene særlig sterke informasjonsgjenfinnings egenskaper. Søkemotorer som AltaVista ga deg gjerne 100.000 treff, men du var heldig hvis dokumentet du lette etter lå blant de første 20. I den senere tid har søkemotorer inkludert mer og mer av det som er å regne som informasjonsgjenfinningsegenskaper.

Informasjonsgjenfinning går i hovedsak ut på å finne dokumenter som handler om et bestemt tema. Og allerede her er kanskje ett av de største problemene. Hva handler et dokument om? Har du en artikkel om trefase-commit, vil du til en medstudent si at den er om distribuerte transaksjoner, til en annen at den er om databaser, til fetter at den er om at han ikke skal miste penger i en banktransaksjon og til mormor at den er om data!

Ved tradisjonelt søk i databaseverdenen vil et tuppel tilfredsstille et kriterium eller ikke – eksakt treff (exact match). Det gir heller ingen mening å rangere resultatet med mindre det er eksplisitt angitt i spørringen. I IR-systemer derimot ønsker en i stedet for å finne dokumenter som omhandler et bestemt tema og en ønsker omtrentlige treff (approximate match). En ønsker nemlig at systemet skal forsøke å hjelpe brukeren til å finne de dokumentene han/hun ”egentlig” er ute etter. Dette gjøres f.eks. ved å utføre leksikalske analyser på dokumenter og spørringer og relevansrangere dokumenter. Det gir derfor stor mening i å rangere resultatet av et søk i et IR-system for å hjelpe brukeren med å finne de riktige dokumentene.

Jeg vil i dette kapittelet forklare en rekke egenskaper og begreper som angår informasjonsgjenfinning generelt. I siste del av kapittelet (3.6) presenteres FAST Data Search, som eksempel på et IR-system, og som den søkemotoren vi benyttet oss av i utvikling av prototypen. Mesteparten vil være hentet fra bøkene [27-29].

3.1 Indeksering

Med indeksering innen IR-systemer menes vanligvis det å assosiere ett eller flere nøkkelord med et dokument. Nøkkelord er her et leksikalsk atom, typisk et ord, deler av et ord eller fraser. En kan se på indeksering som en matematisk relasjon som avbilder den mengden nøkkelord $\{kw\}$ et dokument doc er om:

$$Indeks : doc_i \xrightarrow{om} \{kw_j\}$$

Den inverse relasjonen avbilder, for hvert nøkkelord, den mengden dokumenter det forekommer i:

$$Indeks^{-1} : kw_j \xrightarrow{beskriver} \{doc_i\}$$

Vokabularet av nøkkelord kan enten være kontrollert eller ukontrollert. Kontrollert vokabular ved indeksering betyr at indekseringen må foregå ved utvelgelse av nøkkelord fra et forhåndsdefinert sett. Indekseringen kan enten være manuell eller automatisk.

Manuell indeksering

Manuell indeksering betyr at en person knytter nøkkelord til dokumentet. Dette har sine klare fordeler da et menneske lettere kan skjønne hva et dokument faktisk handler om. Fremfor alt kan en (kompetent) person se relasjoner mellom dokumenter som omhandler det samme, men som ikke har leksikalsk likhet. Personer kan også lettere definere bredere/smalere og relaterte termer for å beskrive dokumentet. Den største ulempen er selvfølgelig arbeidsmengden. Å manuelt indeksert alle dokumentene som store søkemotorer indekserer i dag ville nesten vært en umulighet. I følge Search Engine Watch's septemberrapport fra 2003[30] har både Google, Inktomi[31] og AllTheWeb[32] indeksert over 3 milliarder dokumenter. På websiden sin opplyser nå Google at de indekserer over 4 milliarder (september 2004).

Et annet problem er at nøkkelordene en bruker for å beskrive innholdet i et dokument ikke er uavhengig av kontekst. Det er liten sannsynlighet for at forskjellige personer vil indeksere de samme dokumentene likt. Likevel brukes dette i flere systemer, ofte med hjelp av automatiske indekseringsmetoder, og gjerne i systemer som ikke skal være helt generelle, men har et mer spesifikt domene. Et typisk eksempel på kontrollert manuell indeksering er forfattere av vitenskaplige artikler som blir bedt om å beskrive artikkelen ved et begrenset antall nøkkelord fra en forhåndsdefinert liste. Videre i oppgaven kommer jeg bare til å forholde meg til automatisk indeksering.

Automatisk indeksering

Automatisk indeksering betyr at en bruker algoritmer for å trekke ut nøkkelord automatisk og forsøke å oppnå den samme effekten som manuell indeksering, slik at nøkkelordene beskriver hva dokumentet handler om. En forutsetter da at et dokument handler om de ord som forekommer i dokumentet. Selv om en tok med alle ord i et dokument, ville det ikke nødvendigvis gi noen merverdi i forhold til adekvate nøkkelord Det finnes derfor en rekke teknikker for å trekke ut de viktigste ordene og analysere dokumentet for å hjelpe brukeren. Fra et dokument kommer inn til et IR-system og til det er ferdig indeksert går det normalt gjennom flere faser, hvor hver av disse vil gjøre en analyse av dokumentet og enten utvider eller minske den informasjonen som finnes i dette før det til slutt blir indeksert. Jeg vil nedenfor beskrive en del av de teknikkene som blir brukt/stegene dokumentet går gjennom ved automatisk indeksering.

3.1.1 Dokumentvask

De fleste søketeknologier støtter filer på flere formater som PDF, HTML og Word. Eksempelvis støtter Oracle Text (se 4.2) over 150 og FAST Data Search (se 3.6) over 225 dokumentformater. For at data skal bli indeksert på en konsekvent måte må disse derfor konverteres til ett felles internt format før det blir behandlet videre. Den enkleste metoden er kun å "vaske" vekk all formatering og metainformasjon. Da mister en imidlertid mye informasjon om

hva dokumentet er om. Sannsynligvis vil et dokument som har Descartes i en overskrift være mer aktuelt for en som søker om informasjon om Descartes enn om det nevnes ett eller annet sted i teksten. Det vanligste er derfor at en bygger opp ett internt format f.eks. i XML med en generell formatering på overskrifter, tittel, forfatter samt annen metainformasjon en kan finne i dokumentet. En annen teknikk er også at den kun henter ut ”synlig” tekst. Eksempelvis var det mange HTML sider som forsøkte å lure søkemotorene ved å legge inn mengder med nøkkelord med liten skrift og i samme farge som bakgrunnsfargen slik at siden skulle bli mer relevant ved søk og dermed komme høyere opp i søkemotorer. Når en så har dokumentet på en konsistent intern formattering kan andre leksikalske analyser gjøres.

3.1.2 Automatisk språkdeteksjon

For at et IR-system skal kunne håndtere dokumenter på flere språk er automatisk språkdeteksjon viktig. De fleste prosessene som foregår før indeksering av dokumentet (beskrives nedenfor), må vite hvilket språk det er snakk om. Det ville virket mot sin hensikt om ’Lie’ som et egennavn på norsk skulle blitt utvidet til ’lies’ eller ’lay’ hvis det ble oppfattet som engelsk.

3.1.3 Ordoppdeling.

En viktig oppgave er å dele teksten inn i ”ord” – *ordoppdeling* (Tokenizing). Her er ”ord” i anførselstegn fordi det ikke nødvendigvis er gitt hvordan en skal dele opp teksten. For vesteuropeiske språk vil denne oppgaven stort sett være å dele opp teksten i ord ved å se på et ord som en sammensetning av alfanumeriske bokstaver separert med mellomrom (etter å ha fjernet spesialtegn og punktum). Men behandling av bindestreker er et problem. Skal ordene slås sammen? Deles opp i to ord? Et godt eksempel er databaser som i engelsk litteratur har blitt skrevet både som ”data base”, ”data-base” og ”database”. Det er selvsagt snakk om samme begrep, men kan dette oppdeles på en konsistent måte? Andre utfordringer er behandling av spesialtegn. Skal en indeksere ”it’s” som ”it’s” eller ”its”? Hvordan skal en behandle andre ”spesialord” som f.eks. filnavn. En vil jo ikke at ’windows.exe’ skal indekseres som to ord. Hvordan skal en behandle URL’er? Dette er avgjørelser som må tas på et tidlig tidspunkt og ofte kan det spesifiseres i søkemotoren hvordan en ønsker å behandle disse.

Symbolske språk, som japansk, skiller ikke mellom ”ord” ved hjelp av mellomrom på samme måte som en gjør i vestlige språk. Andre regler gjelder her for hvordan en skal dele opp teksten.

3.1.4 Stoppord

Fra tekstanalyser ser en at ganske få ord opptar en ganske betydelig del av vanlig tekst. Ord som ’en’ og ’og’ forekommer nesten i enhver tekst. Disse er derfor ikke særlig egnet som nøkkelord i og med at de forekommer i de fleste dokumenter, og de forteller ikke noe om hva dokumentet er om. Derfor ønsker en for å spare plass ofte å fjerne disse unødvendige ordene, kalt *stoppord* (Noise Word), fra dokumentet før det blir indeksert. En endelig mengde med stoppord blir samlet i en *stoppordliste* (Negative dictionary / stop list). Problemet med en slik liste er at for å kunne definere den må en se på ordfrekvensene i dokumentdomenet før de blir indeksert, noe en ikke har tilgang på. Hva som er

stoppord kan også variere fra domene til domene, men det vanligste er å bruke analyser fra andre domener og anta at fordelingen vil være den samme.

Det er altså vanlig å fjerne de ord som forekommer flest ganger i et dokument, men hva med de som bare forekommer en gang (singleton)? Kan en anta at ordet verken beskriver dokumentet, eller vil bli brukt som et nøkkelord ved søk?

Ikke alle IR-systemer benytter stoppordlister, og bruken av dem varierer også. FAST Data Search fjerner ikke stoppord og argumenterer med at indeksen ikke blir uhåndterlig mye større tatt i betraktning den verdiøkningen en kan få ved søk (”på ball” kan bety fest og en ønsker ikke dokumenter om ”sparke ball”). Andre systemer som Oracle Text fjerner stoppord, men markerer at det har vært et stoppord der slik at et søk på sammensetningen ”bare en lek” gir treff på teksten ”bare på lek” men ikke på ”bare lek”. Microsofts SQL server 7 tar til sammenligning ikke vare på det og vil derfor gi treff på begge. Google søker ikke til vanlig i stoppord og derfor vil et søk på *to be or not to be* ikke gi deg dokumenter om Shakespeare, men kun gi deg de dokumenter som inneholder *not*. Ved frasesøk (*”to be or not to be”*) benytter den seg av flere indekser og du får et mer forventet svar.

3.1.5 Lemmatisering/stemming og ordnormalisering

Om en søker på ”filosof” ønsker en også å få treff på dokumenter som inneholder ordet ”filosofer”. Dvs det er ønskelig at en skal søke over alle bøyninger av søkeordet. Tradisjonelt sett har dette blitt gjort ved hjelp av to teknikker, hhv. stemming og lemmatisering. Begge teknikkene vil føre til høyere erindring (se 3.5.1).

Stemming går ut på at en, før en indekserer dokumentet, gjør om ordet til sin rotform. Dvs fjerner alle bøyningsendinger i en rekursiv prosess. Disse blir så indeksert. Tilsvarende gjøres med søkeordene før søk. Det betyr at kun ord i sin rotform vil bli å finne i indeksen, som dermed kan bli betydelig mindre, En kompresjon på 10 til 50 prosent kan oppnås avhengig av nøkkelordvokabularet og hvor aggressiv algoritme en bruker, dvs hvor mange regler som brukes. Problemene er imidlertid mange, det kan lett hende at ordene mister sin betydning. Hvis en søker på ”basilika” og ønsker å få treff fra arkitekturhistorien, men så får en treff på dokumenter som handler om ”basilikum” vil nok brukeren være relativt misfornøyd.

En litt mer avansert metode er lemmatisering. Her bruker en metoder for å forstå ordet og så slå opp alle bøyningene. Dette medfører at alle former av ord med uregelmessig bøyning (f.eks. en bok flere bøker) vil bli indeksert. Normalt vil en ekspandere substantiv til entall og flertall, men det er også muligheter for å lemmatisere adjektiver og verb. I så fall vil et ord bare bli ekspandert innenfor en av disse klassene.

En bruker også andre teknikker for å normalisere ord slik at en ikke bare får treff på andre bøyninger av et ord, men også andre skrivemåter (eks lønsj/lunch). Søker en på ord med betonte bokstaver vil en at f.eks. et søk på ”matiné” vil gi treff både på ”matine” og ”matiné”. Visse språk som tysk har også forskjellige,

likeverdige, stavemåter for ord som for eksempel München og Muenchen. Det er da mulig å tillate alternativ stavemåte slik at uansett hvilken av de to stavemåtene en søker på, vil en få treff i dokumenter som inneholder minst en av dem. I tillegg har en ofte mulighet til oppsplitting av sammensatte ord før indeksering.

3.1.6 Tesaurus

Noen ganger kan ord dekke samme mening, men likevel være så forskjellige at det ikke er mulig å lage noen form for generisk algoritme for å bestemme at to ord har samme betydning (for eksempel veranda og terrasse). For å løse slike problemer benytter en seg av en tesaurus.

En tesaurus er som en ordliste ordnet som en graf der ord har relasjoner til andre ord. En god tesaurus bør kunne gi både synonymer og antonymer (ord med motsatt betydning) for ethvert ord, i tillegg til *bredere* og *smalere termer* (broader og narrower terms) samt andre sterkt relaterte ord.

Et problem oppstår imidlertid ved homografer, to ord med forskjellig betydning, men som skrives likt. Eksempelvis bønner som både er noe man ber og noe man spiser. Da forsøker en ved hjelp av syntaktiske og semantiske analyser å avgjøre hvilken betydning som menes.

Som oftest tar en utgangspunkt i en generell tesaurus når en skal lage et IR-system, men avhengig av domenet den skal brukes, må den ofte tilpasses. I bedriftstilpassede systemer kan det også tenkes at det legges inn produktspesifikke relasjoner (eksempelvis søker en på ”laptop” ønskes også treff på ”iBook”). Andre nyttige bruksområder er relasjoner mellom korrekt stavete ord og tilhørende vanlige feilstavelser (common misspellings) og mellom ord som uttales likt (soundex).

Ved indeksering kan tesaurus brukes for å sørge for at alle ord som settes inn er på en bestemt form, som et avansert alternativ til stemming, eller at en kan bygge opp egne indekser med andre former av ord som kan søkes i ved ønske når en skal søke slik FDS gjør.

3.1.7 Navngjenkjenning

Det er ikke ønskelig at alle ord skal gjennomgå stemming og lemmatisering. Egennavn og produktnavn er eksempler på ord en ikke ønsker at skal gå gjennom lemmatisering, stavekontroll osv. Dette vil normalt være en liste på samme måte som stoppordlisten må bli satt opp av en administrator. Andre indikasjoner på navn kan f.eks. være ord i bare store bokstaver.

3.1.8 Generere dokumentsammendrag/”teaser”

Når en gjør fulltekst søk er det gjerne basert på et begrenset antall nøkkelord. Og alle dokumentene som søkemotoren vil finne vil nesten aldri være aktuelle. Det er derfor viktig å ha en mulighet for å kunne lese et par av de ”viktigste” setningene i et dokument for å få inntrykk av hva det handler om. Dette ser en spesielt på internettsøkemotorer der gjerne de ti beste treffene vises med et lite sammendrag. En har to måter å trekke ut et sammendrag på.

Statisk sammendrag blir generert før dokumentet blir indeksert og vil være det samme så lenge dokumentet finnes i søkemotoren. En forsøker å trekke ut relevant informasjon om dokumentet som beskriver hva det er om.

Dynamisk sammendrag blir generert på bakgrunn av dokumentet og spørringen som ble gjort. Dette genereres altså en gang per treff og vil således se annerledes ut for samme dokument for to forskjellige spørringer som gav treff på dette. Sammendraget vil typisk trekke ut setninger, eller deler av setninger, som inneholder nøkkelordene som ble gitt i spørringen.

3.2 Datastrukturer

Hittil har jeg kun beskrevet begreper og teknikker som brukes før selve dokumentet blir indeksert, men hvilke datastrukturer ligger til grunn for å kunne tillate raskt oppslag i så store mengder data? Jeg skal her forklare hovedprinsippene i de vanligste.

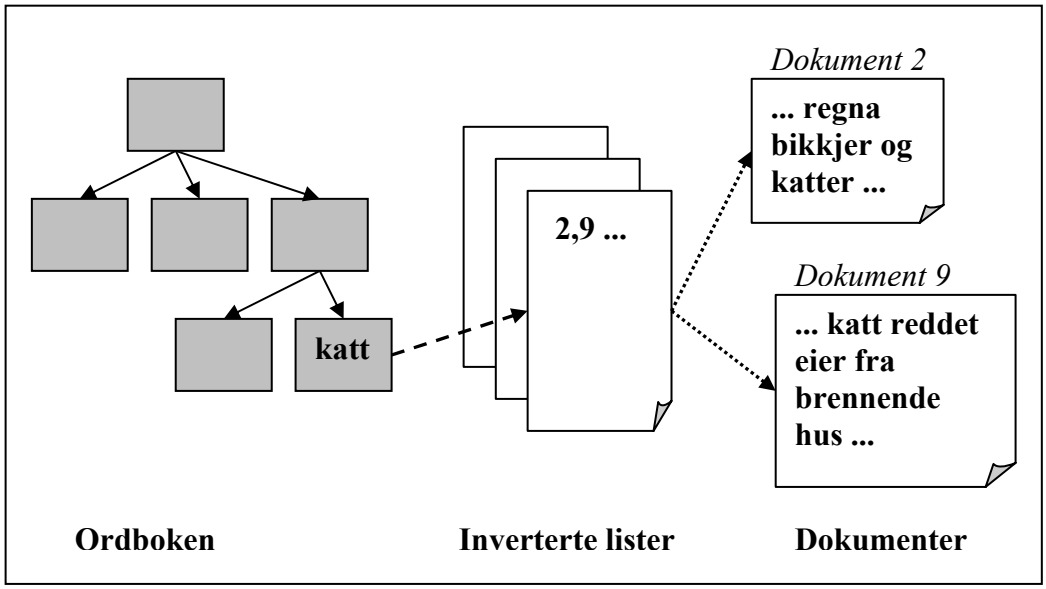
Inverterte filstrukturer, IFS

Som jeg beskrev i begynnelsen av avsnittet kan en se på indeksen som en funksjon som beskriver hva et dokument handler om, og den inverterte indeksen som en funksjon som viser, for hvert nøkkelord, hvilke dokumenter det angår. IFS er en datastruktur som tar utgangspunkt i dette. I stedet for å søke gjennom alle dokumenter for å se hvilke som inneholder et nøkkelord tar en i stedet utgangspunkt i nøkkelordene. Tilsvarende som når en leter i en god gammeldags lærebok etter et tema, så begynner en som oftest ikke å lete fra starten av, men slår opp i indeksen og finner hvilket kapittel som omhandler temaet.

Det finnes flere varianter av IFS, og kommersielle implementasjoner er ofte mer avansert, men i prinsippet består IFS av to komponenter:

- *Ordboken* (dictionary). Dette er en liste over alle unike nøkkelord som finnes i dokumentene, den har også pekere til den *inverterte listen* (Inversion List). Denne vil igjen være optimalisert for kjapt å slå opp et nøkkelord, ofte sortert eller som et B-tre, men også andre teknikker benyttes.
- Den inverterte listen, IL, har pekere fra nøkkelordet til alle dokumenter som inneholder dette. Den kan også ha mer avansert informasjon som ordposisjon i dokumentet slik at en kan svare på spørringer med fraser og spørringer angående nærhet av to nøkkelord, men dette går selvfølgelig på bekostning av regnekraft og plassbruk (dette kan også løses på andre måter).

Figur 1 nedenfor viser hvordan en IFS fungerer.



Figur 1: Oversikt over inverterte filer

Her er ordboken illustrert ved hjelp av et søketre og nøkkelordet katt peker på en invertert liste som igjen forteller at blant annet dokument 2 og 9 inneholder dette ordet.

I et moderne IR-system benyttes ofte flere IFS'er for å indeksere hver sine deler av dokumenter; en på tittel, en på overskrifter, en på forfatter, en på sammendraget osv. Dermed kan en lettere spesifisere søket og det går fortere samt at disse kan brukes ved rangering av dokumenter (finnes nøkkelordet i tittelen eller overskrifter er dokumentet trolig mer relevant enn om det finnes et annet sted i dokumentet). IFS er den mest brukte datastrukturen, spesielt for boolske søk IR-systemer, men de benytter seg også av tilleggsteknikker for å kunne tilby mer avansert funksjonalitet.

Signaturfiler

Signaturfiler går veldig enkelt fortalt ut på å konvertere ord/termer til binære strenger som blir en "signatur". En spørring blir også gjort om til en slik signatur og søket foregår ved at en sammenligner bit posisjoner i denne med de allerede forhåndsutregnede signaturfilene for dokumentene en skal søke i. Ved hjelp av denne kan en så eliminere alle som ikke treffer. De dokumenter som har signaturer som ikke har falt bort må så gjennomsøkes for å sjekke at de faktisk inneholder søkeordene. For å regne ut signaturfilene tar en blokker av nøkkelord i dokumentet og regner ut en hash over tegnverdien til hver av bokstavene som til slutt slås sammen (ved hjelp av bitvis OR) til den endelige blokksignaturen til dokumentet. Ett dokument kan således få flere signaturer.

Vektorrommodeller

Å lete opp bøker om et bestemt tema i et bibliotek går som en drøm. En bare spør betjeningen om hvor en kan finne bøker om et bestemt tema, og så har du hyllemeter på hyllemeter med bøker om akkurat det temaet du leter etter. Alle kan nåes bare ved å strekke ut hånden. Litt bortenfor finner du kanskje bøker som grenser til det temaet du ba om, og som kan vise seg å være vel så

interessant.. Dette går greit fordi bøkene er samlet i samme område i rommet. I vektorrommodeller forsøker en å lage semantiske rom der en prøver å samle dokumenter som omhandler det samme i nærheten av hverandre. Men i stedet for å begrense seg til et tredimensjonalt rom, snakker en i *vektorrommodeller* (Vector Space Models) om flere tusen dimensjoner.

For å få til dette tar en gjerne utgangspunkt i en *indeksmatrise* (Index matrix, term-by-document matrix) hvor en relaterer alle dokumentene til nøkkelordene ved å angi hvilke ord som forekommer i hvilke dokumenter. I sin enkleste form kan den se noe slikt ut:

Nøkkelord	Dokument 1	Dokument 2	Dokument 3	Dokument 4
Descartes	1	0	0	1
Aristoteles	0	1	0	1
Sokrates	1	0	1	1

Her ser en at nøkkelordet 'Descartes' forekommer både i dokument 1 og 4, men ikke i 2 og 3. Tilsvarende ser en at Dokument 4 inneholder alle tre nøkkelordene. Her kan en se på hver av nøkkelordene som en dimensjon i vektorrommet. Ethvert dokument kan da beskrives som en vektor i dette rommet. I dette minieksemplet kan en forholdsvis lett forestille seg disse dokumentene som vektorer i et tredimensjonalt rom, men problemene kommer når en ønsker alle mulige nøkkelord som dimensjoner, da snakker en lett om mer enn 100 000 ord. Hvis en så ganger opp det med 10^7 dokumenter, et forholdsvis beskjedent antall for et stort IR-system så blir matrisen $10^5 \times 10^7$ - hvilket blir mye å jobbe med (de største søkemotorene i dag behandler gjerne 300-400 ganger flere dokumenter).

Tilsvarene ser en også på spørringer som vektorer i dette rommet ved søk (en spørring er jo bare en sammensetning av utvalgte nøkkelord, akkurat som et dokument). I sin enkleste form, hvis en bare ønsker eksakt treff ord for ord vil det være å gå gjennom matrisen for å finne de aktuelle dokumentene, men til dette benyttes gjerne IFS. Ved bruk av få søkeord vil IFS trolig være best, men fordelen er at hvis en gir flere søkeord så kan en hente ut de dokumentene som ligger nærmest søkevektoren i dokumentrommet. Dermed trenger en heller ikke ha eksakt treff med alle søkeordene i spørringen.

Denne modellen brukes oftest som en videreføring der en sammenligner hvor nære ulike dokumenter ligger hverandre (vinkelen mellom vektorene) ved hjelp av en teknikk kalt *latent semantisk indeksering* (Latent semantic indexing, LSI). Det bygger på antagelsen om at de dokumenter som ligger nære hverandre også omhandler det samme eller om et beslektet tema. Dette blir igjen parallelt med eksemplet i biblioteket. En kan dermed finne dokumenter som er om det samme eller lignende tema som du søkte på, men ikke nødvendigvis hadde de eksakte nøkkelordene du brukte ved søk. Denne teknikken benyttes i moderne IR-systemer ved hjelp av "vis lignende dokumenter" funksjonalitet.

Jeg har her kun skissert hovedtrekkene og tankegangen bak en slik modell, men faktiske implementeringer benytter seg av flere teknikker for bl.a. å redusere

størrelsen på indeksmatrisene ved hjelp av diskret matematikk, men disse teknikkene vil jeg ikke gå nærmere inn på her.

3.3 Søk

Et søk består av flere prosesser, og er avhengig av flere forhold. For eksempel har vi nettopp sett at organisering av data i inverterte filer eller anvendelse av vektorrommodeller har betydning for søkemulighetene. Som første trinn i søket går spørringen gjennom flere steg på tilsvarende måte som ved indeksering. Dette kan innebære bl.a. oppsplitting av ord, stavekontroll, stemming osv. Måten dette gjøres på avhenger av hvilke typer spørringer en tillater (se nedenfor om spørreprosessering).

Siden det er store variasjoner i kunnskap om bruk av søkemotorer, er det vanskelig å si noe generelt om hvilke spørreteknikker som er ”best”. Noen brukere er avanserte og erfarne og ønsker seg rike søkemuligheter, selv om det måtte være komplisert. Andre brukere har liten eller ingen erfaring, og ønsker å ha det så enkelt som mulig. Nedenfor følger en beskrivelse av de vanligste spørreteknikkene

3.3.1 Spørreteknikker

Boolske spørringer

Ved boolske spørringer setter en sammen spørringer ved hjelp av de boolske operatorene AND, OR og NOT. Søker en på ’Descartes AND Artistoteles’ vil en kun få dokumenter som inneholder begge ordene, mens hadde en brukt OR ville en også fått dokumenter som kun inneholdt ett av dem. Det er litt verre med NOT, normalt sett betyr det en ekskludering eller invertering, men i søkesammengheng ville et søk på ’NOT Sokrates’ betydd alle dokumenter som ikke inneholder Sokrates, hvilket i de fleste tilfeller ikke gir noen mening. De fleste søkemotorer vil derfor ikke tillatte slike søk, men bruker operatoren AND NOT hvilket betyr en ekskludering, ikke fra alle dokumentene, men fra de som en allerede har søkt på (med andre ord mengdedifferanse; ofte brukes ’-’ som alternativ operator). Eksempelvis vil søket ’Descartes AND NOT Aristoteles’ gi deg alle dokumenter om Descartes hvor det ikke nevnes Aristoteles, dvs en vil finne alle dokumenter om Descartes og så ekskludere de som også nevner Aristoteles.

En av de store fordelene med boolske spørringer er at en ved hjelp av parenteser kan gruppere og bygge opp forholdsvis avanserte spørringer selv med så få begrensede operatorer. Men det er også problemer, jeg vil omtale tre eksempler fra [27]. For det første, i et rent boolsk spørrespråk er det ikke muligheter for å kunne vekte ord i forhold til hverandre. Enten forekommer et nøkkelord i dokumentet eller så gjør det ikke det. En kan f.eks. ikke søke etter ”musikk av Mozart, helst klaverkonserter”. Enkleste boolske søket ville være ”Mozart AND klaverkonsert”, men det ville ekskludert all annen musikk enn klaverkonsertene hans. En kunne forøke seg med å søke på ”(Mozart OR klaverkonsert) AND Mozart” og håpet rangeringsalgoritmen (se punkt 3.3.3 om rangering) til søkemotoren ville sørget for å plassere klaverkonsertene øverst, men det er ingen garanti for at den hadde oppført seg sånn. Videre viser erfaring at vanlige

brukere, i motsetning til informatikere og matematikere, ikke er helt fortrolige med boolske spørringer og at f.eks. AND og OR byttes om. Eksempelvis kan en bruker som ønsker å finne ut teater- og kinoaktivitetene i byen komme til å søke på ”kino AND teater”, når han egentlig mener ”kino OR teater”. Det tredje problemet består i at presedensen (dvs rekkefølgen de evalueres) til de logiske operatorene ikke nødvendigvis følger en bestemt standard. Det finnes nemlig to måter å evaluere et boolsk uttrykk på. Begge gjør det som finnes i parenteser først, men spørsmålet er hvordan en evaluerer det som finnes innenfor en parentes. Første metode er at en evaluerer NOT så AND og til slutt OR hvor operatorer av samme type blir evaluert fra venstre mot høyre. Den andre metoden er å evaluere alt fra venstre mot høyre. Eksempelet ”A OR B AND C” vil i første tilfelle bli evaluert til ”A OR (B AND C)” mens i det andre til ”(A OR B) AND C” hvilket sannsynligvis vil føre til to forskjellige resultater.

Uklare spørringer

Uklare (fuzzy) søk betegner søk i såkalt uklare mengder. I vanlig mengdelære er det slik at et element enten er med i en mengde eller ikke. I uklare mengder blir hvert element tillagt en *medlemskapsvekt* (membership grade), normalt mellom 0 og 1, som angir hvor sterkt et element er medlem av mengden. Eksempelvis kan en tenke seg mengden høye mennesker. I normal mengelære setter man en bestemt grense – la oss i vårt eksempel bruke 1.90m. I forhold til denne grensen er du enten høy eller så er du ikke høy; det hjelper lite om du er 1.8999m – du er fremdeles ikke høy. I uklare mengder kan derimot et element (i vårt tilfelle en person) få en medlemskaps vekt. En på 1.60m vil få en vekt nærmere 0, mens en på 1,89 vil være nærmere 1. Argumentet for å benytte seg av uklar informasjonsgjenfinning er at systemet, og ofte brukeren, ikke kan nøyaktig bestemme om et dokument vil kunne gi den informasjonen en er ute etter eller ikke. Denne usikkerheten blir modellert ved en ”uklar” evaluering av dokumentet i forhold til spørringen.

Dermed blir uklar informasjonsgjenfinning sterkt relatert til klassisk informasjonsgjenfinning der en beregner en form for relevansgrad og presenterer dokumentene til brukeren sortert på denne. Begrepet uklare søk brukes derfor også om teknikker som stemming og bruk av tesaurus for å utvide søket samt inkludere dokumenter som ”ligner” og bruk av rangering.

Siden dette uklare spørringer har vært mye omtalt og noen fast definisjon er vanskelig å finne. Kommersielle aktører sier ofte at de tilbyr dette selv om de ofte har en egen tolkning av hva som menes. Eksempelvis så benytter Oracle begrepet uklare søk om å utvide nøkkelord til å inkludere ord som staves lignende mens FAST Data Search benytter begrepet uklar logikk (fuzzy logic) om å kunne gjøre mer avansert boolske spørringer som det å uttrykke at tre av fem deluttrykk skal evaluere til sann.

Nærhetsspørringer

Det er ikke alltid det holder å bare kunne gi nøkkelord. Om en søker på to nøkkelord og får treff på en bok hvor disse forekommer med 200 siders mellomrom er det liten sannsynlighet for at det er den boken du er ute etter. En kan derfor ofte angi *nærhet* (proximity) som i SQL/MM der en bl.a. kan si

”Descartes IN SAME PARAGRAPH AS Aristoteles” og på enda mer spesifikt angi maksimalt antall ord som skal være mellom.

En annen mer utbredt nærhetsspørring er å benytte seg av fraser. Søker en på ”René Descartes” vil en kun få treff på de dokumenter der disse ordene kommer etter hverandre. Her vil det ofte variere litt fra system til system om hvordan en behandler stoppord som er en del av en frase (se 3.1.4).

Nærhet blir ofte også benyttet uten at brukeren eksplisitt oppgir det, men spørringer kan skrives om av søkemotoren for å tilby økt funksjonalitet og det brukes ofte som et mål for å evaluere relevansen til et dokument i forhold til et annet.

Spørringer i naturlig språk (NLQ)

Alle spørreteknikkene presentert til nå er basert på at brukeren må komme med et sett med nøkkelord på en eller annen måte. Poenget med spørringer i *naturlig språk* (Natural Language Queries (NLQ)) er at brukeren i stedet for skriver inn et spørsmål eller en ytring/befaling i form av en setning i det naturlige språket til brukeren. Eksempelvis ”Hvilken filosof skrev Cogito, ergo sum?” og ”Gi meg informasjon om impresjonistenes påvirkning på politikken.”. Slike setninger er lette for brukeren å formulere, men er ofte upresise og grammatisk ufullstendige. En bruker kan til og med være ute etter informasjon en ikke implisitt ber om, spør du ”Vet du hva klokken er?” forventer en noe annet en ja eller nei. Å oversette en slik spørring til noe søkemotoren kan forstå er imidlertid ikke trivielt. I sin enkleste form kan det være å fjerne utvalgte ord fra en stoppliste som ”Hvilken”, ”Gi meg” osv. for så å behandle spørringen som et sett med nøkkelord. Problemet med en slik tilnærming er at mye av informasjonen og strukturen i spørringen da vil bli tapt. De systemer som har hatt suksess har stor sett begrenset seg til forholdsvis smale domener og er ofte avhengig av brukerinteraksjon. Nytten av NLQ er derfor foreløpig begrenset. men også generelle søkemotorer som Stochasto[33] hevder å kunne tilby dette innen kort tid.

3.3.2 Spørreprosessering

Fra en bruker skriver inn en spørring må spørringen, som dokumenter ved indeksering, gjennom en prosessering. Dette først og fremst for å kunne tilby mer funksjonalitet. Noe av dette vil være usynlig for brukeren. Det er vanlig å velge en av tre taktikker når en skal omskrive spørringen:

- Automatisk skrive om spørringen før den sendes til søkemotoren.
- Skrive den alternative spørringen som et tips etter å ha søkt.
- En kombinasjon av de over – hvis ingen treff med originalspørringen, skriv om og prøv på nytt.

For å kunne behandle spørringen er søkemotoren avhengig av å vite språket. I de fleste tilfeller vil det være for lite tekst til at automatisk språkdeteksjon er være mulig. Språket må derfor enten være oppgitt i søkeapplikasjonen eller settes av brukeren ved søk.

Det første som skjer med en spørring er at den blir analysert for å finne fraser og egennavn. Dette vil som oftest skje ved at spørringen sjekkes opp mot en liste over spesielle ord som kan være viktige for dokumenter og spørringer, f.eks. produktnavn, domene terminologi, varemerker osv. Disse navnene vil som oftest ikke være å finne i en generell ordliste (f.eks. produktkode ”DWS-F1A”) eller de har en spesiell semantisk verdi for domenet (f.eks. merkenavn som ”FAST Data Search”). I alle fall blir navn/fraser som blir gjenkjent i dette steget ikke behandlet videre. I dette steget kan en også sette sammen to eller flere søkeord til en frase, spørresegmentering (query segmentation)[34], hvis en mener at dette kan være ønskelig. F.eks. kan søket *New York* gjøres om til ***”New York”***.

En god søkemotor bør også kunne tilby stavekontroll. Et generelt problem med stavekontroll er at de er langt fra perfekte, spesielt hvis det ikke er mulighet for å analysere ordet i en grammatisk betydning i en setning hvilket ikke er tilfellet ved enkle nøkkelord. Et annet problem er at det finnes en rekke ord som ikke naturlig vil være i en ordbok, som produktnavn. I en stor dokumentmengde vil det også være tilfeller der ord er feilstavet i dokumentene også. Så hvordan skal en gjøre dette? Hvor ”flink/fantasifull” skal den være? Det siste avgjøres som oftest ved at en ser på redigeringsavstanden mellom to ord, dvs. antall karakter operasjoner en må foreta (legg til, bytt plass, fjern bokstaver) for å gjøre om ordet til et annet. Som oftest settes det en grense på hvor stor redigeringsavstand som tillates. Ett ord kan med en slik metode gjøres om til flere andre så hvilke skal en velge?. Her ser en som oftest på ordforekomster, dvs. ord som forekommer ofte foretrekkes. Hvilken ordfrekvens en ser på kan variere. Ser en på ordforekomstene i dokumentene som er indeksert, kan en få et bedre resultat enn om en ser på ordforekomstene i språket generelt. I og med at det kan få uheldige konsekvenser hvis det gjøres automatisk er dette funksjonalitet som oftest blir tilbudt ved at en ikke gjør om søket, men tilbyr et alternativ søk som ”mente du ___ i stedet for?”.

Om en skal foreta stemming eller lemmatisering av ordene i spørringen er avhengig av hvordan en har indeksert dokumentene. Det er i hovedsak to strategier. Enten gjør en om ordene til sin rotform (stemming) før dokumentene blir indeksert. I så fall må det samme gjøres med søkeordene. Det andre er at ordene blir utvidet (lemmatisert) ved indeksering I så fall er alle bøyningsformene av aktuelle ord lagt inn ved indekseringen – den grammatikalske form som benyttes i søket vil derfor allerede finnes i dokumentet. (se for øvrig omtale av stemming og lemmatisering i forbindelse med indeksering, avsnitt 3.1.5).

Tesaurus kan også brukes når en skal utføre et søk ved at søket blir utvidet til å inkludere andre relaterte ord. I enkelte applikasjoner som Oracle Text, og som definert i SQL/MM kan brukeren i spørringen avgjøre om et søkeord skal utvides til å ta med relaterte ord, utvides eller eventuelt smales inn.

Til slutt kan det være ønskelig å splitte opp fraser for å øke erindringen til søket, gjerne ved hjelp av en stoppordliste.

3.3.3 Relevans og rangering

Det som virkelig skiller en god søkemotor fra en mindre god, sett fra en brukers synspunkt, er hvor relevante de dokumentene er som kommer ut av søket. Dagens søkemotorer er designet for å kunne søke over enorme mengder data, eksempelvis så indekserer de store websøkemotorene over 3 milliarder sider. Den vanlige bruker vil kanskje ta en titt på de første 10-20 dokumentene som den finner enda søkemotoren kanskje forteller om flere hundre tusen treff. Det kritiske blir da hvilken rekkefølge dokumentene returneres i.

Rangering går ut på at en gir de ulike dokumentene en "karakter" i forhold til hvor relevant et dokument er og returnerer de med høyest rangering/relevans først. Problemet her blir å avgjøre hvordan disse "karakterene" skal deles ut – hvor relevant er et dokument. Tradisjonell evaluering av søkemotorer har tatt utgangspunkt at det finnes en "allvitende" ekspert som kan vurdere om et dokument er relevant eller ikke (se 3.5 for evaluering av IR-systemer). En annen måte å vurdere dette er å se på det en kaller *konsensuell relevans* (consensual relevance) [28]. Da ser en i stedet på hvor relevant et sett brukere ser et dokument uavhengig av om en domeneekspert ville rangert det som relevant eller ikke. Denne teknikken ser en ofte brukt i f.eks. tekniske artikler der brukere som leser dem selv kan rangere dem etter hvor nyttig de finner dem.

I tradisjonelle IR-systemer er det likevel vanlig å snakke om to typer automatisk rangering, statisk og dynamisk. Den totale rangeringen som et dokument får blir et forhold mellom disse to.

Statisk/metadata rangering

Den statiske rangeringen av et dokument er hvor relevant et dokument er i forhold til de andre dokumentene i indeksen uavhengig av en spørring og teksten i dokumentet. Denne vil typisk bli beregnet samtidig som et dokument blir indeksert og utpregede egenskaper som har innvirkning er:

- Publiseringsdato, dvs hvor gammelt et dokument er. Generelt sett vil en si at nyere dokumenter er mer relevant en eldre dokumenter.
- Egenskaper satt ved en manuell innsetting av dokumentet. F.eks. hvis en benytter seg av et *dokumenthåndteringssystem* (content management system) kan brukere som registrere dokumenter sette visse egenskaper til et dokument.
- Egenspesifisert rangering. Enkelte kommersielle søkemotorer har f.eks. tillatt at selskaper som betaler for det kan få sine websider rangert høyere enn konkurrentenes.
- Dokumentrelasjoner. Google ble kjent for sin PageRank-algoritme og flere andre har andre lignende teknikker. Tankegangen går ut på at jo flere dokumenter som peker på et dokument (først og fremst HTML dokumenter ved hjelp av ankerreferanser) jo viktigere er dette. Ofte ser en på hvor mange som peker på de dokumentene som peker på det aktuelle dokumentet igjen for å evaluere sannheten av en slik antagelse. I en slik analyse tar en også vare på hvilke(t) ord som ble brukt for å beskrive lenken (ankertekst - teksten en klikker for å komme til det refererte

dokumentet) i og med at et slikt nøkkelord ofte vil være veldig beskrivende for hva dokumentet inneholder.[2]

Ofte vil det også være mulig å velge andre former for statisk rangering. F.eks. sorter på dokumentstørrelse, dato osv. Disse vil da overkjøre all annen rangering.

Dynamisk rangering

Dynamisk rangering går ut på at en evaluerer de dokumentene som gir treff mot en gitt spørring og analyserer hvor godt treff det var. Rangeringen beregnes altså på bakgrunn av teksten i dokumentet og sett opp mot nøkkelordene som ble bruk i søket. Hensyn som ofte tas er:

- Ordplassering i dokumentet. Hvor et nøkkelord forekommer vil avgjøre hvilken verdi det skal ha for beskrivelse av et dokument over et annet. Her vil en da benytte seg av dokumentets struktur og f.eks. bestemme at et dokument som har et nøkkelord i en overskrift eller tittelen på dokumentet skal få en høyere rangering enn et dokument som har nøkkelordet et annet sted i teksten.
- Ordverdien i dokumentet. Hvor godt beskriver et ord et dokument? Ordet ”og” som forekommer i så og si alle norske dokumenter kan ikke sies å beskrive så mye hva et dokument er om. Derfor vil en ofte beregne en verdi til nøkkelordet for å se hvor godt det beskriver dokumentet. En teknikk som ofte benyttes til det er invers dokument frekvens (se avsnittet nedenfor).

Invers dokument frekvens (IDF)

Ved *invers dokument frekvens* (Inverse Document Frequency (IDF)) regner en ut frekvensen (hyppigheten) av et ord i et dokument vektet med antall dokumenter som inneholder ordet. Finnes ordet i mange dokumenter vil det dermed få en lavere verdi og bli vektlagt mindre som et beskrivende ord i dokumentet.

Hvis en har *N* dokumenter og av dem inneholder d_k dokumenter et gitt ord *k* kan en regne ut den inverse ordfrekvensen for *k* som:

$$Log_2\left(\frac{N}{d_k}\right)+1 = Log_2 N - Log_2 d_k + 1$$

Verdien av et ord vil med andre ord minske når d_k vokser. En kan da regne ut invers dokument frekvens vekten av et ord *k* for et gitt dokument *i*, w_{ik} ved å multiplisere den inverse ordfrekvensen av ordet med den absolutte forekomsten av ordet i dokumentet f_{ik} slik: $w_{ik} = f_{ik}\left[\log_2 N - \log_2 d_k + 1\right]$. w_{ik} er nå et mål for ”viktigheten” av ord *k* i dokument *i*.

[2] Dette utnyttes ved såkalte *Google bombs*. Der websider, spesielt bloggere, avtaler å legger ut linker på sine sider med en avtalt beskrivelse. Eksempelvis hvis en søker på "miserable failure" så kommer George W. Bush’s side på det hvite hus opp som første treff.

3.4 Klassifisering og kategorisering av dokumenter

I stedet for å se alle dokumenter samlet i en og samme ”dokumenthaug”, er det ofte ønskelig å kategorisere dokumentene, dvs samle i grupper de dokumenter som handler om samme tema. Dette gjøres primært for å øke presisjonen, men brukes også for å øke gjenfinning i et IR-system. En fordel med å strukturere dokumentene i kategorier, som også gjerne har en trestruktur, er at det å grave seg nedover i kategoriene i seg selv kan benyttes som et alternativ til å søke (som å velge kategorier på Gule Sider). Andre nytteeffekter er at en kan begrense søket til kun å gjelde dokumenter i visse kategorier og på den måten øke presisjonen.

Prinsipielt finnes det to måter å kategorisere dokumenter på. Forhåndsdefinert (overvåket) og ikke-forhåndsdefinert (uovervåket). Overvåket kategorisering betyr at det på forhånd er satt opp ett sett med kategorier som dokumenter kan relateres til. Noen systemer godtar at et dokument kan være i flere kategorier, mens andre kun godtar en kategori. Videre kan en bestemme om dokumenter blir plassert manuelt eller automatisk inn i en kategori. Yahoo![35], Gule Sider og Startsiden[36] er eksempler på systemer som benytter seg av manuell klassifisering.

Uovervåket kategorisering betyr at systemet selv gjør en kategorisering av dokumenter basert på likhet mellom dokumentene. Eksempelvis kan en gruppere resultatet av et søk inn i flere kategorier slik at en lettere kan finne frem til de aktuelle. Vivisimo[37] er et eksempel på en søkemotor som benytter seg av andre søkemotorer for det faktiske søket, men som gjør en likhetsanalyse av resultatene og grupperer dokumentene deretter. Den vanligste metoden for slik automatisk kategorisering er vektorrommodellen (se 3.2) for å sammenligne hvor nære to dokumenter er hverandre i vektorrommet. Det er denne teknikken som benyttes ved såkalte ”vis lignende sider” funksjonalitet som tilbys i moderne søkemotorer i dag. Det benyttes også for å unngå å vise sider som er for like da den kan anta at dette vil være to kopier av det samme innholdet.

Moderne søkemotorer, bl.a. FDS (FAST Data Search), tilbyr i tillegg til overvåket kategorisering en mulighet for å benytte seg av begge teknikkene. Det vil si en mulighet for å gjøre overvåket kategorisering automatisk etter en viss opplæringsfase. En begynner med å manuelt plassere dokumenter inn i kategorier og så vil systemet etter hvert lære seg hva slags dokumenter som skal plasseres i de forskjellige kategoriene.

3.5 Vurdering og Evaluering av en søkemotor

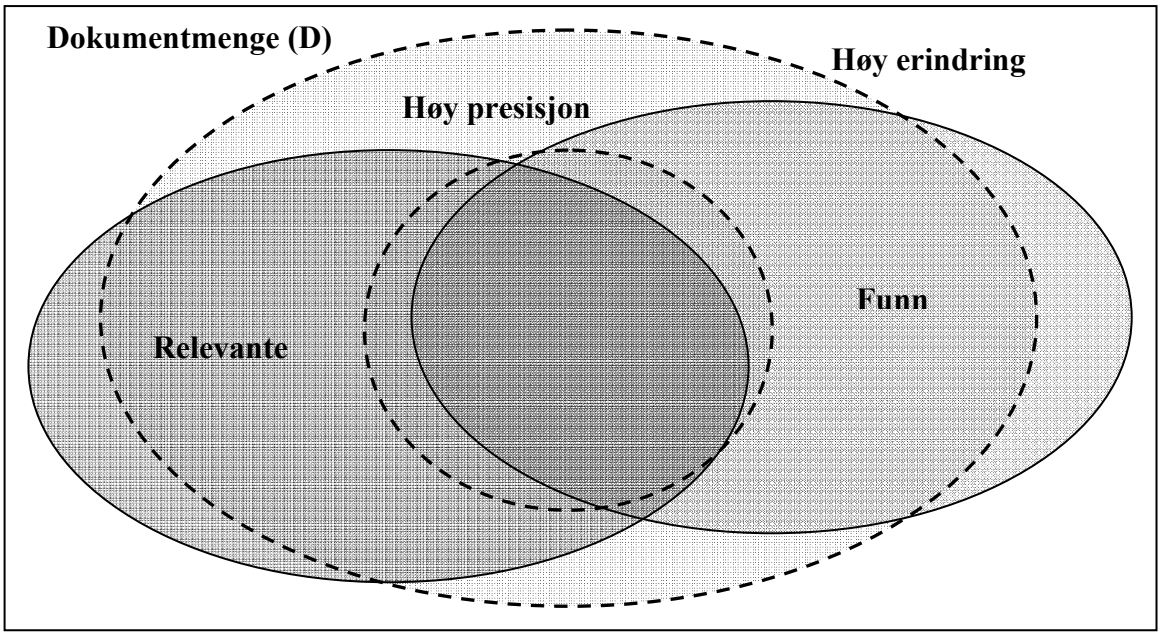
Når en vurderer et IR-system må en se på hvordan de forskjellige oppgavene er løst. Spørsmålene er mange: Indekseres stoppord, noe som øker størrelsen på indeksen, men gir mer konsist svar ved frasesøk? Registrerer indeksen ordposisjonen og benytter denne for å løse frasesøk eller bygges det opp en separat indeks bestående av fraser? En fordel på ett område, kan vise seg å bety en svakhet på et annet område. Ofte vil valget komme forskjellig ut for brukeren avhengig av hans/hennes bruksmønster. Noen brukere er avanserte og ønsker seg rike søkemuligheter, selv om det måtte være komplisert. Andre brukere har liten eller ingen erfaring, og ønsker å ha det så enkelt som mulig. Det er ikke nødvendigvis de samme egenskapene som kreves av en personlig søkemotor på

en laptop som av en internettsøkemotor. Derfor kan en heller ikke si generelt hvilken søkemotor som er "best". Når det gjeller søkeresultatet kan en likevel vurdere noen mer objektive kriterier.

3.5.1 Presisjon og erindring

Hva ønsker vi egentlig av en søkemotor? Vi gir den et ord eller to, og forventer at den skal skjønne hva vi mener, finne de få dokumentene som er relevante blant millioner, og komme tilbake med dem på tideler av et sekund. Dette er selvfølgelig en umulig oppgave, spesielt siden det er så å si umulig å si hva som er relevante dokumenter for akkurat deg eller meg, akkurat denne gangen. Søker du på "bra hotell i Lofoten" vil det variere hva som er relevante treff for deg. Får du opp et fem stjernes hotell er det vel og bra hvis du har lomma full av penger og føler behov for litt luksus, men kanskje et "bra hotell" for deg heller er et billig overnattingssted hvis du skal sykle utover øyene sammen med studentvenner. Med andre ord er "relevante" et subjektivt og ikke et presist begrep. Likevel benytter en seg av begrepet relevante dokumenter når en skal analysere et IR-system for å betegne de dokumenter i dokumentmengden, korpus, som med allvitende kunnskap faktisk er det.

Figur 2 nedenfor viser et venndiagram over forholdet mellom de dokumenter som faktisk ble gjenfunnet av IR-systemet (funn) og de relevante i en dokumentmengde (D). Her innføres to begreper som benyttes ved evaluering av et IR-system: presisjon og erindring. Vanligvis ønsker en både høy presisjon og erindring



Figur 2: Venndiagram over høy presisjon og høy erindring (gjengitt fra figur 1.10 i Finding out about [28])

Med høy presisjon mener en at så mange som mulig av de dokumentene som gjenfinnes skal være relevante, (uten at en tar stilling til hvorvidt en får med alle relevante dokumenter). Høy erindring betyr at de fleste relevante dokumenter

skal bli gjenfunnet (mens en ikke bryr seg om hvorvidt det medfører at også irrelevante dokumenter blir gjenfunnet) Mer konkret kan disse begrepene defineres som:

$$erindring \equiv \frac{|\, funn \cap relevante \,|}{|\, relevante \,|} \text{ og } presisjon \equiv \frac{|\, funn \cap relevante \,|}{|\, funn \,|}$$

Der *funn* er den mengden dokumenter som vil bli returnert ved et søk og *relevante* er den mengden som er "absolutt relevante". Hvis en ønsker å finne så mange som mulig av relevante dokumenter, vil en altså ha så høy erindring som mulig. Problemet med dette er at søket da gjerne returnerer også mange irrelevante dokumenter. Ønsker en derimot høy presisjon , er det viktigst at de dokumentene en får er relevante, mens ulempen gjerne er at en mister en del relevante dokumenter Eksempelvis kan en tenke seg at en patentsøker vil være helt sikker på at ingen har tatt patent på det han/hun forsker på, før en er villig til å gå videre. I dette tilfellet vil nok forskeren ta seg tid til å også gå gjennom urelevante patenter for å være sikker. Men er den samme forskeren ute etter å lese litt i full fart om Descartes for å avklare når han levde, er det viktig at de dokumentene som kommer opp virkelig handler om Descartes, men det er underordnet om han går glipp av noen relevante dokumenter.

3.5.2 Skalerbarhet/hastighet

Et annet viktig kriterium på en god søkemotor vil være hurtig respons. Google hadde ikke vært hva det er i dag, hvis det tok 10 minutter å gjøre et søk, selv om det på mange måter revolusjonerte kommersielle søkemotorer. IR-systemer er noen av de mest kompliserte systemene vi har i dag og det ikke vanskelig å skjønne at det kreves mye maskinkraft for å utføre jobben, selv med gode datastrukturer i bunnen. Et godt IR-system må derfor ha mulighet for å skalere opp både i form av data som skal indekseres, men også i forhold til antall brukere og søk per minutt. Eksempelvis har Google titalls tusener av maskiner for å støtte webcrawling og søk.

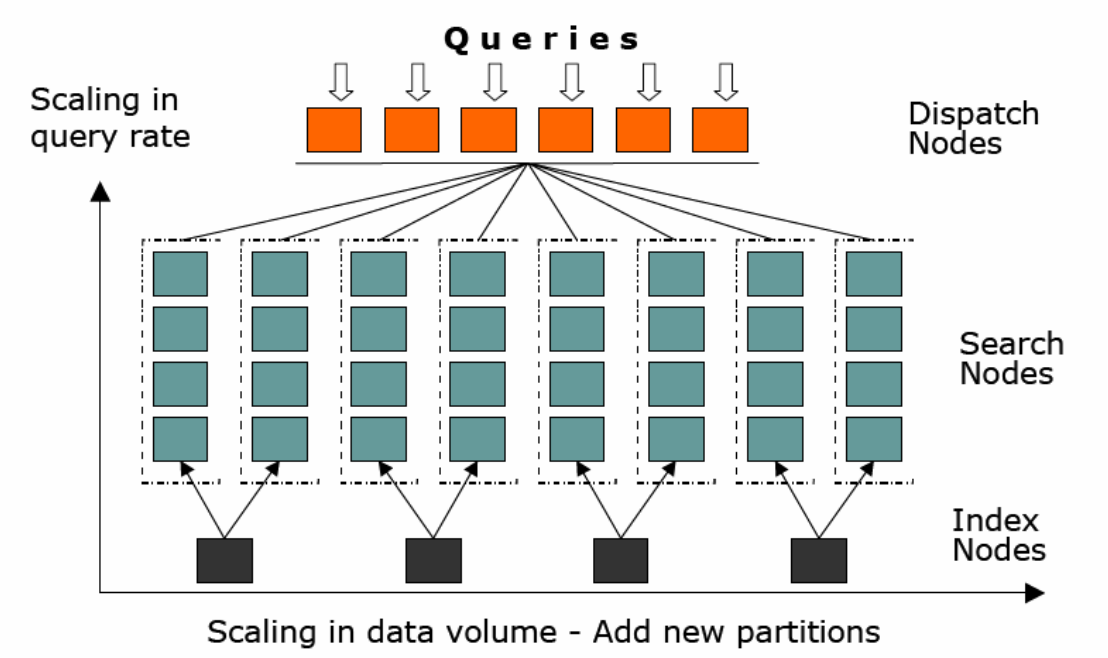
Oppgavene til et IR-system kan forholdsvis enkelt parallelliseres da deloppgavene er forholdsvis distinkte (en prosess kan hente ut en teaser mens en annen gjør språkdeteksjon og en tredje crawler osv). De enkelte deloppgavene egner seg også godt for parallellisering på en enkelt node da noen er svært diskkrevende, mens andre er svært CPU intensive. For ytterligere å øke skalerbarheten kan disse igjen dupliseres slik at noen søk foregår på node a, mens andre på node b og skulle det bli nødvendig å tillate enda flere søk samtidig kan en legge til node c, altså kan en "ubegrenset" øke søkeskalerbarheten ved å øke antall maskiner.

Generelt kan en si at jo mer funksjonalitet som tilbys, jo større plass vil en indeks ta og jo mer maskinressurser kreves. Skal en tilby mulighet for å se en såkalt bufferversjon (cached version) av dokumentet må dette også lagres i systemet. I IR-systemer som behandler store menger data er det ikke uvanlig at indeksen kan komme opp i flere terabytes. Da kan det være ønskelig, om ikke nødvendig, å spre dokumentene og indeksen over flere noder. Eksempelvis kan den inverterte

indeksen sprees slik at A-G befinner seg på node a, H-K på b osv. Skulle datamengden bli større kunne en derfor bare legge til flere noder.

I det en begynner å distribuere oppgavene utover på denne måten, kommer en riktignok opp i en rekke problemstillinger. For det første må en ha egne noder for å samkjøre disse, eksempelvis sette sammen resultatet fra søk i flere noder. Andre problemer kan være at de forskjellige nodene ikke nødvendigvis har samme versjon av indeksen osv.

Figur 3 nedenfor viser hvordan FDS benytter seg av en slik "todimensjonal" skalering der en kan legge til noder horisontalt for å øke dokumentmengden som den skal indeksere og vertikalt for å øke søkeraten den skal takle.



Figur 3: Søkemotorskalerbarhet langs to akser (gjengitt fra figur 15 i Under-the-hood of FAST Enterprise Search Solutions[13]).

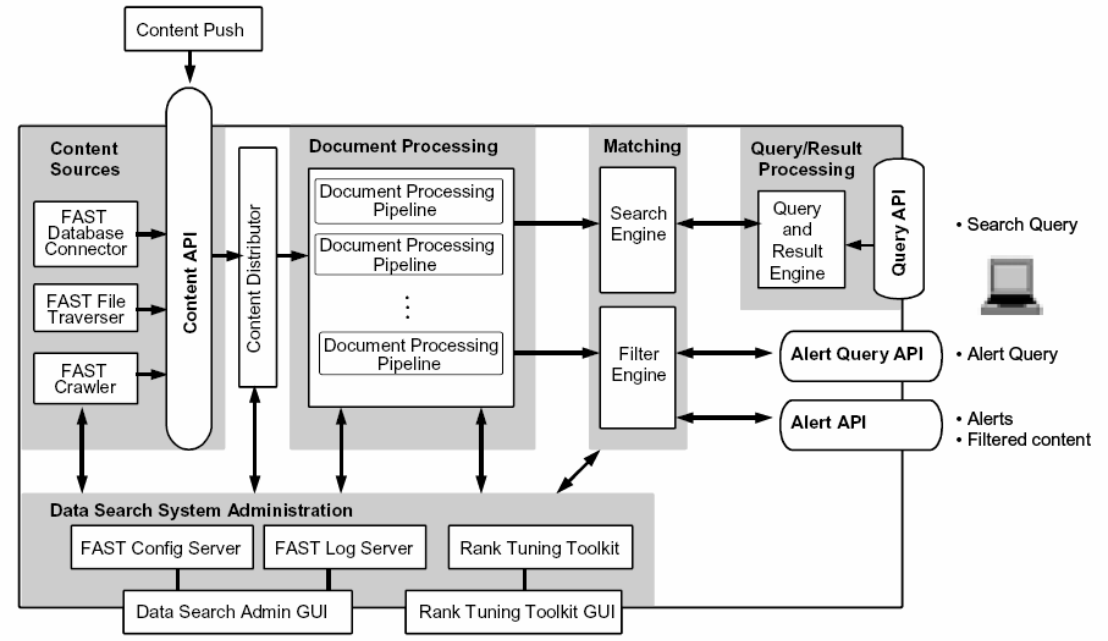
3.6 FDS (FAST Data Search)

FDS er et kraftig, meget skalerbart og konfigurerbart IR-system utviklet av Fast Search & Transfer (FAST). FDS kan brukes til å håndtere flere terabyte med tekstdata fra over 225 forskjellige filformater på 49 språk På grunn av sin skaleringsmulighet kan FDS, selv med slike datamengder, ha søketider på under sekundet selv med veldig mange simultane søk. Søkemotoren benyttes i dag bl.a. bak nettsteder som AllTheWeb og Lycos, men hovedfokus for produktet er virksomhetssøk. Det vil si store bedrifter som trenger å gjøre oppsamlet informasjon fra flere kilder lett tilgjengelig både internt og eksternt. IBM og Dell er blant selskapene som benytter seg av denne teknologien. På grunn av FDS er FAST Search & Transfer et av Europas raskest økende teknologiselskaper og regnes som visjonærere innen sitt domene av senere industrianalyser (Gartner).

Jeg vil her gi en oversikt over hovedtrekkene til FDS og samtidig bruke den som et eksempel på et IR-system (søkemotor) og se litt på hvilke egenskaper den har i forhold til de generelle som jeg har beskrevet ovenfor. FDS er det systemet vi benyttet oss av i vår prototyp.

Arkitektur

FDS består av et sett av moduler, der mange er basiskomponenter som ville inngått i ethvert IR-system, mens andre er mer spesifikke for FDS. Figur 4 nedenfor viser en oversikt over modul arkitekturen i FDS. Den består av fire hovedmoduler med ansvar for: Innholdskilder/innholdsinnhenting, dokumentprosessering, spørring og resultatprosessering og endelig dokumentsammenligning. Nedenfor vil jeg gi en nærmere omtale av hver av disse modulene. (I tillegg er det en modul for administrasjon, men denne vil ikke bli omtalt nærmere.)



Figur 4: Oversikt over arkitekturen i FDS

Innholdskilder og Innholdsinnhenting

FDS kan hente data både fra filområde, web og databaser, og tilbyr en mulighet for å strukturere dokumenter inn i en eller flere samlinger, collections. Fordelen med en slik måte å organisere dokumentene på er at hver av disse samlingene kan ha forskjellig struktur på dokumentene sine og en kan velge å legge dokumenter fra forskjellige kilder inn i forskjellige samlinger. På denne måten kan en også begrense et søk til å kun gjelde begrensede kilder (eks. web) eller begrense samlinger basert på adgangskontroll (eks skal kun ansatte kunne søke i dokumenter fra den interne filserveren)
FDS kan hente data/dokumenter enten ved hjelp av egendefinerte applikasjoner som via det eksponerte programmeringsgrensesnitt kan legge informasjon inn (push), eller via innhentingsmoduler som kan hente data fra web, filområde, eller databaser (pull).

Innhenting fra web - Internettkryping

FAST Web Crawler innhenter informasjon ved å manøvrere/krype seg gjennom nettsider, både intra- og internett, ved hjelp av å følge lenker på websider fra en oppgitt startadresse. Den kan begrenses til kun å finne websider til et sett av angitte domener og kun indeksere disse. Den vil hente data fra flere dokumentformater (FDS støtter flere enn 225 forskjellige) og konvertere disse til et felles XML format. Den vil også beholde relevant struktur- og metainformasjon (eksempelvis title, body og metainformasjon fra HTML).

Innhenting fra filer - Filtraversering

FAST File Traverser vil også lete etter dokumenter, men i stedet for å lete på internett vil denne gå gjennom gitte filområder for å innhente data. Som ved innhenting fra web kan den håndtere flere dokumentformater.

Innhenting fra databaser

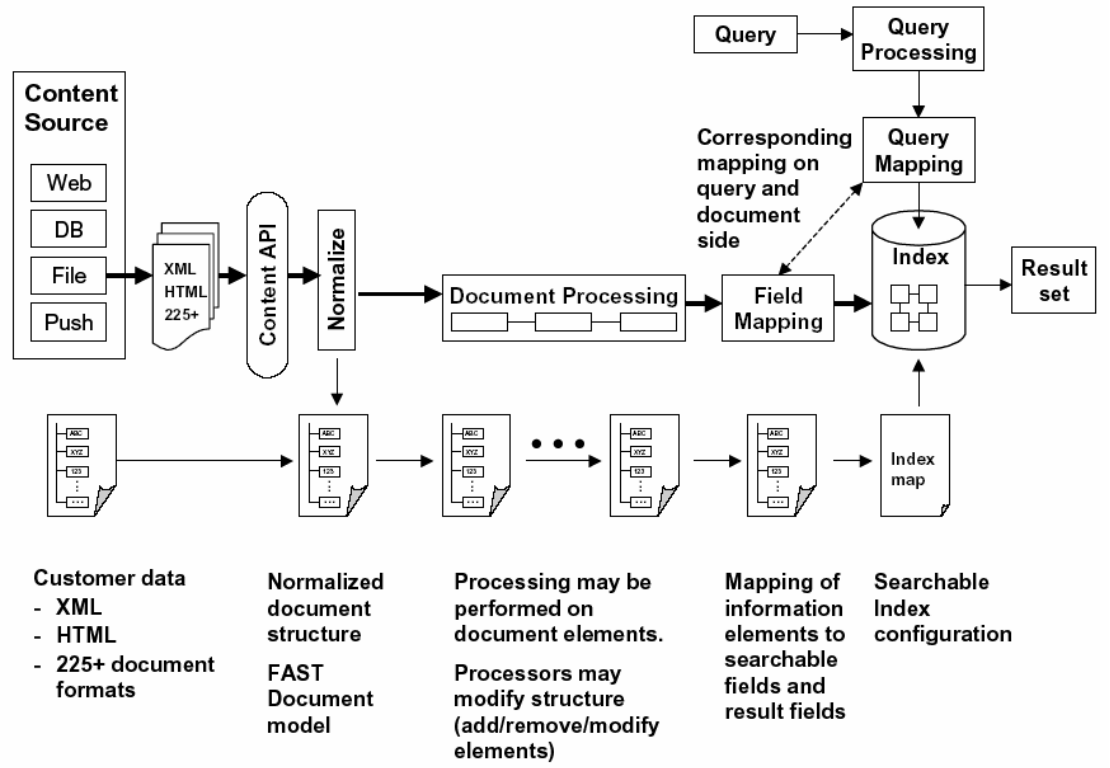
Av flere grunner er det ønskelig å hente ut data fra databaser i tillegg til data fra internett og filområde. Det kan avlaste databasetjenerne, det kan bety en annen måte å søke i data på (eksempelvis er det ikke sikkert at databasen tilbyr fulltekstsøk), og det kan tilby en mer integrert søkeplattform der en søker i tre kilder på en gang. FDS tilbyr tre måter å integrere en database på. Enten kan en programmere et eget program som vil aktivt legge informasjon inn, eller en kan bruke eksportmoduler i databasen for å generere XML filer som så kan behandles ved filtraversering. Den tredje muligheten er å benytte seg av *FAST Database Connector*. Ved at en setter opp en konfigurasjonsfil har FDS dermed mulighet for å hente ut data fra de største kommersielle databaseleverandørene som Oracle og DB2.

Programmeringsgrensesnitt

FAST Content API (Application Programming Interface) er et grensesnitt som gjør at en, hvis ingen av modulene over er tilstrekkelig, kan gjøre en egen integrering mot FDS via C++, COM eller Java. *FAST Content API* er først og fremst aktuelt der en ønsker en raskere oppdatering i FDS, fordi en da kan oppdatere FDS så fort det har skjedd en endring i datagrunnlaget. Med de andre teknikkene beskrevet over, vil systemet kun oppdatere seg ved gitte tidsintervaller

Dokumentprosessering

Fra et dokument blir innhentet i FDS til det er søkbart går det gjennom en rekke skritt. Figur 5 nedenfor viser hovedtrekkene i en slik prosess, der de fleste punktene er beskrevet i kapittel 3.



Figur 5: Dokumentflyt gjennom FDS

De individuelle stegene konfigureres og settes sammen (en kan også legge til egenprogrammerte steg) i en ”pipeline”. En eller flere dokumentsamlinger knyttes så til en slik pipeline i *FAST Document Processing Framework*.

De første stegene i en slik pipeline vil som oftest være dokumentformatgjenkjenning slik at riktig dokumentkonvertering kan utføres. I FDS er det mulig å ha en rekke atributter slik at ikke bare teksten i dokumenter indekseres flatt, men en lager også egne for titler, sammendrag osv. Dokumenter må så konverteres til et normalisert XML dokument med de spesifiserte feltene og ”vaskes”. Videre utføres andre steg for å kunne tilby IR funksjonalitet som blant annet språkdeteksjon, ordoppdeling, egennavngjenkjenning, generering av sammendrag, lemmatisering (se 3.1 for disse teknikkene) og klassifisering (se 3.4). Dokumentene vil så til slutt bli sendt til søke- og filtermodulen for indeksering og sammenligning.

Spørring- og resultatprosessering

Som beskrevet i 3.3.2 må også spørringer analyseres og prosesseres før de sendes til søkemotoren. FDS tilbyr støtte for fjerne fraser (antifrasing), sette søkeutrykk i fraser og stavekontroll av søkeordene ved søk. Det kan spesifiseres om dette skal gjøres automatisk, om det skal foreslåes som et alternativ eller en kombinasjon eksempelvis at det manuelt, men utføres automatisk hvis ingen dokumenter ble funnet.

Søkemotoren er meget skalerbar både med hensyn på datamengde og last, men blir ikke nærmere omtalt her. Se forøvrig 3.5.2 for hvordan FDS skalerer opp i to dimensjoner.

Resultatprosessering er det siste steget som utføres før resultatet vises til en bruker. FDS tilbyr spesifisering av maler slik at en kan spesifisere hvordan resultatet skal presenteres (typisk HTML eller XML). Andre teknikker som også benyttes er dynamiske sammendrag og fremheving av søkeordene i resultatsettet. FDS kan også integreres direkte inn i kundeapplikasjoner ved hjelp av programmeringsgrensesnittet.

Kontinuerlig filtrering av dokumenter

I tillegg til søk i indekserte data, ”vanlig” søk, tilbyr også FDS en filtermotor (*FAST Filter Engine*) som muliggjør kontinuerlig søk i innkommende data. Den evaluerer kontinuerlig alle dokumenter som kommer inn i systemet mot et sett av forhåndsdefinerte filtre/spørringer. Dette settes så sammen med *FAST Alert API* noe som tilbyr andre prosesser å ”abonnere” på slike hendelser. Eksempler på applikasjoner som benytter seg av denne teknologien kan være overvåking av aksjekurser, værmelding eller nyheter.

KAPITTEL 4: EKSISTERENDE TILNÆRMINGER

Jeg vil i dette kapittelet omtale eksisterende tilnærminger på integrasjon av et databasesystem og et IR-system. Først (4.1) vil jeg beskrive SQL/MM som er tilnærming via en standard. I 4.2 og 4.3 vil jeg se på mulighetene for fulltekstsøk i Oracle og MSSQL som eksempel på kommersielle tilnærminger. I 4.4 vil jeg se på utvidelser gjort i PostgreSQL. Til slutt vil jeg i 4.5-4.6 omtale utvalgte artikler fra forskningsfeltet.

4.1 SQL/MM

De forskjellige RDBMS leverandørene så fort et behov hos kundene for å tilby mer avansert funksjonalitet enn det som var definert i SQL-standarden. For eksempel [38] har de fleste implementert muligheten for lagring av boolske data, mange før det ble definert i SQL-99 (derimot er det mange som ikke har implementert MATCH operatoren definert i SQL-92, eksempel WHERE (x,y) MATCH (SELECT a,b FROM c)). Etter hvert som flere utvidelser ble innført i databasene, kom en opp i navneproblemer. For eksempel kom det i 1992 et tillegg til SQL kalt *SFQL* (Structured Full-text Query Language). Det definerte bl.a. CONTAINS som en operator for å betegne om et ord eller frase fantes i en tekst. Den samme operatoren ble imidlertid også brukt i databaser med geometri, og der for å betegne hvorvidt flater/volumer dekket hverandre. SFQL ble derfor kritisere for å "kapre" disse nøkkelordene [39].

I SQL/MM (SQL Multimedia and Application Packages), som er en ISO/IEC (International Organization for Standardization / International Engineering Consortium) standard og ment som et tillegg til SQL-99, omgår en slike navneproblemer. En av nyhetene i SQL-99 var mulighet for definisjon av abstrakte datatyper, ADT'er. Disse kunne en så definere med attributter og metoder. Dermed ble SQL-99 et mer objektinspirert språk der komplekse operatorer som CONTAINS ble sett som en metode til en datatype. Dermed var det ikke lenger noe problem å forså hvilke operasjoner det var snakk om, og flere datatyper kunne implementere metoder med samme navn, men med forskjellig betydning.

SQL/MM er foreløpig i fem deler, hver del innenfor multimedia domener og kan sees på som abstrakte klassebibliotek av ADT'er [9]:

- Del 1: Framework. Dette består hovedsakelig av fellesdefinisjoner for de andre delene.
- Del 2: Full-Text (beskrives nedenfor)
- Del 3: Spatial. Metoder for å behandle 2-d objekter, med objekter som punkter, kurver og polygoner og muligheter for avstands tester, inneholder osv.
- Del 5: Still Image. For bruk i billeddatabaser blant annet for å sammenligne hvor like to bilder er hverandre med hensyn til forskjellige egenskaper.
- Del 6: Data Mining. Standard for dataanalyse-operasjoner.

Opprinnelige var det også en del fire om generelle matematiske typer og operasjoner, men denne ble tidlig skrinlagt.
I del 2 av standarden er det definert en rekke datatyper for å lagre og manipulere tekstdata. Den datatypen som er mest aktuell i denne sammenhengen er *Fulltext* for å lagre ustrukturert tekstdata og *FT_Pattern* for å holde på fulltekst spørringer - *mønstre* (pattern) [10].

4.1.1 Fulltext

Generelt sett er FullText en datatype for å inneholde tekst samt et sett av metoder for å evaluere teksten opp mot ett mønster/søkeuttrykk. Datatypen inneholder to attributter; innhold og språk. Som forklart i kapittel tre om informasjonsgjenfinning er språk ett nødvendig parameter både på søkeuttrykket og teksten for å kunne utføre avansert tekstanalyse. I Fulltext er språk et parameter lagret om hver tekst og kan enten settes implisitt, eller ved en automatisk deteksjon og bruk av en standardverdi (default språk). Til denne datatypen er det så definert en rekke metoder og jeg vil nedenfor beskrive de to viktigste.

fullText.Contains(FT_Pattern search_expression)

Denne metoden evaluerer om teksten fullText tilfredstiller søkemønsteret definert i FT_Pattern (beskrevet nedenfor). I så fall vil den returnere 1.

fullText.Score(FT_Pattern search_expression)

Denne metoden returnerer et tall (innen et implementasjonsspesifikt spekter) som indikerer hvor godt søkemønsteret definert i FT_Pattern evaluerer i teksten fullText. Dette for å kunne gjøre en rangering av treff.

4.1.2 FT_Pattern

FT_Pattern er en datatype for å uttrykke fulltekstsøk/mønstre. Denne datatypen vil kunne evaluere gyldigheten av et søkeuttrykk. I FT_Pattern er det et rikt spørrespråk og BNF definisjonen fra standarden er i seg selv over tre og en halv side. Nedenfor følger en oversikt over de viktigste mulighetene en har i dette språket.

Bruk av ord og fraser

Det er støtte for bruk både av enkeltord og av flere ord satt sammen til fraser. Både ord og fraser kan brukes om hverandre i de fleste operasjoner beskrevet nedenfor. Standarden sier imidlertid ikke noe om hvordan stoppord skal behandles i fraser. Det er støtte for bruk av jokertegn (’_’ og ’%’) i både ord og fraser.

Boolske operatorer

Flere delsøkekriterier kan bindes sammen med de boolske operatorene & (AND), | (OR) og NOT.

Leksikalske operatorer

En rekke IR spesifikk leksikalske operatorer er definert, slik som stemming og lemmatisering (se 3.1.5). Også andre mer avanserte analyser kan gjøres for

eksempel ord som høres ut som (”SOUNDS LIKE *ord*”), eller skrives som, (”FUZZY FORM OF *ord”*) eller utvidet søk ved hjelp av tesaurus.

Kontekst søk

I søkemønsteret kan en angi hvor stor avstand som kan tillates mellom ordenen i søket. innen en gitt kontekst. Mulige kontekster er bokstaver, ord, setninger og avsnitt. En kan da velge om de skal være i samme kontekst (*ord1* IN SAME SENTENCE AS *ord2*) eller spesifisere hvor nært innenfor en kontekst (*ord1* NEAR *ord2* WITHIN 10 WORDS).

Konseptuelle søk

Det er definert operatorer for å utføre konseptuelle søk; IS ABOUT. Det spesifiseres imidlertid ikke hvordan et slikt uttrykk skal evalueres.

4.1.3 Eksempel

Nedenfor følger et avansert eksempel for å vise noe av styrken i SQL/MM fritekstsøk.

```
SELECT * FROM documents
WHERE body.Contains(
      'THESAURUS "computer science" EXPAND SYNONYM TERM OF "pc" &
      (STEMMED FORM OF "Standards" NEAR
      "language" WITHIN 0 SENTENCES IN ORDER |
      SOUNDS LIKE "sequel"') = 1
ORDER BY abstract.Score('"Master thesis"');
```

Denne, kanskje mindre nyttige spørringen vil returnere de dokumenter som: Inneholder synonymer av ”pc” basert på datatesaurusen for det språk som finnes i dokumentet. Eksempelvis ”laptop”. Videre dokumenter som har en ”stemmet” versjon av ”Standards”, eksempelvis ”Standard” før, men i samme setning som ”language” eller ord som høres ut som ”sequel” eksempelvis ”SQL”. Disse dokumentene vil så bli returnert ordnet etter relevans om ”master thesis”.

4.1.4 SQL/MM i kommersielle applikasjoner.

SQL/MM er per i dag kun en papirstandard og det kan diskuteres om det noen sinne vil bli noe mer enn det. Foreløpig er det ingen RDBMS leverandører som hevder kompatibilitet med noen deler av standarden og det er få som hevder at de har planer om dette.

Oracle hevder imidlertid at de planlegger å implementere del tre av standarden (rom-delen/2d), men nevner ingen ting om del 2 (fulltekst). Dette er egentlig merkelig i og med at kommersielle aktører, deriblant Oracle er sterkt representert i komiteen som spesifiserer SQL/MM. Som beskrevet nedenfor (i 4.2 om Oracle Text) har de imidlertid allerede implementert hovedfunksjonaliteten, med en egen (dog lignende) syntaks. Det kan være forretningsmessige grunner til dette f.eks. at en vil minske muligheten for eksisterende kunder til å gå over til en annen leverandør. Men det kan også skyldes implementasjonshensyn.

Jeg vil imidlertid hevde, gitt den siste tids fokus på større mengder tekst i databaser at flere og flere leverandører vil gjøre en gradvis tilnærming for å

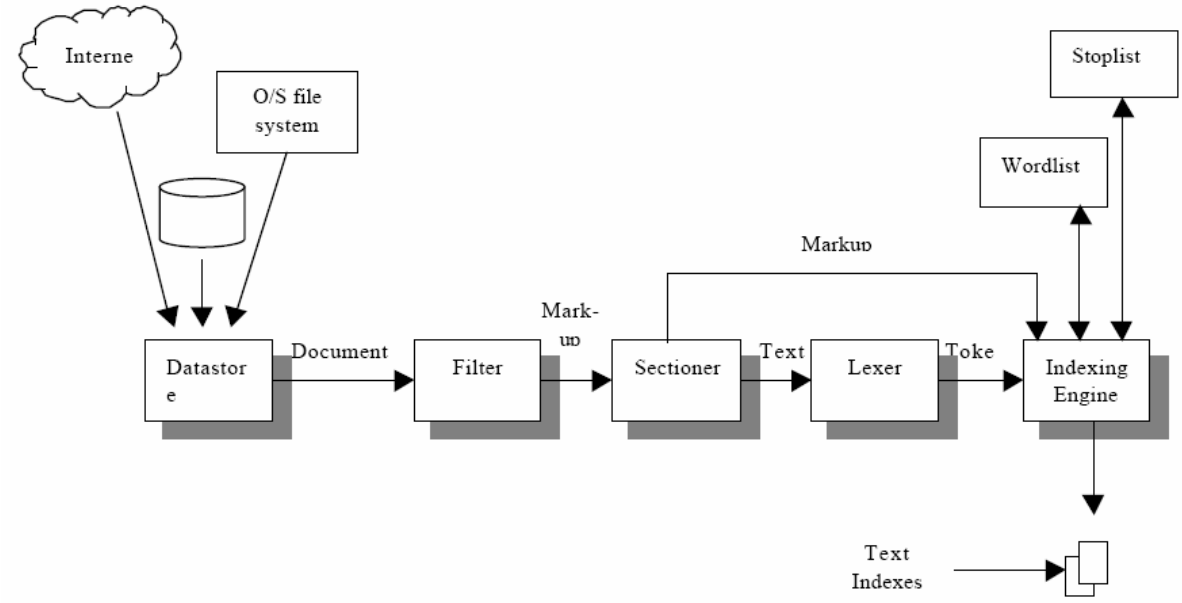
tilfredstille SQL-99 og deler av SQL/MM over tid. Databaseleverandørene har jo en tendens til å ligge en stund etter standarden og de fleste støtter jo i dag SQL-92 med utvalgte deler fra SQL-99.

4.2 Oracle Text

Oracle Text [11;40], også tidligere kjent som interMedia text, er ment å tilby en komplett tekstsøk funksjonalitet for brukere av Oracle databaser. Det er inkludert fra og med Oracle 9i i både standard og enterprise utgavene.

4.2.1 Arkitektur

Oracle Text er organisert som de fleste andre søkemotorer. Figuren nedenfor viser en oversikt over de forskjellige stegene. Jeg skal her gå gjennom hver av dem og forklare hva som er spesielt med disse.



Figur 6: Oversikt over Oracle Tekst arkitektur. Fra [40]

Datalager (Datastore)

De dokumentene/teksten som skal indekseres kan befinne seg enten:

- I databasen. Alle tekstatributter samt LOB (Large Object) i en database kan indekseres.
- På fil. Alle dokumenter som befinner seg på filområder som er tilgjengelige for serveren.
- På internett. All tekst som er tilgjenelig via HTTP eller FTP.
- Brukerdefinert via PL/SQL prosedyrer. Dvs brukeren setter opp en funksjon hvis resultat blir indeksert.

Dokumentvask (Filter)

For at data skal bli indeksert på en konsekvent måte konverteres eventuelt andre formater, som PDF, doc osv, til HTML før det blir sendt vidre. Oracle Text

støtter over 150 forskjellige dokumentformater. Grunnen til at de konverterer til HTML er for å bevare noe av formateringen, som overskrifter og lignende, som grunnlag for rangering og søk av spesielle deler av et dokument senere. I tillegg til disse standardfilterene er det mulig å sette inn og spesialisere egne filtre for andre typer dokumenter eller egne bedriftsspesifikke filer for spesialbehandling.

Oppdeler (Sectioner)

Her deles hvert dokument inn i seksjoner eller avsnitt som en senere kan gjøre spørringer mot (se WITHIN eksemplet i avsnittet om XML søk nedenfor). Normalt vil dette være dokumentdeler som allerde er forhåndsdefinert i HTML eller befinne seg i XML dokumenter.

Leksikalsk analysator (Lexer)

Denne tar imot input fra oppdeleren og deler teksten videre inn i ”ord”. Her er ”ord” i gåsetegn fordi det ikke nødvendigvis er gitt hvordan en skal dele opp teksten. (se 3.1.3) Den tar også i bruk stoppord for å fjerne ”vanlige” ord, men i motsetning til mange andre markerer den hvor det var et stoppord og tar vare på dette. Ethvert stoppord som blir brukt i et søk vil da også matche ethvert stoppord i teksten. Stoppordlisten kan være egenspesifisert eller bruke en standard som følger med forskjellige språk. En kan også spesifisere en rekke andre egenskaper på hvordan teksten blir spittet opp. Eksempelvis kan en spesifisere hvordan en skal håndtere spesielle sammenbindingstegn, slik som på engelsk ”it’s”. I en rekke asiatiske språk, som for eksemple kinesisk skiller en ikke mellom ”ord” ved hjelp av mellomrom på samme måte som en gjør i vestlige språk. Andre regler gjelder her for hvordan en skal dele opp teksten. Oracle Text har innbygget funksjonalitet for å analysere kinesisk, japansk og koreanske tekster.

Indeksering (Indexing)

Til slutt lages en invertert indeks over ordene som blir produsert av den leksikalske analysatoren.

Når en oppretter en indeks i Oracle må en angi hva slags type indeks som skal lages. Det støttes tre typer:

- `CONTEXT`. Den vil normalt brukes for å lage en informasjonsgjenfinnings applikasjon, for eksempel for å indeksere store tekstdokumenter som i HTML, XML, PDF, doc, vanlig tekst osv.
- `CTXCAT`. Denne brukes normalt i forbindelse med kortere tekster, som for eksempel atributter i databasen som produktnavn, leverandør osv. Med denne typen indeks blir det lagt optimalt til rette for blandede spørringer, dvs spørringer der en bruker både tekst indeksen og andre indekser.
- `CTXRULE`. Brukes for å lage en klassifiserings applikasjon. Indeksen lages over en tabell av spørringer som hver har en klassifikasjon.
- `CTXXPATH`. Brukes for å optimalisere xpath søk i XML dokumenter.

Det tar lang tid å indeksere tekst. I utgangspunktet tilbyr ikke Oracle fullstendig samsvar mellom indeksen og data i databasen ved bruk av CONTEXT indeks

(dvs at dokumenter kan befinne seg i indeksen, men er ikke blitt indeksert enda), men en synkronisering med visse mellomrom. Hvor ofte en slik synkronisering skal foregå kan spesifiseres av brukeren. Indekser av typen CTXCAT, beregnet for kortere tekster vil imidlertid alltid være oppdatert og en transaksjon vil ikke fullføre før teksten er indeksert.

Som nevnt er det også mulighet for å lage regelindekser (`CTXRULE`) for å kunne automatisk klassifisere innkommende dokumenter i en bestemt kategori, også kjent som dokument ruting eller dokument filtrering. Da dette ikke er så relevant for oppgaven går jeg ikke noe særlig inn på dette her.

4.2.2 Søkemuligheter

Oracle Text inneholder funksjonalitet for å utføre søk basert på følgende strategier.

- Nøkkelordsøk. Dette er vanlig søk for å finne dokumenter som inneholder bestemte nøkkelord.
- Kontekstsøk. Brukeren har muligheter for å bestemme hvordan ordene skal stå i forhold til hverandre i teksten.
- Bruk av boolske operatorer for å begrense/utvide søket.
- Bruk av lingvistiske egenskaper for å kunne gjøre utydelige (fuzzy) søk.

For å kunne støtte slike funksjoner er det innebygd en strukturert synonymordbok, tesaurus. Denne har integrert alle tesaurus operasjoner spesifisert i ISO-2788 (Documentation - Guidelines for the development of monolingual thesauri) som synonymer og bredere/smalere term. Tesaurusen brukes for å utvide spørringen til å inkludere flere ord.

Intern struktur

Siden indeksfunksjonaliteten I Oracle Text er innebygget i databasen, ligger selve indeksen også her (i imotsetning til MSSQL som har en ekstern søketjeneste den benytter seg av). Indeksen består av fire tabeller referert til som $I, $K, $N og $R. Tabellene befinner seg i samme skjema som teksttabellen som indekseres og vil ha samme navn som den men konkatenert med sine respektive navn (eks. $I). Hvert dokument som er indeksert har en intern DOCID. I tabellen $I er en oversikt over de ord som er indeksert samt en binær representasjon av hvilke dokumenter (DOCID) de forekommer i og på hvilken posisjon. $K er et ”map” mellom interne DOCID’er til eksterne ROWID’er for kjapt oppslag fra DOCID til ROWID. $R er for det omvendte av $K, altså fra ROWID til DOCID. $N brukes ved optimalisering av indeksen og inneholder en liste over slettede DOCID’er.

Et eksempel på bruk kan være følgende:

```
CREATE INDEX philosopher_description_idx ON
      philosopher(description)
      INDEXTYPE IS CTXSYS.CONTEXT;

SELECT score(1), name, born
FROM philosopher
```

```
WHERE CONTAINS (description, 'french NEAR "Cogito, ergo sum"', 1)
> 0
      AND born > 1400
ORDER BY score(1) DESC;
```

Spørringen over er et eksempel på en sammensatt spørring, dvs at den benytter seg både av tekstsøkefunksjonalitet og annen "vanlig" SQL begrensning Forhåpentligvis vil søket returnere dokumenter som omhandler Descartes.

For at optimalisereren skal velge den beste eksekveringsplanen basert på kjøretids egenskapene til spørringen må den ha mulighet til å velge hva den skal gjøre i hvilken rekkefølge. Oracle tekst tilbyr derfor to separate måter å evaluere et tekstpredikat opp mot en kolonne:

- Rammeverket kan sette opp tekstindeksen som en kilde som gir ut et sett med ROWID'er som tilfredstiller predikatet for kjernen.
- Rammeverket kan svare på spørsmålet "Tilfredstiller denne raden med ROWID=X det gitte predikatet?"

XML støtte

Ved å lage en indeks av typen `CTXXPATH` blir den optimalisert for indeksering av XML dokumenter. Dermed blir det mulig å søke med egne operatorer i disse dokumentene. Blant annet kan en søke innenfor et atributt, eller gjøre såkalt stisøk for å hente ut spesielle deler av et dokument. Eksemplet nedenfor er ment for å vise noe av muligheten:

Gitt et dokument om filosofer:
<?xml version="1.0"?>
<philosopher name="David Hume">
<born>1711</born>
< deceased >1776</deceased>
<work>Treatise of Human Nature (1739 - 40)</work>
<abstract> British empiricist. He argued against the proofs for God's existence. In his Treatise of Human Nature (1739 - 40), he held that moral beliefs have no basis in reason, but are based solely on custom.</abstract>
<description> … </description>
</philosopher> …

Dette tillater søk over filosofer som var empirister (det står i sammendraget) som henter ut fødselsåret:

```
SELECT
p.philosopher.extract('/philosopher/born/text()').getStringVal()
FROM philosopher p
WHERE contains(philosopher, 'empiricist
INPATH(/philosopher/abstract)') > 0;
```

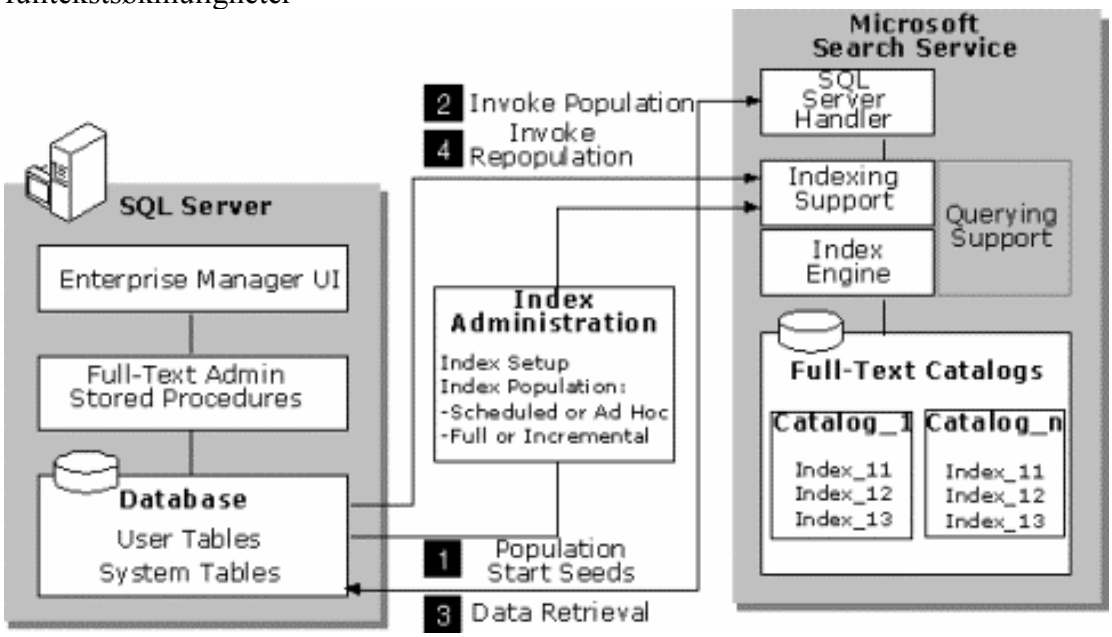
4.3 MSSQL

Fra og med Microsoft SQL Server 7.0 [12] er det integrert funksjonalitet for fulltekst søk. I motsetning til Oracle Text er ikke søkefunksjonaliteten fullstendig integrert i SQL serveren, men et påbygg som benytter seg av Microsoft Search

Service. Det betyr at indeksene ligger på filsystemet under kontroll av Microsoft Search Service og ikke av SQL serveren.

4.3.1 Arkitektur

Figuren nedenfor viser hvordan arkitekturen for tekstsøk i Microsoft SQL er lagt opp. SQL serveren har kun metadata som forteller om hvilke relasjoner som har fulltekstsøkmuligheter



Figur 7: Oversikt over MSSQL tekstsøk arkitektur

Det er lagt til en rekke funksjoner i databasen for å administrere databaser som skal ha fulltekstsøk funksjonalitet, samt at systemtabellene er utvidet noe. En database som er satt i stand for fulltekstsøk vil ha en eller flere kataloger knyttet til seg, mens én katalog kun kan være tilknyttet én database. Informasjon om hvilke atributter i hvilke tabeller som er linket til hvilke kataloger finnes i systemtabeller. Nedenfor vil jeg komme nærmere inn på en del av disse komponentene.

Microsoft søketjeneste (Microsoft Search Service)

Microsoft Search Service er en Windows NT tjeneste som har to hovedfunksjonerr: funksjonalitet for å bygge opp de inverterte indeksene, og tjenester for å behandle fulltekstsøk spørringer. Microsoft søketjeneste må kjøre på samme maskin som SQL tjeneren. Nedenfor følger en beskrivelse av hovedmodulene:

Indekseringsstøtte (Indexing Support)

Informasjon om at en tabell skal indekseres vil bli sendt til denne med en unik beskrivelse av hvilken database og relasjon som skal indekseres. Når tjenesten er klar for å behandle forespørselen vil den kontakte SQL server-håndtereren.

SQL server-håndterer (SQL Server Handler)

Denne driveren plugges inn som et komponent i Microsoft Search Service for å kunne behandle SQL Server data ved indekseringsprosessen.

Indekseringsmotoren (Index Engine)

Denne har til oppgave å gå gjennom innkommende data for å ”vaske” det før det blir indeksert. Foruten å bygge opp den inverterte indeksen er oppgaver å splitte opp data i ”ord” samt å fjerne de mest brukte ved hjelp av en stoppordliste.

Indekskatalogområde (Full-Text Catalogs)

Her vil de faktiske indeksene ligge. Dette vil være et normalt, men begrenset katalogområde på serveren. Normalt vil all fulltekstindeksdata for en database ligge i en slik katalog, men det er muligheter for å splitte dette opp.

SQL-serveren har altså ingen kontroll over indeksdataene bortsett fra at den har metadata som forteller hvilke atributter som har en slik indeks. Den har derfor heller ingen funksjonalitet for å koordinere sikkerhetskopiering og lignende med denne tjenesten, men den har mulighet for å kunne gjøre reindeksering.

4.3.2 Administrasjon

For å kunne tilby fulltekstsøk på et atributt må en gjennom følgende steg, her forklart ved bruk av funksjoner i databasen, men dette kan også ordnes via GUI i SQL Server Enterprise Manager. Hvis jeg ønsker å indeksere `description` attributtet i `philosopher` relasjonen som ligger i databasen `testing` må jeg:

1) Forsikre meg om at Microsoft Search Service kjører på maskinen.
2) Åpne for fulltekstsøk i databasen etter at jeg har koblet opp:
   ```
   sp_fulltext_database 'enable'
   ```
3) Lage en tom fulltekst katalog:
   ```
   sp_fulltext_catalog 'testingCatalog', 'create'
   ```
4) Siden tabellen må ha et unikt nøkkelatributt som kan returneres av funksjonene må jeg oppgi indeksen for dette når jeg skal tillate fulltekstsøk på en relasjon. La oss si at philosopher har en unik nøkkel indeks `PK_id`. Kan da åpne for fulltekstsøk på relasjonen:
   ```
   sp_fulltext_table 'philosopher', 'create',
   'testingCatalog', 'PK_id'
   ```
5) Så må jeg registrere hvilke attributter som skal bli indeksert:
   ```
   sp_fulltext_column 'philosopher', 'description',
   'add'
   ```
6) Før indeksen kan bli laget må jeg aktivere tabellen:
   ```
   sp_fulltext_table 'philosopher', 'activate'
   ```
7) Nå kan jeg endelig lage indeksen i katalogen:
   ```
   sp_fulltext_catalog 'testingCatalog',
   'start_full'
   ```

I tillegg må en eventuelt sette opp tjenester for reindeksere i ny og ne for å holde indeksen oppdatert.

4.3.3 Søkemuligheter

Microsoft SQL Server tillater to søkepredikater; `CONTAINS` og `FREETEXT`. Begge disse kan brukes i en `WHERE` del som en boolsk funksjon; `CONTAINS()` og/eller `FREETEXT()` eller i en FROM del, som en funksjon som returnerer tabellrader i form av `CONTAINSTABLE()` og `FREETEXTTABLE()`. Tabellfunksjonene gir en tabell med to attributter der det ene er nøkkelattributtet

i den indekserte relasjonen, mens det andre gir rangeringsverdien (mellom 0 og 1). Nøkkelattributtet som returneres er et av atributtene i relasjonen som har blitt bestemt som unik nøkkel. De radene som returneres er kun de som tilfredstilte søkebetingelsen.

Legg merke til at når spørreparseren i databasen går gjennom spørringen og finner et `CONTAINS` eller `FREETEXT` predikat vil den skrive om spørringen til å bruke `CONTAINSTABLE()`, respektive `FREETEXTTABLE()`, dvs gjøre det om til en `JOIN` spørring, men filtrere bort rangeringsattributtet etter at den har fått tilbake resultatet fra søketjenesten.

Spørrespråket i CONTAINS() og CONTAINSTABLE()

`CONTAINS()` og `CONTAINSTABLE()` har samme syntaks. Første parameter er en eller flere kolonner som det spørres mot, hvis `*` blir gitt, betyr det alle kolonner som er registrert for fulltekstsøk. Andre parameter er betingelsen. Betingelsen er et boolsk uttrykk satt sammen ved hjelp av `OR`, `AND` eller `AND NOT` av følgende uttrykk:

Enkelt uttrykk.

Brukes for å søke på bestemte ord eller uttrykk i teksten.

Trunkering

Brukes for å kunne søke på deler, starten av, ett eller flere ord. Dette markeres ved å bruke et enkelt uttrykk etterfulgt av en stjerne (*). Hvis en har et enkelt uttrykk bestående av flere ord, vil alle ordene i uttrykket utvides. ”mal kost *” vil for eksempel gi treff både på ”malaria kostnader” og ”malplasserte kosmetikere”.

Kontekst uttrykk

`NEAR` brukes når en ønsker å begrense søket til at to deluttrykk skal stå i nærheten av hverandre. Effekten blir som å bruke `AND` bortsett fra at rangeringen blir endret.

Generaliserings uttrykk/Utvidet uttrykk

`FORMSOF` brukes for å utvide et enkelt uttrykk slik at en tar med andre lingvistisk genererte former. Den som foreløpig er implementert er `INFLECTIONAL` som betyr at den tar med alle bøyningsformer av et verb eller substantiver. Eksempel på bruk i en `WHERE` del vil være: `WHERE CONTAINS(description, 'FORMSOF (INFLECTIONAL, bestemme)')` Dette vil gi treff på dokumenter som inneholder ”bestemt” og ”bestemmende”.

Vektet uttrykk

`I MSSQL gir ISABOUT` funksjonen mulighet for å gi ett av uttrykkene mer relevans enn ett annet. Ikke alle uttrykkene trenger å være i et dokument for at det skal gi treff, men jo flere som er jo høyere rangering vil den få.

Naturlig språk spørringer - FREETEXT() og FREETEXTTABLE()

`FREETEXT()` og `FREETEXTTABLE()` skal tilby grunnleggende naturlig språk spørringer. Spørringen blir delt opp i en rekke søkeuttrykk, genererer basalformen av ordene, tildeler heuristisk vekt, og prøver å finne dokumenter som inneholder meningen med søket, mer enn de eksakte ordene gitt. Syntaksen er eksempelvis

```
SELECT f.RANK, p.name, p.born
FROM philosopher AS p,
      FREETEXTTABLE(description, 'Hvem mente at all kunnskap
      kommer fra erfaring av årsak virkning og ikke logikk?') AS
      f
WHERE p.id = f.[KEY]
ORDER BY k.RANK DESC;
```

Der resultatet forhåpentligvis hadde vært dokumenter som omhandlet David Hume.

4.4 Eksisterende utvidelser i PostgreSQL

4.4.1 GiST

Etter som databasene kom i bruk utenfor de tradisjonelle bruksområdene, ble det behov for stadig nye spesialiserte søketrær for å løse applikasjonsspesifikke problemer. GiST, Generalized Search Tree, [41-43] er en aksessmetode implementasjon, som kan utvides til å implementere spesialiserte søketrær som B+-tre og R-tre. Det er utviklet som et forskningsprosjekt for å lete etter likheter mellom forskjellige søketrær. For eksempel viste de at R- og B-tre implementasjonene i PostgreSQL som begge var implementert med rundt 3000 linjer kode kunne implementeres ved bruk av GiST på rundt 500 linjer hver.

GiST er en implementasjon av et generelt balansert søketre der hver node har et predikat og en peker. Eksempelvis vil predikatet i et B-tre være et tallskille og pekeren vil skille mellom de som er over og under det tallet. For å spesialisere et generelt GiST tre må en kun implementere et sett av funksjoner (consists, union osv) som behandler de spesifikke sammenligningen av en gitt datatype, mens den generelle implementasjonen vil ta seg av resten av trestruktur implementasjonen som lagring, generell søking og integrasjon mot PostgreSQL for å støtte databasespesifikke egenskaper som transaksjoner og join.

4.4.2 TSearch

Tsearch er en modul som kan lastes inn i PostgreSQL for å muliggjøre fulltekstsøk. Den gjør dette ved å benytte seg av GiST. Det er egentlig bare en ny postgresqltype, 'txtidx', som er en streng med "ord" satt innenfor apostrofer skilt med mellomrom. Apostrofer og mellomrom innen ett "ord" må derfor markeres med en '\' foran. Det er igjen data av typen 'txtidx' som bli indeksert for å tillatte fulltekstsøk. Dette er ikke spesielt brukervennlig, så det følger med et sett av funksjoner for å gjøre bruken enklere. Disse funksjonene og typene må lastes inn i hver database som skal benytte seg av denne funksjonaliteten.

I modulen følger det med to funksjoner for å automatisk generere slike: txt2txtidx() og tsearch(). txt2tidx() benytter seg av morfologi når den skal generere listen med ord, slik at kun ord i sin basisform vil bli indeksert, tsearch() derimot bare splitter opp dokumentet. Begge funksjonene vil bruke stoppordlister for å fjerne de vanligste ordene. Hvilken variant en velger avhenger av hvilken operator en ønsker å bruke for å søke. Tsearch muliggjør nemlig to typer operatorer mot data av typen 'txtidx': '##' og '@@'.

Valget av operator avhenger i sin tur av hvordan en ønsker at søket skal gjøres. '##' operatoren tar et søk av typen 'mquery_txt' og benytter seg av morfologi for å gjøre om spørringen til sin basisform før den slår opp i indeksen, mens '@@' operatoren tar et søk av typen 'query_txt' og beholder søket i sin originalform. Eksemplet nedenfor viser forskjellen:

```
SELECT 'cats&philosophers'::mquery_txt,
'cats&philosophers'::query_txt;
```

Vil returnere:

```
'cat' & 'philosoph' og 'cats' & 'philosophers'
```

Hvilken funksjonon en velger for å generere data til 'txtidx' atributtet avhenger derfor av hva slags type spørringer en ønsker å gjøre senere.

For å benytte seg av tsearch må en derfor legge til et attributt i tillegg til tekstatributtet som en ønsker å søke i. Dette attributtet vil inneholde listen av ord som forekommer generert opp av enten txt2txtidx() eller tsearch() og en vil normalt lage en trigger som vil sørge for at oppdateringer i attributtet vil medføre at denne verdien beregnes på nytt. Dette nye "ekstraattributtet" (i eksemplet nedenfor kalt dataidx) lager så en indeks ved hjelp av indeksaksessmetoden gist. Eksemplet nedenfor viser så hvordan et fulltekstsøk kan utføres ved hjelp av tsearch:

```
SELECT title, data FROM data_table WHERE dataidx ##
'Kant&philosopher';
```

TSearch benyttes bl.a. i OpenFTS (Open Source Full Text Search engine)[44;45], en åpenkildekode søkemotor. OpenFTS bruker PostgreSQL med TSearch for manipulasjon av fulltekstindeksen, men oppgaver som crawling, dokumentvask, spørreprosessering og lingvistiske operasjoner foregår utenfor databasen.

4.5 Implementasjon av utvidede indekser i POSTGRES

I artikkelen [46] beskriver forfatterne en prototyp der de implementerer en ny aksessmetode i PostgreSQL. Prototypen er en ny type indeksaksessmetode Functional B-Tre, FB-Tre, som muliggjør indeksering av funksjoner på atributter, rettere sagt indeksere opp resultatet av en funksjon. Bakgrunnen for dette er ønsket om en måte å indeksere på som kan benyttes i flere domener. Eksemplet som blir benyttet i artikkelen er en bibliografisk database der en

ønsker å indeksere nøkkelordene i sammendragene til artikler. Dette gjøres ved minst mulig forandringer i selve databasen og ved å benytte seg av den allerede uttestede B-Tre implementasjonen som er i databasen. Det argumenteres videre for at denne formen for utvidelser kan være ønskelig for flere IR systemer fordi en kan utnytte fordelen ved datauavhengigheten som gis i et RDBMS. Flere IR systemer tar utgangspunkt i en invertert filstruktur og brukeren eksponeres for den underliggende datastrukturen og dens begrensninger. Et eksempel på spørring fra prototypen de implementerte er:

```
SELECT title FROM reports
WHERE extract(reports.abstract) @= ”database”;
```

Extract(),er en egendefinert funksjon som trekker ut nøkkelordene fra en tekst og returnerer disse som en liste. Videre implementerte de en likhetsoperator (@=) som kunne sammenligne disse, dvs nøkkelordene som returneres fra funksjonen mot et enkelt nøkkelord. Hvis en måtte trekke ut alle nøkkelordene for hvert søk ville det vært en veldig tidkrevende operasjon, men hvis en kunne indeksere opp dette resultatet, ville søketiden bli betraktelig mindre. Dette ble implementert ved at den nye indeksaksessmetoden FB-Tre ved innsetting mottok teksten (sammendraget) og så splittet dette opp og igjen satte de enkelte ordene inn i et vanlig B-Tre.

De påpekte at siden PostgreSQL er et databasesystem basert på systemtabeller for mulig utvidelser, og med mulighet til å laste objektfiler uten å måtte rekompilere systemet, var en slik utvidelse mulig uten for mye arbeid. Det er mye av den samme tankegangen vi har benyttet i vår prototype selv om dette ble gjort på en gammel versjon av databasen som bl.a. benyttet seg av PostQUEL i stedet for SQL.

Det er verdt å nevne at i dagens versjon av PostgreSQL er det mulighet for indeksering over funksjonsresultater, men likevel ikke mulig å gjøre slik som beskrevet i artikkelen. Årsaken er at databasen ikke ville visst hvordan den skulle indeksert listen av nøkkelord da indeksering kun er mulig over funksjoner som returnerer enkeltverdier.

4.6 Integrering av IR og RDBMS ved kooperativ indeksering

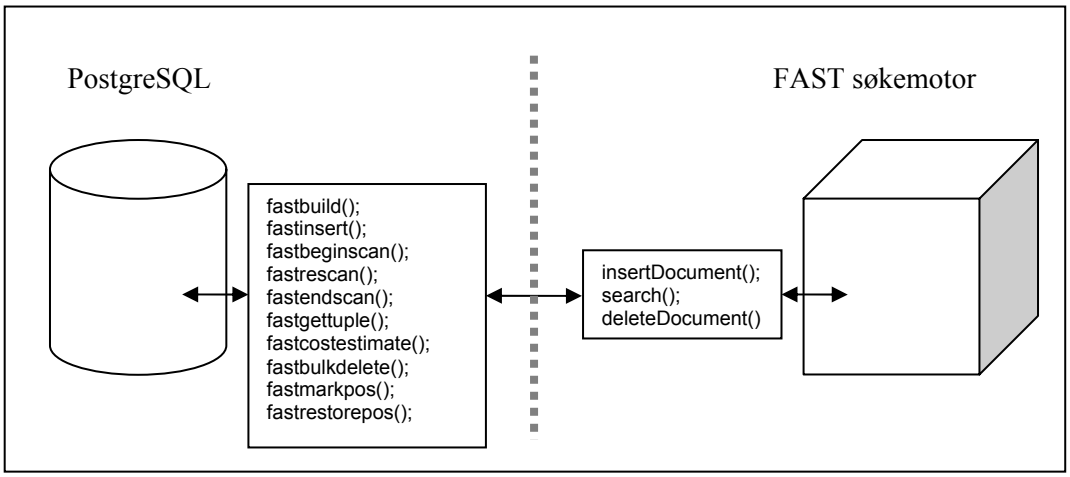
Forskere fra Oracle og universitetet i Massachusetts presenterer[6] en prototyp som implementerer en form for invertert filstruktur ved hjelp av relasjoner i databasen. De tok utgangspunkt i at databaser vil en ha en annen hyppighet av oppdateringer og endringer enn i tradisjonelle IR-systemer. Derfor valgte de en struktur der dokumentene lå som egne tupler, men også hadde attributter for metainformasjon for å kunne gjøre rangering på en god måte (bl.a. ordfrekvenser). Den ”inverterte tabellen” hadde ett tuppel per nøkkelord per dokument. Normalt vil en i en IFS (se 3.2) kun ha en forekomst av et nøkkelord med pekere til de aktuelle dokumentene. Dette ble begrunnet med at ved endringer i datagrunnlaget trengte en ikke å bygge den iverterte strukturen på nytt, men simpelthen slette de tuplene som var aktuelle for det gitte dokumentet.

Disse relasjonene ble så indeksert på forskjellige måter både med enkelt og multikolonneindekser (går over flere attributter) ved hjelp av eksisterende aksessmetoder (B*-tre). Det viste seg ved eksperimentering at dette ikke holdt mål ved større mengder data, men at denne implementasjonen medførte en tidsøkning som var høyere enn lineær. Jo mer data som lå i den inverterte tabellen dess lengre tid brukte den og jo mer lagringsplass ble brukt på disse strukturene (>3 ganger tekstmengden).

De gjorde så utvidelser i databasen for å tilby en mer tilpasset datastruktur. De implementerte en versjon av B+-treet der en også lagret alle data som lå i relasjonen som ble indeksert. Begrepet INDEX ONLY ble innført slik at en relasjon faktisk ikke trengte å inneholde noen tupler fysisk da all data lå lagret i indeksen. Dette viderefører de i konseptet *kooperativ indeksering* (cooperative indexing). Med dette mente de at både en applikasjon og DBMS hadde ansvar for å bygge og tolke indeksstrukturer og at deres aksessmetode kun var et eksempel. Dette er mindre aktuelt for oppgaven, og vil ikke omtales nærmere. Det viktigste for vårt arbeid var tankegangen om at databasen fortsatt overholder transaksjonsstøtte og lignende, men samtidig tilbyr en form for utvidbar aksessmetodeimplementasjon.

KAPITTEL 5: IMPLEMENTERING AV INDEKSAKSESSMETODEN

I dette kapittelet presenteres vår prototyp der vi implementerte en ny indeksaksessmetode i PostgreSQL for å muliggjøre fulltekstsøk ved hjelp av FDS. Først vil jeg gi en kort oversikt over prototypen og felles strukturer i de to modulene. Deretter vil jeg beskrive systemkataloger i PostgreSQL og implementasjonen/integrasjonen som ble gjort mot disse systemkatalogene for å legge til en ny aksessmetode. Til slutt vil jeg beskrive implementasjonen og konfigurasjonen som ble gjort mot FDS.



Figur 8 Oversikt over prototypen

Figur 8 viser hovedtrekkene i prototypen. Modulen mot PostgreSQL består av et sett metoder som blir kalt fra backend ved bruk av aksessmetoder. Disse er implementert i C slik at vi kan benytte oss av rammeverket i kildekoden til databasen. Denne modulen tar seg av all kommunikasjon med databasen og har ikke som forutsetning at det er FDS som indekserer dokumentene eksternt, men tilbyr et forenklet API for indeksering av dokumenter eksternt. Det eneste som forutsettes er at det finnes en implementering av tre metoder for bruk av ekstern indeks, henholdsvis for innsetting, sletting og søk. Modulen mot FDS er en implementasjon av disse tre metodene for manipulasjon av data og er skrevet i C++ for å benytte FDS sitt eget API for kommunikasjon med søkemotoren. Vi definerte følgende felles strukturer for bruk til overføring av data mellom de to modulene.

5.1 Felles strukturer

Figur 9 gir en oversikt over de strukturer som overføres mellom de to modulene. Det eneste som overføres fra den eksterne indeksen er altså en lenket liste over de tuplene (identifisert ved blokkid og offset) som søket traff på. Alt utenom FastSearchResult er kopi av PostgreSQL strukturer. Vi valgte likevel å redefinere dem som egne strukturer fordi vi dermed unngikk å importere PostgreSQL spesifikke headerfiler inn i eksterne søkemodulen skrevet i C++. Dette både fordi

vi ønsket å holde dem så separert som mulig (for eventuelt å gi mulighet til implementasjon av andre eksterne søkemoduler) og fordi vi fikk en navnekonflikt i PostgreSQL headerfiler som brukte reserverte C++ ord.

```
Typedef unsigned short fastuint16;
Typedef fastuint16 FastOffsetNumber;
Typedef struct FastBlockIdData
{
      fastuint16  bi_hi;
      fastuint16  bi_lo;
} FastBlockIdData;

Typedef struct FastItemPointerData
{
      FastBlockIdData ip_blkid;
      FastOffsetNumber ip_posid;
} FastItemPointerData;

Typedef FastItemPointerData *FastItemPointer;

Typedef struct FastSearchResult
{
  struct FastSearchResult* next;
  FastItemPointer item;
} FastSearchResult;
```

Figur 9 Delte datastrukturer

5.2 Systemkataloger i PostgreSQL

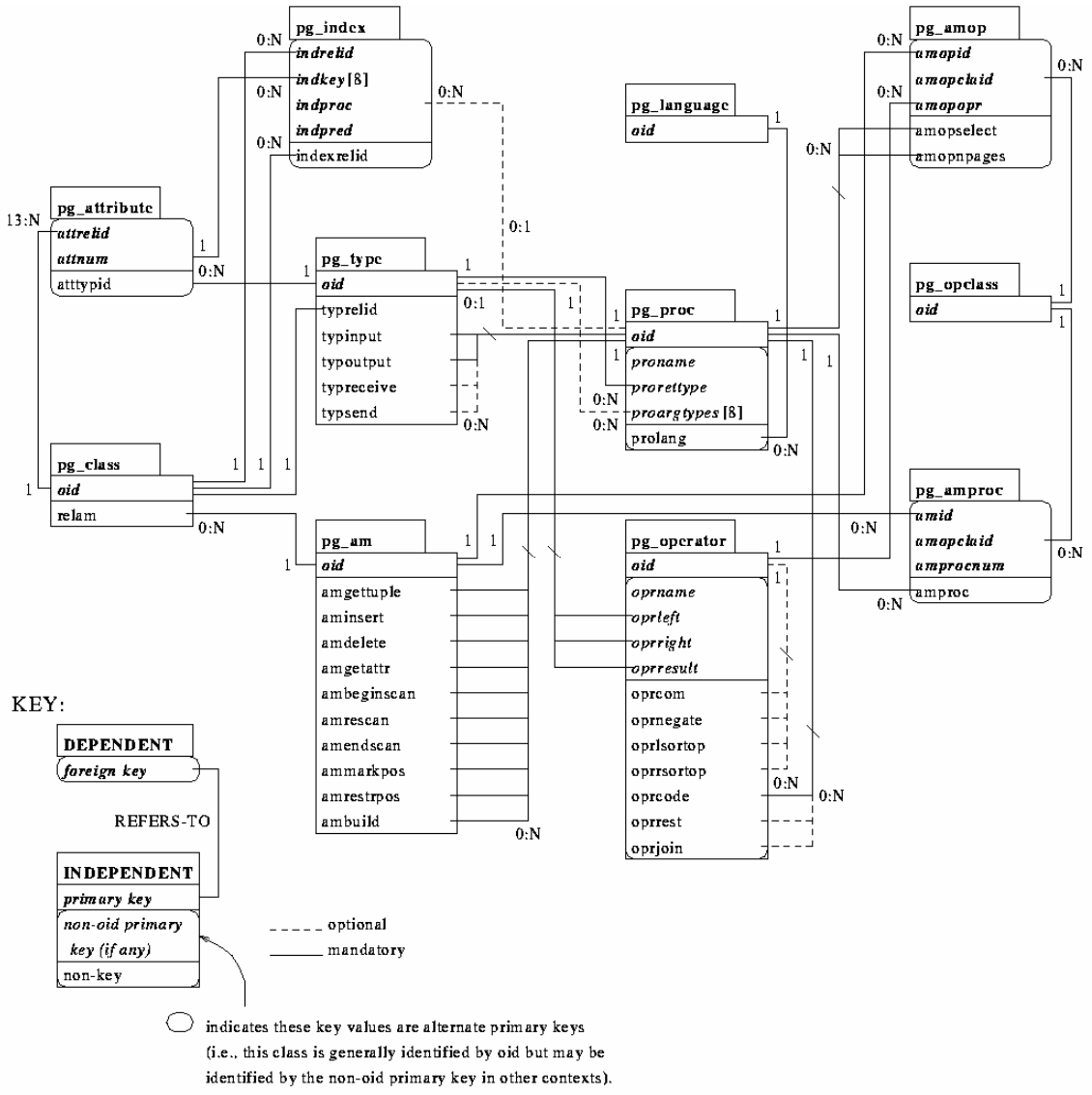
En av hovedgrunnene til at vi valgte PostgreSQL er at den i så stor grad belager seg på bruk av systemkataloger. I store relasjonsdatabaser brukes systemkatalogene til å lagre informasjon om databaser, tabeller, kolonner i tabeller osv. Disse presenteres som vanlige relasjoner til en bruker, og DBMS'en bruker dem internt. I PostgreSQL finnes også annen informasjon som datatyper, funksjoner og aksessmetoder (indekser). Disse tabellene kan endres av en bruker, og dermed kan PostgreSQL utvides uten å endre kildekode. En annen sterk egenskap ved PostgreSQL er at en kan laste egenskrevet kode ved bruk av dynamisk lasting av objekt kode [47]. Mange av de vanligste systemkatalogene blir manipulert via SQL kommandoer, som `CREATE FUNCTION`, mens andre må en manipulere direkte.

Tabell 1 gir en oversikt over de viktigste systemkatalogene i PostgreSQL og deres funksjon. Figur 10 nedenfor viser også sammenhengen mellom disse. I vår implementasjon utvidet vi `pg_am`, `pg_amop` og `pg_opclass`.

Katalog navn	**Beskrivelse**
`pg_database`	Databaser på serveren
`pg_class`	Tabeller (bl.a.)
`pg_attribute`	Tabell kolonner
`pg_index`	Indekser
`pg_proc`	Funksjoner
`pg_type`	Data typer (bade enkle og komplekse)

Katalog navn	Beskrivelse
pg_operator	Operatorer
pg_aggregate	Aggregatfunksjoner
pg_am	Aksessmetoder
pg_amop	Aksessmetode operatorer
pg_amproc	Aksessmetode hjelpefunksjoner
pg_opclass	Aksessmetode operatorklasse

Tabell 1: Liste over de viktigste systemkatalogene i PostgreSQL



Figur 10: Diagram over de viktigste systemkatalogene i PostgreSQL og relasjonene mellom dem (hentet fra [47])

5.3 Implementasjon og integrasjon mot PostgreSQL

Oppbyggingen av systemkataloger i PostgreSQL gjør at vi kan legge til og fjerne indekser og typer ved å legge til tupler i systemkataloger med informasjon om funksjoner som skulle gjøre jobben. Dette viste seg imidlertid å være dårlig dokumentert, antagelig fordi det ikke er så vanlig å lage egne indekser. Ved å studere kildekoden til B-tre og hash-indeksen forstod vi hva som måtte gjøres. Nedenfor følger en nærmere beskrivelse av arbeidt som ble gjort.

5.3.1 Utvikling av aksessmetoden og registrering i pg_am

I pg_am ligger oversikten over de indeksaksessmetodene som støttes av databasen [48]. Normalt vil en her finne ”standard” indekser som btree, rtree, hash og gist. Vi la her til en ny aksessmetode som vi kalte *fast*. Tabell 2 viser atributtene i pg_am, en kort forklaring på hva de betyr samt hvilke valg vi gjorde. De siste attributtene inneholder referanse til funksjoner. Rettere sagt OID’er til tuppelet der funksjonen er registrert i *pg_proc*. I tabellen viser jeg for enkelhetsskyld navnet på funksjonen i stedet. Disse funksjonene hadde vi skrevet på forhånd og utgjør ”selve” implementasjonen av indeksaksessmetoden mot PostgreSQL. Nærmere beskrivelse av disse funksjonene følger i neste avsnitt.

Attributt	**Type**	**Forklaring**	**Vårt valg**
`Amname`	`name`	Navn	`fast`
`Amowner`	`int4`	Id til eier (ikke i bruk)	`1`
`amstrategies`	`int2`	Antall operator strategier	`1`
`amsupport`	`int2`	Antall hjelpemetoder.	`0`
`amorderstrategy`	`int2`	Sorteringsstrategi.	`0` (ingen)
`amcanunique`	`bool`	Unike indekser	`false` (kan ikke brukes for å overholde unikbegrensning)
`amcanmulticol`	`bool`	Flere kolonners indeks.	`false` (indekserer bare ett atributt.)
`amindexnulls`	`bool`	Indeksere NULL verdier	`false` (gir ingen mening)
`amconcurrent`	`bool`	Samtidig oppdatering	`true` (men det må indekseringsmotoren ta seg av)
`ambuild`	`regproc`	Lager en ny indeks.	`fastbuild`
`aminsert`	`regproc`	Setter inn data.	`fastinsert`
`ambeginscan`	`regproc`	Starter et søk.	`fastbeginscan`
`amendscan`	`regproc`	Slutter et søk	`fastendscan`
`amrescan`	`regproc`	Gjenopptar et søk. (kalles bl.a. når queryoptimalisereren ønsker å splitte et søk opp i to delsøk, f.eks ved bruk av OR)	`fastrescan`
`amgettuple`	`regproc`	Gir neste tuppel i et søk.	`fastgettuple`
`ammarkpos`	`regproc`	Markerer nåværende søk.	`fastmarkpos`
`amrestrpos`	`regproc`	Gjenoptar et tidligere pågående søk.	`fastrestorepos`

Attributt	Type	Forklaring	Vårt valg
`ambulkdelete`	`regproc`	Brukes for å rydde opp I en indeks. Kalles bl.a. av VACUUM.	`fastbulkdelete`
`amcostestimate`	`regproc`	Estimerer kostnaden ved å gjøre et søk I indeksen.	`fastcostestimate`

Tabell 2: Oversikt over pg_am med beskrivelse og hvilke verdier vi satte inn

fastbuild()

Denne funksjonen tar seg av alt som skal gjøres når en lager en indeks på et atributt, eksempelvis:

```
CREATE INDEX fts_index ON tabell USING fast (data);
```

I vårt tilfelle blir det lite, i og med at indeksstrukturen ikke skal lages lokalt, men tas vare på i en ekstern indeks. Funksjonen må sørge for at data som allerede er i tabellen når indeksen blir laget også blir indeksert. Det gjør den ved at den kaller på postgreSQL-rammeverkmetoden `IndexBuildHeapScan`. Den tar med en callback-metode som parameter, og denne vil sørge for å sette inn data ved å igjen kalle på vår hjelpemetode `_fast_doinsert`.

fastinsert()

Denne funksjonen blir kalt fra backend ved innsetting av data som skal bli indeksert.

```
Datum fastinsert(PG_FUNCTION_ARGS)
{
  //The index relation:
  Relation rel = (Relation) PG_GETARG_POINTER(0);
  //The data to index:
  Datum *datum = (Datum *) PG_GETARG_POINTER(1);
  //Pointer to the data on disk:
  ItemPointer ht_ctid = (ItemPointer) PG_GETARG_POINTER(3);
  //Resultstruct:
  InsertIndexResult res;
....
  res = _fast_doinsert(rel, datum, ht_ctid);
  PG_RETURN_POINTER(res);
}

InsertIndexResult _fast_doinsert(Relation rel, Datum * datum,
          ItemPointer ht_ctid) //IndexTuple itup)
{
  TupleDesc itupdesc;
  itupdesc = RelationGetDescr(rel);
  InsertIndexResult res;

  text* t = DatumGetTextP(*datum);
....
  char * result = insert_document(str,
                       ItemPointerGetBlockNumber(ht_ctid),
                       ItemPointerGetOffsetNumber(ht_ctid),
                       RelationGetRelid(rel),
```

```
                        DatabaseName,
                        host);
....
  return res;
}
```

Det viktigste den gjør er å kalle på hjelpemetoden `_fast_doinsert()` som vil bygge opp riktige strukturer og kalle `insert_document()` i FAST modulen (se 5.4.2) som vil sette dokumentet inn i den eksterne indeksen.

fastbeginscan() og fastendscan()

Fastbeginscan blir kalt fra backend når det skal gjøres et søk i indeksen. Den kalles bare én gang pr spørring, og har normalt som oppgave å gjøre klart for et søk, sette låser, bygge eventuelle strukturer osv. Dette er noe som ikke er så aktuelt for oss i og med at PostgreSQL rett etterpå vil kalle `fastrescan()` og vi vil gjøre det faktiske søket der (se `fastrescan()` for forskjell og årsaken til dette). Fastendscan blir kalt når et søk er ferdig og vil frigjøre tildelt minne.

fastrescan()

Denne funksjonen blir kalt hver gang backend ønsker at det skal gjøres et oppslag i indeksen. Eksempelvis vil queryoptimizeren i en spørring som

```
SELECT * FROM table
WHERE data contains 'hovedfagsoppgave' OR data contains
'masterthesis';
```

sannsynligvis dele opp dette i to indeksoppslag (hvis den bestemmer seg for å bruke indeks), ett per betingelse, og så sette sammen svaret i ettertid. Dette vil medføre ett kall til `fastbeginscan()` og to kall til `fastrescan()`. Fastrescan har som oppgave å lage en struktur som sendes videre til `fastgettuple()`. I de andre indeksene i en database, som B-tre, vil dette være en peker til hvor i indeksen en er, slik at en kan traversere seg videre derfra. I tilfellet med en ekstern indeks vil det neppe lønne seg å gå ut til indeksen for hvert treff og be om å få neste, i mange tilfeller vil det trolig ikke være mulig heller.

I vårt filfelle kaller vi i stedet for `_do_fastscan()` i FAST modulen (se 5.4.2) som vil utføre søket i den eksterne indeksen i sin helhet og så returnere alle treffene som en lenket liste (`FastSearcResult`). Vi bygger også opp datastrukturer som blir benyttet av `fastgettuple()` for å navigere oss gjennom søkeresultatet samt kunne frigjøre minne ved endt søk.

fastgettuple()

Denne metoden har som oppgave å fortelle backend hvilket tuppel som er det neste treffet. Normalt vil dette eksempelvis være å traversere videre i søketreet til en finner neste node som tilfredstiller kriteriet. I og med at selve søket ble utført i `fastrescan()` og alle treffene hentet ut her blir bare oppgaven å markere neste i listen som neste treff og returnere `true`. Når det ikke er flere treff, fortelles dette til backend ved å returnere `false`. Dette vil da føre til at `fastendscan()` blir kalt og søket er ferdig.

fastmarkpos() og fastrestorepos()

Disse funksjonene vil kaste en feil hvis de blir kalt. Normalt vil deres oppgaver være å markere/lagre hvor en befinner seg i et søk i indeksen og lese dette opp igjen for å fortsette søket. De benyttes når en bruker indeksen for å gjøre joinoperasjoner. De fleste tilfeller der en utfører join vil ett av attributtene være fremmednøkkel til en annen relasjon. Det gir liten mening å bruke et stort tekstfelt som fremmednøkkel og heller liten mening å gjøre en join mellom to tekstfelter basert på fulltekstsøk generelt. Vi har derfor valgt å ikke implementere dem i prototypen vår.

fastbulkdelete()

Når en gjør en `DELETE` kommando i PostgreSQL blir ikke indekser informert om dette. Tuplene markeres bare som slettet, men bevarer sin id og plass på disken. Når slike tupler likevel ikke blir rapportert som treff i et indeksoppslag, kommer det av at backend oppdager det, og ikke rapporterer et slettet tuppel som treff. PostgreSQL har en egen kommando for å "rydde opp" i databasen; `VACUUM`. Denne kjøres normalt en gang i døgnet, men kan også kjøres manuelt fra kommandolinje. Når VACUUM blir kalt,vil PostgreSQL fjerne eventuelle slettede elementer. Hvis disse er indeksert vil indeksaksessmetoden få beskjed om dette ved at `fastbulkdelte()` kalles. I denne funksjonen gjør vi så et sekvensielt scan over alle data og fjerner de dokumenter som skal slettes i den eksterne indeksen ved å kalle på `_do_fastdelete()` (se 5.4.2).

```
Datum fastbulkdelete(PG_FUNCTION_ARGS)
{
  Relation  indexRel = (Relation) PG_GETARG_POINTER(0);
  IndexBulkDeleteCallback callback =
             (IndexBulkDeleteCallback) PG_GETARG_POINTER(1);
  void *callback_state = (void *) PG_GETARG_POINTER(2);
....
  /* get the relation the index */
  Form_pg_index indexData = indexRel->rd_index;
  Relation heap = RelationIdGetRelation(indexData->indrelid);
....
  /* walk through the entire relation */
  hscan = heap_beginscan(heap, SnapshotAny, 0, (ScanKey) NULL);

  while (heap_getnext(hscan, ForwardScanDirection))
    {
      if (callback(&hscan->rs_ctup.t_self, callback_state))
        {
          ItemPointerData heaptup = hscan->rs_ctup.t_self;
....
          char* res =
             _do_fastdelete(ItemPointerGetBlockNumber(&heaptup),
                            ItemPointerGetOffsetNumber(&heaptup),
                            RelationGetRelid(indexRel),
                            DatabaseName, host);
....
  result->num_pages = num_pages;
  result->tuples_removed = tuples_removed;
  result->num_index_tuples = num_index_tuples;

  PG_RETURN_POINTER(result);
```

```
}
```

`Fastbulkdelete()` har også ansvar for å oppdatere systeminformasjon om antall tupler og antall sider som relasjonen bruker for bruk ved spørreplanlegging. De innebygde indeksene i PostgreSQL går her gjennom indeksen for å få hurtigere tilgang på dataene, men siden vi i prototypen ikke lagrer noe om indeksen lokalt er vi her nødt til å traversere relasjonen uten bruk av indeks.

fastcostestimate()

Når en gjør en spørring mot databasen finnes det nesten alltid flere mulige løsninger. For at spørreplanleggeren skal kunne gjøre et best mulig valg forsøker den å beregne kostnaden ved de forskjellige løsningene. Dette basert på estimater av antall tupler, antall unike verdier til et gitt atributt, om det er indekser på de gitte atributtene det skal gjøres en betingelse på osv. I og med at vi ønsker at indeksen alltid skal bli brukt setter vi her `indexStartupCost` og `indexTotalCost` til 0. Dette fordi vi ikke har noen mulighet for å evaluere et fulltekstsøkeuttrykk uten å gå via FDS.

Registrering av den nye aksessmetoden

Da vi hadde implementert metodene over, måtte vi registrere dem i PostgreSQL. Databasen tillater dynamisk lasting av C og C kompatibel C++ kode eksempelvis slik:

```
CREATE OR REPLACE FUNCTION fastbuild(internal, internal,
internal) RETURNS void
       AS 'fast_index.so' LANGUAGE 'c';
```

Når en legger til en ny funksjon i PostgreSQL ved hjelp av CREATE FUNCTION vil dette resultere i at det kommer et nytt tuppel i systemkatalogen *pg_proc* [47]. Da alle de nødvendige funksjonene var lagt til, kunne vi registrere den nye aksessmetoden ved å legge inn ett tuppel i *pg_am* slik:

```
INSERT INTO pg_am (amname, amowner, amstrategies, amsupport,
amorderstrategy, amcanunique, amcanmulticol, amindexnulls,
amconcurrent, amgettuple,  aminsert, ambeginscan,  amrescan,
amendscan,  ammarkpos,  amrestrpos,  ambuild,  ambulkdelete,
amcostestimate)
       VALUES ('fast', 1,  1,  0, 0,  false,  false, false, true,
         (SELECT oid FROM pg_proc WHERE proname='fastgettuple'),
         (SELECT oid FROM pg_proc WHERE proname='fastinsert'),
          ...
         (SELECT oid FROM pg_proc
               WHERE proname='fastcostestimate')
);
```

PostgreSQL er nå klar over at det finnes en ny indeksaksessmetode med navn *fast* og en kan indeksere et attributt i en tabell ved hjelp av SQL-kommandoen:

```
CREATE INDEX index_name ON table USING fast(attribute_name);
```

5.3.2 Knytte aksessmetoden opp mot en operator

Ovenfor har jeg beskrevet hva vi gjorde for å lage og registrere en ny indeksaksessmetode i PostgreSQL, men den har liten nytte så sant det ikke går an å søke med den. Vi må derfor knytte aksessmetoden opp mot en operator på samme måte som B-treindekser benyttes med en av operatorene =, >=, >, <, <= og !=.

For å få til dette må vi lage en ny operator, CONTAINS, registrere denne i en ny operatorklasse og så binde aksessmetoden til denne operatorklassen. Grunnen til denne tredelingen er at flere aksessmetoder kan benyttes med flere operatorer på flere datatyper.

Definere CONTAINS operatoren

Operatorer blir registrert i systemtabellen *pg_operator*, men kan manipuleres ved hjelp av kommandoen CREATE OPERATOR. På grunn av begrensninger i PostgreSQL kan ikke en operator bestå av ”vanlige” bokstaver. Vi valgte i stedenfor @@ som operator. Jeg kommer imidlertid fortsatt til å referere til CONTAINS som operatornavnet – bortsett fra i kjørende eksempler. Vi definerte operatoren slik:

```
CREATE OPERATOR @@ (
       leftarg   = text,
       rightarg  = text,
       procedure = contains_operator
       );
```

Det er altså en operator som mottar to argumenter av typen text og er implementert i funksjonen *contains_operator*. Dette er en ”dummy” funksjon som vi implementerte og lastet som de andre funksjonen beskrevet over, men som kun vil kaste en feilmelding. Dette fordi denne operatoren ikke skal kunne benyttes uten å benytte seg av indeksen (vi har ikke mulighet til å avgjøre om et gitt dokument evaluerer sant gitt en fulltekstspørring). En nærmere beskrivelse/analyse av dette kommer i 7.2.1.

Definere en ny operatorklasse

Operatorklasser, beskrevet i systemkatalogen *pg_opclass*, gir en oversikt over hvilke datatyper en indeksaksessmetode kan behandle. I og med at vi kun implementerte denne integrasjonen til å behandle tekst blir det kun ett tuppel her:

```
INSERT INTO pg_opclass (opcamid, opcname, opcintype, opcdefault,
       opckeytype, opcnamespace, opcowner)
VALUES ((SELECT oid FROM pg_am WHERE amname='fast'),
       'fast_op_class',
       (SELECT oid FROM pg_type WHERE typname='text'),
       true,
       0,
       (SELECT oid FROM pg_namespace WHERE nspname = 'public'),
       1);
```

Knytte operatoren opp mot indeksaksessmetoden

Til slutt kunne vi knytte CONTAINS operatoren opp mot aksessmetoden vår. Denne relasjonen finnes i *pg_amop*. Her vil det være ett tuppel per operator knyttet til hver operatorklasse. Vi registrerte denne tilknytningen slik:

```
INSERT INTO pg_amop
  (amopclaid, amopstrategy, amopreqcheck, amopopr)
VALUES (
  (SELECT oid FROM pg_opclass
   WHERE opcamid =
      (SELECT oid FROM pg_am WHERE amname='fast')
   AND opcname='fast_op_class'), --Our operator class
   1, --Strategy no 1
   false, --Recheck the hit? Can't do that outside fast
  (SELECT oid FROM pg_operator
      WHERE oprname='@@'
      AND oprleft=(SELECT oid FROM pg_type WHERE typname='text')
      AND oprright=(SELECT oid FROM pg_type WHERE typname='text')
   ) - Our operator
);
```

Da dette var gjort hadde PostgreSQL nok informasjon i systemtabellen til å skjønne at CONTAINS operatoren var knyttet opp mot fast-indeksen. Hvis det kom en spørring som benyttet seg av CONTAINS operatoren, ville den derfor undersøke om attributtet som operatoren ble brukt på var indeksert med indeksaksessmetoden vår. I så fall ville den be indeksen om et kostnadsoverslag ved å kalle på `fastcostestimate()`. Denne ville som nevnt returnere 0 og spørreplanleggeren vil dermed benytte seg av aksessmetoden vår.

5.4 Implementasjon og konfigurasjon mot FDS

Implementasjonen mot FDS bestod i hovedsak av to deler. Det ene var å konfigurere FDS slik vi trengte den. Det andre var å implementere et kommunikasjonslag som ble kalt fra metodene vi hadde lastet som en del av indeksaksessmetoden beskrevet over. Dette kommunikasjonslaget bruker FDS's programmeringsgrensesnitt, API, for å manipulere data i søkemotoren.

5.4.1 Konfigurasjon av FDS

Som nevnt er FDS en meget konfigurerbar søkemotor – hvor det meste kan konfigureres gjennom et sett av filer. Indeksprofilen, som muliggjør domenespesifikke søk, er en av de viktigste. Den kan sees på som databaseskjemaet til FDS (bestående kun av en relasjon), med dokumenter som tupler. Indeksprofilen definerer de individuelle søkbare feltene og resultatfeltene ("attributtene") innen en søkeklynge[49]. I tillegg definerer den en rekke egenskaper på hvordan feltene skal behandles; skal de kunne sorteres, hvilken datatype skal lagres her, hvilke leksikalske analyser foretas på de forskjellige feltene osv. Endringer i indeksprofilen vil igjen propagere endringer til de rette konfigurasjonsfilene. Ethvert dokument i FDS er plassert i én samling (collection), men det kan eksistere flere samlinger innen en søkeklynge (med samme indeksprofil).

Da vi skulle spesifisere indeksprofilen i vår prototyp måtte vi velge om vi ønsket at alle dokumentene skulle være i en samling eller om hver indeks som ble laget i databasen skulle generere opp en ny. Vi valgte det første. Først og fremst fordi FDS API'et ikke hadde noen mulighet for å manipulere samlinger, men også fordi det ville utelukke muligheten for å søke i dokumenter plassert i flere ulike indekser (samlinger i FDS). Vi hadde behov for et felt å lagre data i, samt informasjon som ville identifisere tuppelet i databasen. Den viktigste konfigureringen av FDS gjorde vi ved hjelp av indeksprofilen og vi lagde følgende:

```
<?xml version="1.0"?>
<!DOCTYPE index-profile SYSTEM "index-profile-2.0.dtd">
<index-profile name="FastIndex">
  <field-list>
    <field name="data" lemmas="yes" sort="yes"
         result="dynamic" />
    <field name="teaser" result="dynamic"
         result-element-name="data" />
    <field name="block" type="integer" index="no" />
    <field name="offset" type="integer" index="no" />
    <field name="indexoid" type="integer" index="yes" />
    <field name="database" type="string" index="yes" />
    <field name="host" type="string" index="yes" />
    <field name="rowoid" type="integer" index="no" />
  </field-list>

  <composite-field rank="yes" name="content"
         default="yes" lemmas="yes">
     <field-ref name="data" />
  </composite-field>
</index-profile>
```

Nedenfor følger en nærmere beskrivelse av de ulike feltene:

data

Dette er feltet for å lagre de faktiske data som det skal søkes i. Vi ønsker at data her skal lemmatiseres og kan sorteres hvis ønskelig. Dynamisk resultat betyr at det er mulighet for å hente ut et dynamisk dokumentsammendrag basert på søkeordene (”teaser” – se 3.1.8). Et statisk resultat ville derimot returnert all data i dokumentet. Når vi benytter FDS som indeksaksessmetode gjør vi søk i FDS, men henter opp ”orginaldata” fra databasen identifisert ved hjelp av de andre feltene i indeksprofilen.

indexOid, database og host

Disse feltene identifiserer unikt hvilken indeks som data i dokumentet kommer fra. Siden disse feltene blir oppgitt når en gjør søk fra databasen (for kun å søke i riktig indeks) må disse også indekseres slik at søk går fortere.

block og offset

Sammen med informasjon fra feltene over bestemmer disse unikt hvor på harddisken dokumentet befinner seg. Vi trenger denne informasjonen når vi skal hente opp data i databasen, men i og med at de aldri er aktuelle å søke på trenger de ikke å indekseres. Skulle en implementert CONTAINS operatoren som skissert i 7.2.1 måtte disse imidlertid også blitt indeksert.

rowOid

Dette feltet er OID'en til det indekserte dokumentet. Dette blir benyttet av contains_table – funksjonen (se 5.5) for å hente opp det riktige tuppelet etter søk.

content

I indeksprofilen er det også muligheter for å definere sammensatte felt. Denne grupperingen kan gis ulike parametere som vekting av de ulike feltene og default[50]. Fordi vi ønsket muligheten for default søk i datafeltet la vi det inn her med denne egenskapen.

5.4.2 Kommunikasjonslaget

FDS har to programmeringsgrensesnitt, API, for å kommunikasjon med søkemotoren utenfra; innholdsgrensesnitt og søkegrensesnitt. Innholdsgrensesnittet brukes for å manipulere dokumenter i søkemotoren med operasjoner som "legg til" og "slett". Søkegrensesnittet brukes for søkeoperasjoner. Begge grensesnittene er tilgjengelig i Java, C++ eller COM. Funksjonene vi skrev mot PostgreSQL måtte implementeres i C for både ha tilgang til PostgreSQL-kildekoden. Det var derfor naturlig at vi implementerte kommunikasjonslaget i FDS i C++ slik at disse kunne linkes inn i en delt objektfil og kalle hverandre samt lastes dynamisk inn i PostgreSQL. Skillet er likevel viktig siden vi nå beholder alt som har med PostgreSQL spesifikk kode i en modul og alt som har med FDS i en annen.

I kommunikasjonslaget har vi tre metoder som manipulerer og søker data:

insert_document()

Dens oppgave er å generere en dokumentid samt sette dette inn i FDS. Som nevnt i delkapittelet over har vi en rekke felt i indeksprofilen som til sammen unikt identifiserer data. Disse settes sammen til en dokumentid som blir benyttet i FDS (dette er imidlertid ikke garantert en unik global id, men dette vil i de aller fleste praktiske løsninger ikke være noe problem, og vi valgte å se bort fra det i vår prototyp):

```
docId = host/database/indexoid/block/offset
```

Koden blir forholdsvis enkel og benytter seg av FDS API'et for å sette inn dokumentet:

```
extern "C" char* insert_document(char * document, int block, int
      offset, int indexOid, char * database, char * dataHost) {

  try {
```

```
    icontent_manager_ptr cm(get_content_manager(HOST, PORT,
                                                COLLECTION));
    content_factory_ptr cfact(create_content_factory());
...
    string docId = getDocumentId(block, offset, indexOid,
                          database, dataHost);
    document_ptr doc(cfact->create_empty_document(docId));

    document_element_ptr data_elem(new string_element("data",
d));
    doc->add_element(data_elem);

...
    cm->add_content(doc.get());
...
```

do_fastscan()

Funksjonen mottar en fulltekstspørring og parametre som identifiserer hvilken indeks det er som brukes (slik at vi kan begrense søket til kun de dokumenter som finnes i indeksen). Den vil så bygge opp en sammensatt spørring bestående av søkeuttrykket og indeksidentifikasjon som FDS forstår. Funksjonen gjør spørringen mot FDS og leser ut resultatene og bygger opp en lenket liste med tuppelidentifikasjon (fysisk plassering på disk). Indeksaksessmetodene benytter seg videre av denne listen for å hente opp de riktige dokumentene (se `FastSearchResult` i 5.1 for beskrivelse av listen).

```
extern "C" FastSearchResult* _do_fastscan(char* searchKey,
     int indexOid, char * database, char * dataHost)
{
    FastSearchResult* fres = NULL;
...
    query_factory_ptr fact(create_query_factory());
    search_parameter_list params;
    params.push_back(fact->create_parameter(
                parameters::QUERY, cs_key+
                " AND database:"+cs_database +
                " AND host:"+ cs_datahost +
                " AND indexoid:"+ oid.str()).release());
    params.push_back(fact->create_parameter(
           parameters::TYPE, searchtype::ADVANCED).release());

    iquery_ptr query(fact->create_query(params));
    isearch_engine_ptr se(
        create_search_factory(cs_host, cs_port)->
              create_search_engine(keepalive, timeout));
    iquery_result_ptr result(se->search(query));
    fastresult = makeSearchResult(result.get(), "");
...
   return fastresult;
}
```

Metoden `makeSearchResult()` "oversetter" resultatet fra et FDS `iquery_result` til `FastSearchResult`.

do_fastdelete()

Som navnet tilsier har denne funksjonen i oppgave å slette et dokument fra FDS. Denne vil bli kalt fra PostgreSQL når et dokument skal fjernes, I likhet med `insert_dokument()` bygger den opp dokumentid'en og gir beskjed til søkemotoren om at dokumentet må slettes.

Det er imidlertid ikke behov for en oppdaterings funksjon for endringer i dokumentet. Måten dette blir løst på er at PostgreSQL markerer det gamle tuppelet som ugyldig og lager et nytt tuppel i stedet for. Det betyr at en på dette stadium vil ha to dokumenter i FDS som egentlig bare er to versjoner av det samme. Siden FDS ikke får beskjed om at et dokument er ugyldig, kan et søk resultere i retur av "slettede" dokumenter. Det ugyldige dokumentet når imidlertid ikke frem til brukeren fordi søkeresultatet går gjennom backend i databasen. PostgreSQL har oversikt over hvilke dokumenter som er slettet, men enda ikke fjernet, og vil ikke akseptere et ugyldig dokument som en del av resultatet. Tuppelet vil ikke bli slettet før en VACUUM operasjon. Når dette utføres, vil aksessmetoden få beskjed og kalle på `do_fastdelete()` som vil slette dokumentet fra søkemotoren.

5.5 Tabellfunksjoner

CONTAINS-operatoren har den svakhet at vi ikke har mulighet til å hente ut metainformasjon som for eksempel rangering fra søket. Resultatet av en spørring der en benytter CONTAINS operatoren blir automatisk sortert synkende på rangeringsverdi, men en har ikke mulighet for å begrense søket for eksempel til en rangeringsverdi høyere enn M eller til å få ut sammendraget, noe som ofte er ønskelig ved fulltekstsøk. Inspirert av MSSQL sin implementasjon av fulltekstsøk (se 4.3) lagde vi derfor en tabellfunksjon som kunne gi oss dette. En tabellfunksjon i PostgreSQL er en funksjon som ikke returnerer en enkelt verdi, men et eller flere tupler. Dermed kan dette sees som en relasjon og plasseres i FROM delen i en spørring. For å slippe å angi "tabelldefinisjonen" hver gang en bruker spørringen, kan en bestemme at funksjonen skal returnere en type. Vi valgte dette og definerte følgende type:

```
CREATE TYPE contains_table_result AS
(oid Oid, rank int4, teaser text);
```

Vi kunne da deklarere en tabellfunksjon som returnerte tupler bestående av OID'en til tuppelet, rangeringsverdien og et dokumentsammendrag som vist nedenfor:

```
CREATE OR REPLACE FUNCTION contains_table(text, text, text)
RETURNS setof contains_table_result
LANGUAGE 'C' STRICT AS 'fast_index.so';
```

Implementeringen av denne funksjonen bestod egentlig bare i at spørringen ble videresent til FDS og ved hjelp av block og offset parametrene fra søkeresultatet gjenfant vi OID'en til det aktuelle tuppelet. Rank og teaser hentet vi også ut fra søkeresultatet og kunne bygge og returnere et sett av tupler.

De tre parametrene den mottar er henholdsvis tabellnavn, attributt og søkeuttrykk. Siden vi her ikke benyttet oss av aksessmetodeimplementasjonen

måtte vi altså motta som parametre hvilket attributt det faktisk skulle spørres over. Dette måtte selvfølgelig først være indeksert ved hjelp av vår indeksaksessmetode ellers ville jo ikke dokumentene finnes i FDS. Ett eksempel på søk kan være:

```
SELECT teaser
FROM CONTAINS_TABLE('philosopher', 'desc', '"Cogito, ergo sum"')
ORDER BY rank ASC
LIMIT 10;
```

I eksempelet her får vi sammendraget fra de ti filosofene som dårligste passer med søkeuttrykket. Siden vi henter ut OID'en kan vi også lage mer komplekse spørringer for å hente ut annen relasjonsinformasjon i tillegg til det tabellfunksjonen tilbyr, for eksempel:

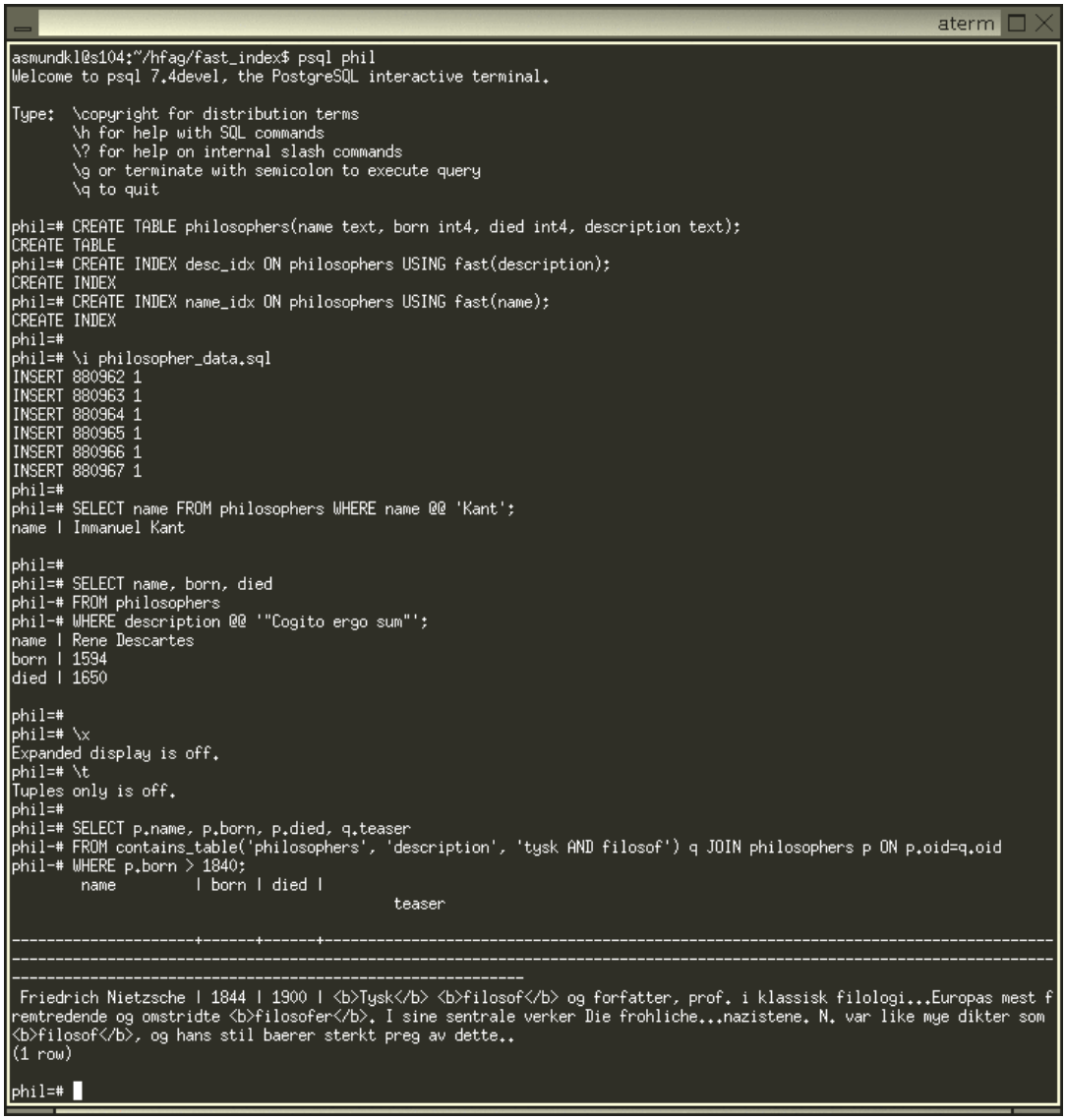
```
SELECT p.name, p.born, s.teaser
FROM philosopher p JOIN
      CONTAINS_TABLE('philosopher', 'desc',
            '"The meaning of life" AND 42') s ON s.Oid = p.Oid
WHERE p.born > 1850;
```

En viktig mangel i vår implementasjon av denne tabellfunksjonen er at vi ikke tar høyde for om tuplene er slettet eller ikke. Finnes det derfor et dokument i FDS som returneres ved en spørring vil vi hente opp dette og ikke filtrere bort de tupler som er markert som ugyldige. Når en joiner inn den indekserte relasjonen som i eksemplet over, unngår et dette problemet. Det er ikke en umulighet å utvide funksjonen til å ta hensyn til dette, men vi ønsket først å fremst å vise konseptet og la ikke videre arbeid inn i denne funksjonen i prototypen.

5.6 Kjøreeksempel

Figur 11 på neste side viser bruk av indeksaksessmetoden. Først lager jeg en tabell som skal inneholde data om filosofer (jeg hentet ned beskrivelse av utvalgte filosofer fra Caplex). Deretter lager jeg to fulltekstindekser over henholdsvis navn og beskrivelse.

Første spørring viser hvordan navneindeksen kan hjelpe på enkelt søk selv på kortere tekster for navn. Dette kan være ønskelig hvis en ikke er sikker på hvordan fornavnet skrives, om personen mellomnavn, om dette skrevet ut eller forkortet osv. Den andre spørringen benytter seg av fulltekstindeksen over beskrivelsen og frasesøk for å finne ut hvem som sa "Cogito, ergo sum". Den siste spørringen viser et eksempel på en sammensatt spørring med bruk av *contains_table()* funksjonen for å hente ut et dynamisk dokumentsammendrag med en begrensning på ikke indeksert data om når filosofen er født.



Figur 11: Kjøreeksempel på bruk av indeksaksessmetoden.

KAPITTEL 6: ANALYSE OG YTELSESTEST

Jeg har nå presentert en prototyp av en integrering av et IR-system og en relasjonsdatabase. Jeg vil i dette kapittelet drøfte hvilke sterke og svake sider dette systemet har. Først vil jeg diskuterer hvorvidt vi har fått til en fullstendig integrasjon. Deretter vil jeg presentere en ytelsestest vi utførte for å se på det samlede systemet.

6.1 En fullstendig integrasjon?

En fullstendig integrasjon bør inneholde [6]:

1) Mulighet for lagring, indeksering, oppdatering og søk i både strukturerte og ustrukturerte data.
2) Støtte for transaksjoner.
3) Et spørrespråk for IR-søk, integrert i spørrespråket til databasen.
4) Støtte for samtidig oppdatering og søk.
5) Skalerbar dokumentindeksering og søk.

Nedenfor drøfter jeg hvordan prototypen vår tilfredstiller disse kravene.

6.1.1 Lagring, indeksering, oppdatering og søk

Vår implementasjon tar utgangspunkt i PostgreSQL og gjør ingen endringer i den eksisterende funksjonaliteten slik at alle disse punktene er ivaretatt for strukturerte data lagret i databasen.

Når det gjelder tekstdata gjennomgår jeg hver av punktene enkeltvis. Data blir fremdeles lagret i PostgreSQL. Dermed er første punktet oppfylt. Samtidig med lagring vil aksessmetoden sørge for oversendelse til FDS slik at dokumentet blir indeksert. På den måten gjøres indekseringen tilsynelatende i selve databasen uten at vi trenger å implementere denne teknologien fra bunnen av. Oppdatering i PostgreSQL fungerer som tidligere beskrevet ved at gamle data blir markert som ugyldige og at nye blir satt inn. Vi har implementert mulighet for sletting av dokumenter, og siden PostgreSQL holder kontroll med gyldigheten av tupler, har også prototypen støtte for oppdatering.

Det er imidlertid visse begrensninger knyttet ved sletting av tabeller og indekser. Vi har sett at dokumenter først blir fjernet fra den eksterne indeksen når `VACUUM` kjøres og ikke når en bruker `DELETE` kommandoen for å slette data fra en relasjon. Hvis en derimot gjør operasjoner som `DROP TABLE, DROP INDEX` eller `TRUNCATE` får ikke aksessmetoden beskjed om dette. Dette fordi PostgreSQL går ut fra den antagelsen alle indeksaksessmetoder lagrer sine data ”inni” databasen og at den derfor kun frigjør de områder indeksen opptar på disken som systemet har kontroll over. Dermed kan en risikere at det finnes en rekke dokumenter i FDS som ikke lenger befinner seg i databasen. For å unngå slike situasjoner ved bruk av vår implementasjon må en først slette alle tuplene for så å bruke `VACUUM` for å sørge for at denne informasjonen propagerer ut til søkemotoren før en gjør `DROPT TABLE` eller `TRUNCATE`. Ved `DROP INDEX` er det litt verre. En kan ikke kreve at brukeren må slette alle radene i tabellen før

han/hun kan slette indeksen. For å rydde opp med vår implementasjon måtte en i så fall kopiere all data over i en annen tabell for så slette all data slik at de blir slettet i FDS og så kopiere dem tilbake etter at en har slettet indeksen. Skulle vi støttet disse operasjonene slik at de fungerte etter hensikten måtte vi utvidet grensesnittet i PostgreSQL slik at aksessmetoder også fikk informasjon om sletting av indeksen.

Søk foregår ved vanlige SQL kommandoer ved at vi har innført en ny operator og en ny funksjon. Selve søkeuttrykket uttrykkes i FDS sitt spørrespråk. Dette diskuteres nærmere under 6.1.3.

Dermed er både lagring, indeksering, oppdatering og søk mulig i vår prototyp både i strukturerte og ustrukturerte data via SQL kommandoer.

6.1.2 Transaksjonsstøtte

Som nevnt i avsnitt 2.3 er transaksjonsstøtte en av de essensielle egenskapene i et moderne databasesystem. Dette er nødvendig både for at enkeltbrukere skal være sikret at transaksjonene overholder ACID-egenskapene atomisitet, konsistens, varighet, og for at samtidig bruk av databasen ikke kan føre til inkonsistens/feil i data (isolasjon).

Behovet for transaksjonsstøtte vokste frem i databaseverdenen der det typiske er mange korte oppdateringer/endringer av flere samtidige brukere. Til forskjell fra dette, har Informasjonsgjenfinningssystemer tradisjonelt bygget på antagelsen om få oppdateringer og mange søk/oppslag. Dermed har ikke transaksjonsbegrepet vært like essensielt for informasjonsgjenfinningssystemer og er vanligvis ikke tilbudt i disse systemene. Dette har heller ikke vært særlig fokusert på i forskning [5]. IR-systemer krever dessuten mye maskinressurser for å løse oppgavene og er derfor ofte spredt ut over flere maskiner, noe som gjør transaksjonsstøtte betraktelig mer komplisert og arbeidskrevende.

Spørringer som kjøres i vår prototyp går gjennom PostgreSQL som i utgangspunktet støtter alle ACID-kravene. Men siden vi benytter oss av et eksternt system til indeksering er dette ikke lenger garantert. Nedenfor vil jeg gå gjennom de enkelte ACID-kravene og se hvilke problemer som kan oppstå.

Atomisitet

Atomisitet er i korthet et ”alt eller intet”-krav. Hvis en transaksjon setter inn et tuppel, vil dette bli satt inn i FDS for indeksering i det innsettingen gjøres selv om ikke transaksjonen har fullført. PostgreSQLs transaksjonshåndtering vil da sørge for at kun den pågående transaksjonen vil se dette og sørge for at det blir markert ugyldig hvis denne transaksjonen måtte avbryte. Ved neste VACUUM vil dette også slettes i FDS. Hvis det hadde skjedd en feil ved oversendelsen til FDS, ville aksessmetoden feilet og hele transaksjonen ville bli avbrutt.

Hvis imidlertid FDS feiler i å indeksere dokumentet, oppstår et problem fordi kallet til FDS om indeksering er asynkront og det gis ikke tilbakemelding om indekseringen har gått bra eller ikke, kun om ordren ble mottatt. Hvis

søkemotoren feiler ved indeksering av dokumentet skulle transaksjonen bli avbrutt, men dette blir ikke oppdaget så lenge det ikke finnes en form for tilbakekallgrensesnitt. Nyere versjoner av FDS tilbyr et grensesnitt som kan gi tilbakekall når et sett av dokumenter er ferdig indeksert og garantert skrevet til disk, og når denne indeksen er "live". Dermed kunne en muligens implementere noe som hadde garantert atomisitet. Likevel hadde det vært vanskelig siden transaksjoner måtte henge til alle dokumentene var ferdig indeksert, eller det måtte være mulighet for å omgjøre transaksjoner som allerede hadde fullført rekursivt. Atomisitet er med andre ord ikke garantert.

Dette kunne kanskje være løsbart ved å implementere et synkront metodekall for innsetting i FDS. Dette er imidlertid ikke ønskelig, da FDS indekserer dokumenter i grupper for å parallellisere prosessen. Å sette inn et dokument da vil kunne ta uforholdsmessig lang tid, og transaksjonene tilsvarende lang tid, noe som er uakseptabelt. Dessuten hadde det minsket samtidighet i databasen fordi pågående transaksjoner holder låser.

Konsistens

Konsistenskravet går normalt på at etter hver fullført transaksjon, skal databaserestriksjonene være overholdt i form av fremmednøkler og andre regler. Vår aksessmetode tilbyr ikke funksjonalitet for å overholde restriksjoner om unikhet i attributter så dette vil ikke by på problemer.

Et annet konsistenskrav til en transaksjon kan være at etter fullført transaksjon skal alle indekser være riktig oppdatert. I vårt tilfelle har vi ingen garanti for at et dokument er ferdig indeksert i FDS etter fullført transaksjon, tvert imot vil det som oftest ikke være tilfellet. Med andre ord vil det midlertidig oppstå en inkonsistens i form av tidsforskyvning mellom de dokumenter som faktisk befinner seg i databasen og de dokumenter som er søkbare.

Et mer alvorlig problem kan oppstå hvis FDS ikke får fullført indekseringen av dokumentet. Det kan skje en feil under indeksering eller systemet kan gå ned før indeksen er skrevet til disk. Da er det ingen mulighet for å støtte konsistens med mindre en har garanti fra den eksterne indeksen om gjenoppretting og varighet.

Isolasjon

Isolasjon betyr at transaksjonen skal utføres uten innvirkning av andre transaksjoner. Serialiserbarhet betyr at selv om to transaksjoner overlapper i tid, skal de ha samme effekt som om de ble utført etter hverandre. Ved bruk av MVCC i databasen er både isolasjon og serialiserbarhet garantert. En oppdatering i databasen vil medføre at det gamle elementet markeres ugyldig og et nytt element legges til i indeksen. Hvis det da er en pågående transaksjon som er interessert i det gamle tuppelet vil det ikke slettes fra indeksen selv om den første transaksjonen foretar en VACUUM. Så lenge en går via databasen og antar at atomisitet er oppfylt er derfor dette garantert.

Varighet

Databasen vil sørge for at data vil være varig i databasen. Det vi derimot ikke har noen garanti for er at den eksterne indeksen har denne egenskapen. En gjenoppretting i databasen kan medføre at tupler som var blitt satt inn i indeksen ikke blir gjenopprettet i databasen hvis transaksjonen ikke fullførte. For å garantere samsvar i data ved gjenoppretting måtte en i så fall ha reindeksert hele relasjonen. Dette er imidlertid uakseptabelt, da store datamengder kan føre til at en full reindeksering kan ta flere dager. Alternativt kan en tenke seg at FDS selv garanterer varighet, men da dette systemet er distribuert over flere noder, og versjonsoppdatering av indekser ikke er synkronisert, vil en naiv implementasjon av dette sannsynligvis redusere ytelsen betraktelig.

Konklusjon

Den eneste grunnen til at prototypen til en viss grad overholder ACID-kravene er i den grad databasen som tar seg av det. Gyldighetshåndteringen av elementer i PostgreSQL skjer på block/offset nivå i buffer manageren og for at en full integrasjon skulle kunne finne sted, der data i indeksen også kunne være gyldig uten å måtte gå via databasen, måtte det vært en tilsvarende modul i FDS som PostgreSQL kunne samarbeidet med.

Vår tabellfunksjon contains_table benytter seg ikke av aksessmetodegrensesnittet, men går direkte til FDS. Ugyldige tupler vil derfor bli returnert som gyldige inntil en VACUUM operasjon utføres. Dette kan bety at ingen av ACID-egenskapene kan garanteres. Hvis en implementerer gyldighetssjekk av tupler i funksjonen vil den ha samme begrensninger som CONTAINS operatoren.

Siden IR-systemer og relasjonsdatabaser er såpass forskjellig er det ikke uventet at overføring av tradisjonell transaksjonsstøtte til IR-systemer er problematisk. For eksempel er det problematisk at IR-operasjoner som innsetting og oppdatering av dokumenter kan gå over lang tid sammenliknet med vanlige databasetransaksjoner. I et integrert system kan det derfor bli nødvendig å inngå kompromisser eller å operere med to forskjellige transaksjonsmodeller avhengig av hvilke operasjoner som utføres og på hvilke typer data. Håkon Clausen har i sin oppgave drøftet disse problemene grundigere og skissert mulige løsninger[18].

6.1.3 Spørrespråk

Spørrespråket vil være en viktig del av et fullstendig integrert system. For å vurdere spørrespråket, vil jeg ta for meg hvordan spørringer i SQL ble utvidet for å støtte funksjonaliteten, og hvordan selve fulltekstspørringen utformes.

Spørrespråk i SQL

SQL/MM belager seg på at SQL-99 er implementert og derfor at datatyper kan tilknyttes funksjoner. Et søk i henhold til standarden vil være på formen:

```
SELECT <SELECT CLAUSE> FROM table
WHERE text_attrib.Contains(FT_Pattern);
```

I tilfeller der en ikke kan tilby denne formen (hvis en ikke støtter at datatyper kan ha metoder) kan en, for å være konform med standarden implementere det som en funksjon:

```
SELECT <SELECT CLAUSE> FROM table
WHERE Contains(text_attrib, FT_Pattern);
```

I vår prototyp støtter vi ingen av de foreslåtte løsningene fordi ingen av dem hadde vært mulig uten å gjøre endring i kildekoden til PostgreSQL. Vi implementerte i stedet CONTAINS som en operator – igjen for å enkelt integrere den med indeksaksessmetodegrensesnittet til databasen:

```
SELECT <SELECT CLAUSE> FROM table
WHERE text_attrib CONTAINS FDS_query;
```

Vi implementerte også en funksjon, men da som tabellfunksjon. Dette for å kunne hente ut metadata som rangeringsverdien (som også MSSQL gjør det 4.3).

```
SELECT <SELECT CLAUSE>
FROM contains_table(table, text_attrib, FDS_query);
```

I SQL/MM gjøres dette ved at en i SELECT delen henter ut verdien basert på en spørring. Eksempelvis:

```
SELECT text_attrib.Score(FT_Pattern) FROM table
WHERE text_attrib.Contains(FT_Pattern);
```

Her behøver imidlertid ikke søkeuttrykkene være de samme, hvilket betyr ekstraarbeid i de tilfeller dette er ønskelig, men samtidig økt fleksibilitet. SQL/MM tilbyr ikke mulighet for å hente ut dokumentsammendrag, verken dynamisk eller statisk – hvilket trolig er ønskelig. Alle disse tilnærmingene er forholdsvis like og underliggende struktur vil i mange tilfeller avgjøre hvilken av dem en ønsker å benytte seg av.

Fulltekstspørrespråk

Når det gjelder fulltekstspørrespråk finnes det ingen utbredt standard meg bekjent. FDS som moderne søkemotor tilbyr et veldig enkelt søkespråk bestående av nøsting av boolske operatorer med ord og fraser. Dette er tilsvarende i SQL/MM.

FDS har kraftig underliggende funksjonalitet som brukeren ikke blir eksponert for. *FT_Pattern* i SQL/MM er derimot veldig rettet mot at brukeren skal kunne spesifisere, per spørring, hvilken type leksikalske operasjoner som skal utføres osv. I FDS er dette funksjonalitet som gjerne settes i konfigurasjonsfiler (som indeksprofilen) og som ikke kan endres for hvert søk. Eksempelvis kan en i SQL/MM eksplisitt angi ordnærhet i nærhetssøk, mens i den versjonen av FDS vi benyttet oss av vil ordnærhet kun gi utslag på rangeringen av dokumenter. En ser altså at det finnes funksjonalitet i FDS som behandler dette, men det er ikke

mulig å angi for en bruker. Ved fullstendig integrasjon burde dette bli eksponert hos brukeren. I så fall måtte en utvidet spørrespråket i FDS.

Som allerede påpekt, mener jeg at heller ikke SQL/MM er et fullstendig språk i et integrert system. Det gir ikke mulighet for mer avansert IR-funksjonalitet som generering av dokumentsammendrag, "finn lignende", kategorisering og ulik vekting av søkeord som alle er tilgjengelig i FDS (det siste kan riktignok simuleres ved at en setter sammen ulike Score() ledd i SQL/MM som en sorterer etter). Dette kunne vi implementert i vår prototyp ved egne funksjoner for f.eks. finn lignende.

6.1.4 Samtidig oppdatering og søk.

Både PostgreSQL og FDS tilbyr høy grad av samtidighet hver for seg og vår prototyp legger ingen begrensninger på noen av dem i denne sammenheng. Vi må derfor kunne si at det integrerte systemet tilbyr høy grad av samtidighet både ved oppdateringer og søk.

6.1.5 Skalerbar dokumentindeksering og søk.

FDS er et meget skalerbart system. Både med hensyn på mengde data og spørrefrekvens. Det er designet for å kunne spres utover mengder av noder. PostgreSQL er derimot, som de fleste databasesystemer, mindre skalerbart med tanke på spredning utover flere datamaskiner. Dette er først og fremst på grunn av problemer knyttet til transaksjonshåndtering. I vår prototyp er tanken at PostgreSQL vil være der data er lagret. Når det gjelder lagring og dokumentindeksering er oppgaven som PostgreSQL skal gjøre minimal og indekseringsarbeidet vil foregå i FDS. Jeg vil derfor hevde at dokumentindeksering i det samlete systemet i høyeste grad er skalerbart.

Ved søk vil også all IR-funktionallitet gjøres av FDS, men databasen vil ha flere oppgaver enn ved innsetting - spesielt hvis en gjør sammensatte søk. Potensielt vil PostgreSQL her være en flaskehals med hensyn på skalerbarhet. Hvis en imidlertid antar at prototypen blir brukt i systemer som har en oppførsel som først og fremst tilsvarer IR-systemer med mye les og lite skriv finnes det gode replikasjonsmuligheter for PostgreSQL som vil gjøre skalerbarheten større også på databasesiden.

6.2 Ytelsestest

For å verifisere antagelsen om at vår prototyp kunne tilføre fulltekstsøkefunksjonalitet ikke bare bedre, men også raskere enn de andre (begrensede) muligheter som finnes i PostgreSQL, gjorde vi en enkel test på søk. Vi laget tre testdatabaser for henholdsvis å teste tsearch, LIKE og vår prototyp mot hverandre og målte søketiden ved forskjellig datamengde.

For databaser defineres batch-skalering som det å kjøre samme spørring mot en database hvis menge data vokser med en bestemt faktor. Denne er lineær hvis databasestørrelsen og tiden spørringen tar vokser med samme faktor mens maskinresurser holdes konstant [51]. Ved bruk av indeksaksessmetoder forventes en vekst som er langt under lineær (trestrukturer har typisk en vekst i størrelsesordenen log(n)). Inverterte filer (se 3.2) vil også normalt ha en vekst

som er sublineær. Dette fordi forekomsten av unike ord stort sett er langt mindre enn det totale antall ord.

6.2.1 Testdatabasene

Nedenfor følger en presentasjon av testdatabasene vi laget:

Tsearch

I tsearch databasen (se 4.4.2 for beskrivelse av tsearch) hadde vi følgende tabell:

```
CREATE TABLE data_table (
        data text, -- dokumentene
        dataidx  txtidx – ”ordene” i dokumentet
);
```

Ved innsetting av dokumenter setter en kun data inn i *data*-attributtet. Ved hjelp av følgende trigger vil dokumentet bli gjort om fra text til txtidx og satt inn i *dataidx*-attributtet:

```
CREATE TRIGGER txtidxupdate BEFORE UPDATE OR INSERT ON data_table
FOR EACH ROW EXECUTE PROCEDURE tsearch(dataidx, data);
```

Dermed er dokumentet klargjort for indeksering:

```
CREATE INDEX t_idx ON data_table USING gist(dataidx);
```

Eksempelspørringen jeg gjorde ser da slik ut:

```
SELECT count(*) FROM data_table WHERE dataidx ## 'hans&grete';
```

I våre testdata resulterte dette i 34 treff.

LIKE

For å sammenligne hastigheten med sekvensielt søk laget jeg en database hvor jeg testet spørringen bare ved å bruke LIKE, dvs jeg benyttet meg av PostgreSQL sin ILIKE operator (ikke en del av SQL standarden [17]) for å ikke ta hensyn til store og små bokstaver. Relasjonen og eksempelspørring ser da slik ut:

```
CREATE TABLE data_table (data text);
SELECT count(*) FROM data_table
        WHERE data ILIKE '% hans %' AND data ILIKE '% grete %';
```

ILIKE er ikke like uttrykksfult som tsearch og FDS. Vi valgte altså ut de dokumenter som inneholdt nøkkelordene med et mellomrom både foran og bak, dette for at vi ikke skulle få for mange treff på andre ord (eksempelvis ”hanske” og ”sankthans”), men dette utelukker også ord som står rett ved andre tegn som komma og punktum. Søket returnerte derfor 29 treff.

Vår prototyp

For å teste vår prototyp brukte vi operatoren CONTAINS ved søk. Vi trengte ikke metainformasjon fra søket som contains_table funksjonen kunne gitt oss og

ville sammenligne samme strukturen i spørringen som de andre. Relasjon, indeks og eksempelspørring ser slik ut:

```
CREATE TABLE data_table (data text);
CREATE INDEX fast_index ON data_table USING fast (data);
SELECT count(*) FROM data_table WHERE data @@ 'hans and grete';
```

Spørringen resulterte i 32 treff. Årsaken til at FDS returnerte to færre dokumenter enn tserach skyldes forskjellig lingvistiske analyser i dokumentet og ved søk. Eksempelvis baserer tsearch seg på stemming (mens FDS har en mer avansert lemmatiseringsteknikk) og den skriver derfor om spørringen til ord i sin rotform som eksempelet under viser:

```
akl=# select 'hans&grete'::mquery_txt;
   mquery_txt
-----------------
 'han' & 'grete'
```

De to dokumentene som tserach returnerte ”for mye” inneholdt bare ”han” og ikke ”hans”.

6.2.2 Datasettet

I mangel av en god ”standard” tekst å benytte i testen, valgte vi å bruke crawleren i FDS til å hente data fra odin.dep.no - en informasjonsside fra regjeringen og departementene. Vi anså dette for å være en god kilde til tilfeldig menneskegenerert tekstdata. Selv om dette er en ganske omfattende side, ble det bare ca 30 MB råtekstdata. Nedenfor vises noen nøkkeltall fra testdataene:

Antall dokumenter	7170
Antall ord	4083837
Antall unike ord	251233

Tabell 3 Spesifikasjon av testdata

For å hente dataene inn i databasen forandret vi indeksprofilen til at datafeltet skulle være statisk i stedet for dynamisk (se 5.4.1 for beskrivelse av indeksprofilen) og dermed lagret FDS en kopi av nettsidene. Deretter laget vi en ny variant av contains_table som vi kalte host_search som hentet ut all data fra FDS direkte inn i PostgreSQL:

```
INSERT INTO data_table
SELECT data FROM host_search('odin', 'data');
```

For å teste skalerbarheten på store mengder data valgte vi å laste dette datasettet opp til 30 ganger. På den måten fikk vi 870 MB med tekstdata. Ideelt skulle vi hatt en større datamengde slik at vi slapp å gjøre det, men av tidsårsaker la vi ikke mer arbeid i å skaffe det.

6.2.3 Gjennomføring av testen

Testen ble utført på en maskin med følgende spesifikasjon:

CPU	2 x Pentium Xeon 2.2Ghz
Minne	5,6 GB
Disk	UltraWide2 SCSI, RAID
OS	ReadHat Linux 8.0 2.4.18-14bigmem
FDS	FSD 3.2.2
PostgreSQL	PostgreSQL 7.4.3

Tabell 4 Testmaskinspesifikasjon

Selve testen ble gjennomført ved at vi lastet datasettet 30 ganger kumulativt, og mellom hver gang utførte vi de respektive spørringene angitt over. Ved hjelp av Unix verktøyet *time* målte vi tiden det tok å gjøre selve søket. Vi tok ikke tiden det tok å laste inn data, selv om dette er et interessant kriterium. Hovedgrunnen til at dette ikke var mulig, er at vi ikke kunne vite når dokumentene er ferdigindeksert i FDS. For å være garantert dette la vi inn noen ekstra minutter venting etter hver gang vi lastet inn data i FDS.

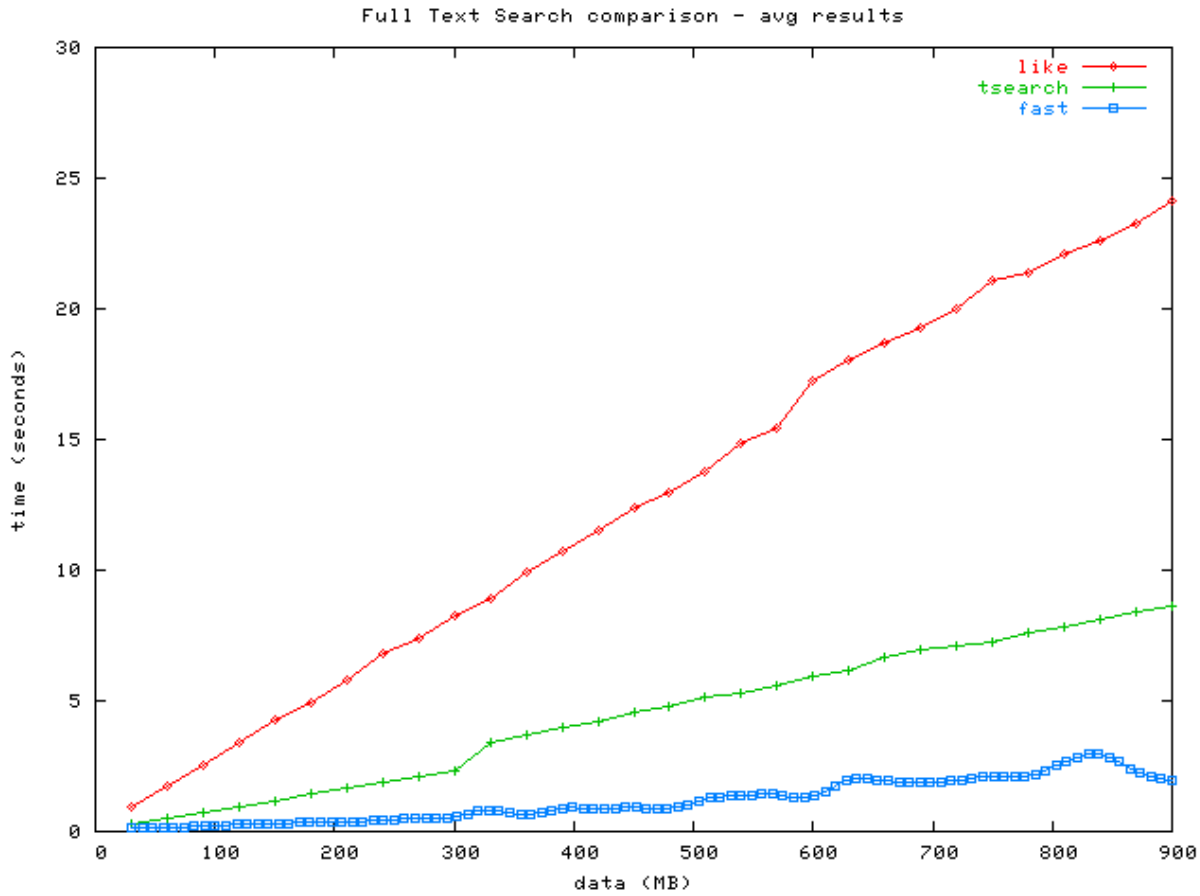
Vi utførte teste ved hjelp av et lite shell-script som lastet inn data og utførte spørringen for hver iterasjon over de tre testdatabasene. Nedenfor vises det viktigste:

```
for ((i=1; i<30; i++)); do
   psql -q -n -o /dev/null -c "\i data.txt" $1
   if [ "$1" == "fast" ]; then
      sleep 10m
   fi
   time=$(./time -f "%e" 2>&1 psql -q -n -a -c
      "\i $1/search.sql" $1 >> "result-output-$1-$2.txt")
Done
```

En total runde med å laste testdata og gjøre spørringen 30 ganger på alle testbasene tok omlag 13 timer. Vi gjennomførte dette 10 ganger for å unngå feildata på grunn av innvirkning fra andre faktorer som last fra andre programmer akkurat når en aktuell spørring ble kjørt.

6.2.4 Resultat og diskusjon

Figur 12 nedenfor viser gjennomsnittstiden for hvert søk etter hvert som datasettet økte. En ser at alle tre metodene hadde en mer eller mindre lineær økning, men med forskjellig stigningstall.



Figur 12: Resultat av ytelsestest

At bruk av ILIKE hadde en slik oppførsel var ingen overraskelse. Det er å forvente at sekvensielt søk over dobbelt så mye data tar dobbelt så lang tid noe som er en typisk batch-skalerbar transaksjon, og som kunne vært skalert ved å spre data over flere disker. Videre ser en, som forventet, at både tsearch og vår indeksaksessmetode, fast, hadde en betydelig bedre ytelse med fast over dobbelt så rask som tsearch igjen. Vi opplevde også noen ekstremverdier, men de kom ved forskjellige tidspunkt og skyldes antakeligvis andre programmer som også opptok maskinresurser.

Det er verdt å merke seg at denne testen kun gir en indikasjon på ytelsen et slikt sammensatt system kan ha. For det første burde vi ideelt sett hatt et større datasett fordi forskjellene i systemene da antakeligvis vil vise seg tydeligere (mye av FDS sin styrke vises først når det er snakk om store datamengder) og fordi vi hadde sluppet å laste samme datasettet flere ganger. Det siste medfører bl.a. at det ikke kommer nye nøkkelord inn i dokumentmengden, men også at antall dokumenter som gir treff på et nøkkelord vokser lineært med datamengden som lastes.

Videre er det ikke helt ”rettferdig” å sammenligne FDS med de andre, da den er flertrådet og kan utnytte begge CPU’ene på testmaskinen, mens i PostgreSQL kjøres en transaksjon kun på en CPU. Samtidig er dette også noe av styrken ved

bruk av ekstern indeks. FDS kunne vært skalert utover flere noder, noe som bare ville forsterket denne effekten ved større mengder data og spørringer.

Testen hjalp oss ikke bare til å vise styrke ved vår implementasjon, men også til å finne svakheter. Eksempelvis overføres ganske mye data fra FDS til PostgreSQL ved søk (plasseringen av tuppelet på disk samt et dokumentsammendrag). Siden antall treff øker lineært med mengden data (vi lastet samme datasett flere ganger) vil tiden det tar å overføre data vokse tilsvarende. Selve søket i FDS med å finne ut hvor mange dokumenter som traff var veldig rakst og søketiden beholdt seg mer eller mindre konstant, selv med større mengder data. Det som altså tok lengre tid var både det å hente opp alle treffene fra FDS, generere et dynamisk dokumentsammendrag for alle dokumentene og å overføre disse dataene. FDS er, som de fleste andre søkemotorer, optimalisert for å gjøre et raskt søk og presentere de første N treffene. Det er sjelden en ønsker å få frem alle treffene på en gang. For å få et optimalt system kunne vi utvidet PostgreSQL til å skjønne at bruk av `LIMIT` og `OFFSET` i en SQL spørring kunne blitt videresendt til FDS slik at kun det antall (aktuelle) dokumenter som var ønskelig ble hentet ut. Det er riktignok en del problemer forbundet med det. Spørreparseren kan ikke automatisk anta at en begrensning på antall treff som skal vises gjelder for fulltekstsøket. For eksempel ved sammensatte spørringer kan det være andre betingelser som gjør hovedbegrensningen, eller en annen sortering som gjør at det ikke er de beste FDS treffene som skal ut. En måte å omgå dette på er å tilby denne formen for begrensning i *contains_table()* funksjonen. Igjen er problemet her at FDS ikke har oversikt over hvilke dokumenter som er gyldige i databasen så hvis en spør etter de ti beste dokumentene fra FDS kunne en risikert at brukeren bare blir presentert med noen få, kanskje ingen dokumenter, enda det finnes flere som gir treff i FDS.

For å minske overheadet med overføring av data fra FDS til PostgreSQL kunne en i eksempelspørringen sett kun på antallet dokumenter som gir treff. I så fall ville vår indeksaksessmetode hatt overlegent bedre ytelse, men en måtte forutsette at FDS hadde kunnskap om gyldigheten av dokumenter. Dette diskuterer jeg nærmere i 7.2.4.

Denne testen er ikke noen ”benchmark” over implementasjonen vår. Som nevnt så er datagrunnlaget for dårlig (lite) og det burde vært generert flere tilfeldige spørringer i stedet for bare vår enkle ene. For å vise styrken til FDS ville det også vært mer naturlig å hatt et høyere trykk med flere spørringer. Testen hjalp oss likevel til å se begrensninger ved implementasjonen og finne flaskehalser.

KAPITTEL 7: AUTOMATISK UTFLATING OG VIDERE UTVIDELSER

7.1 Automatisk utflating av data fra en relasjonsdatabase til en søkemotor.

Uthenting av strukturerte data fra en database for å gjøre det indekserbart i en søkemotor er en omfattende prosess. Dette innebærer å "flate ut" strukturerte data slik at det kan indekseres som et dokument. Normalt er dette en manuell oppgave som kan være tidkrevende. En må sette seg inn i databasen, og skrive kode som kan hente ut ønsket informasjon. Å automatisere, i hvert fall deler av, en slik prosess vil kunne gjøre en integrering enklere. Jeg stilte derfor spørsmålet: Er det mulig å benytte seg av systemtabellene for å utføre en form for automatisk utflating / trekke ut strukturert data og gjøre det om til semistrukturert data f.eks. i form av et XML dokument?

I PostgreSQL ligger det aller meste av systemdata i systemtabeller som også er tilgjengelig for en bruker. Informasjon om fremmednøkler, attributter og tabeller gjør at en ved å lese disse kan bygge seg opp et bilde om hvordan databasen er uten å nødvendigvis å ha tilgang til selve skjema definisjonen i SQL (CREATE TABLE spørringer osv.).

Jeg forutsetter at ved en eventuell utflating vil en alltid ta utgangspunkt i en bestemt relasjon, som jeg vil referere til som baserelasjonen. Eksempelvis kan en ønske å hente ut innholdet i en produktdatabase. Da kan en tenke seg at produktrelasjonen vil være baserelasjonen, og når en henter ut data i form av XML, vil produkt ligge som øverste nivå. I andre sammenhenger er kanskje fokus på produsentene i den samme databasen, og da vil data om produsentene være baserelasjonen. Da vil også data som hentes ut ha en annen struktur.

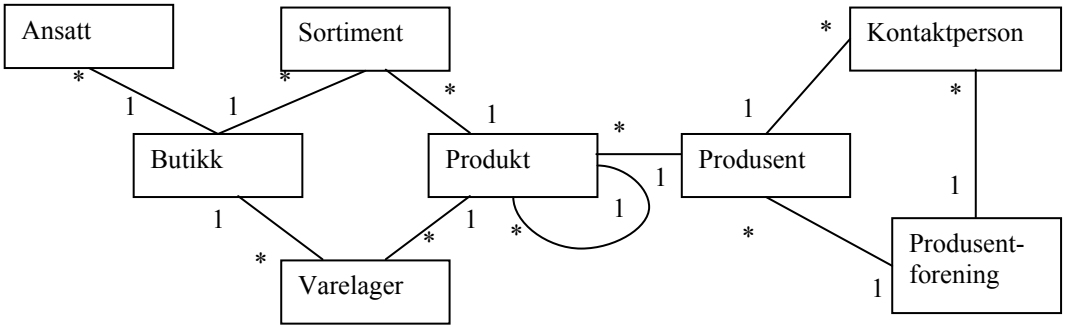
For å finne ut hvordan en skal trekke ut data må en gjøre en analyse av relasjonsskjemaet. For å forenkle en slik analyse kan en se for seg relasjonsskjemaet som en rettet graf, der enhver relasjon vil være en node, og en fremmednøkkel fra A til B vil fremstå som en rettet kant fra A til B. I en database er det imidlertid ingenting som tilsier at en slik fremmednøkkel kun leses en vei. Selv om det strengt tatt kun er én relasjon som vil vite om, og må overholde en slik fremmednøkkel (A i dette tilfellet) er det ingenting som tilsier at en ikke også kan gå fra B til A. Eksempelvis kan en for en gitt produsent enkelt finne ut hvilke produkter den produserer. I og med at en kan gå begge veier i databasen kan en derfor også se på relasjonsskjemaet som en urettet graf. Årsaken til at jeg ønsker å se på relasjonsskjemaet som en graf er ønsket om å abstrahere opp strukturen i databasen slik at det blir lettere å se hvordan jeg kan hente ut de data jeg ønsker.

7.1.1 Eksempeldatabasen

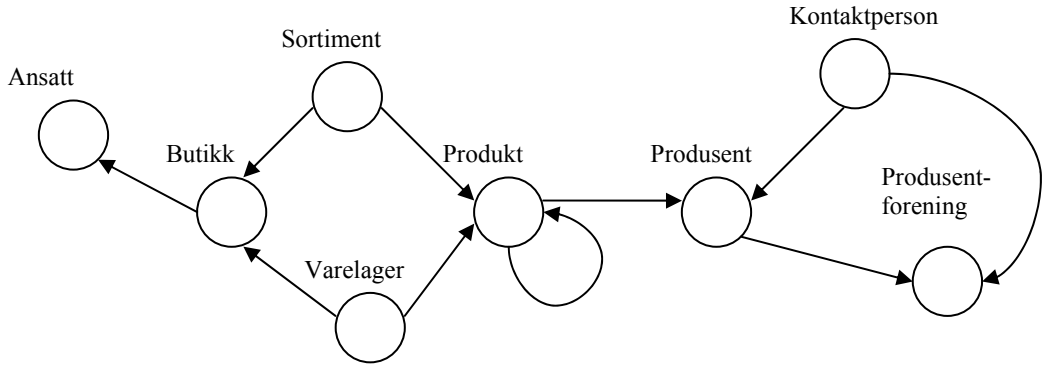
Jeg har valgt å lage meg en forholdsvis enkel produktdatabase for å demonstrere det jeg har gjort, og forklare tankemåten underveis. Samtidig har jeg lagt inn

noen ”feller” som synliggjør noen av problemene en kan komme over. Eksempelvis er det en sirkelreferanse fra produkt til produkt. Denne kan tenkes som en relasjon til et ”hovedprodukt”, eller ”se også” produkter, men er først og fremst med for å illustrere problemområder med slike referanser. Av samme grunn er både kontaktpersonene til produsenter og produsentforeninger lagret i samme relasjon; ”kontaktperson”, noe som ikke er gitt ville bli gjort i det virkelige liv.

Figurene nedenfor viser relasjonsskjemaet til produktbasen fremstilt som en foreklet UML tegning, samt hvordan en kan se på denne som en rettet graf.



Figur 13: Forenklet UML diagram over databasen



Figur 14: Eksempeldatabasen som en rettet graf

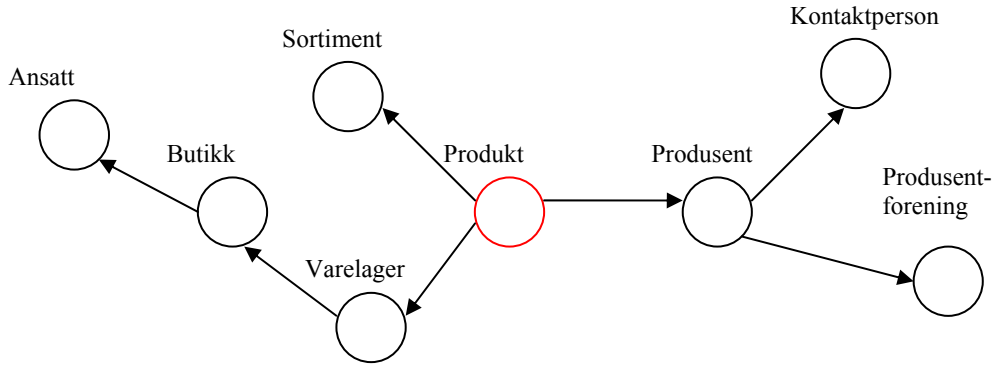
7.1.2 Analyse av systemtabellene

Det jeg ønsket meg var en hjelpetabell med informasjon om hvordan de ønskede relasjonene relaterte til hverandre og angivelse av hvilke attributter som kunne fortelle hvordan et eventuelt datautdrag kunne gjøres. Hvis en ser eksempeldatabasen som en urettet graf, fremgår det for eksempel at en kan komme til ”sortiment” både direkte fra ”produkt” og via ”varelager” og ”butikk”. I dette eksemplet ser en fort at det er det første som er ønskelig, men vil det alltid være slik? Kan en konkludere med at korteste vei alltid vil være ønskelig? I alle de tilfeller jeg så på og tenkte meg, var det slik. Ved hjelp av systemtabellene pg_class, pg_attribute og pg_constraints som (bl.a.) beskriver henholdsvis relasjonene, attributtene i relasjonene og fremmednøklene som finnes i en database kan en utføre en slik analyse ut fra baserelasjonen.

Hovedproblemet med en slik analyse er å finne ut hvor en skal gjøre en avskjæring. En relasjonsdatabase av litt størrelse vil nesten alltid være såpass kompleks at når en ser på den som en graf, vil den ha sykler. Dette vil igjen medføre at når en skal generere XML, kan en risikere at den vil gå i en uendelig løkke, dessuten at en får med veldig mye data som ikke nødvendigvis vil være så relevant. La oss si at vi ikke gjorde noen form for avskjæringer, men kun gravde oss nedover langs alle kantene i grafen når vi skulle generere XML for produkt A. Dette produktet vil kanskje ligge i varelageret til flere butikker, som igjen har et sortiment med mange andre forskjellige produkter. En slik utflating av produkt A ville dermed ha med informasjon om alle produkter som måtte være i det samlete sortimentet som de butikker som har produkt A på lager og deres tilhørende informasjon osv. Men det vil ikke være all informasjon som en kan grave opp fra databasen kun ved å følge fremmednøkler som er relevant for produkt A. Måten jeg forsøkte å løse dette på var å gjøre om grafen til en asyklisk rettet graf der en bestemte fra hvilke noder en skulle følge videre for å flate ut data. Jeg har forsøkt å løse dette på flere ulike måter, Disse alternativene beskrives nedenfor sammen med fordelene og ulempene med hver av dem.

Minimumspennviddetre

En av de enkleste måtene å se på problemet på er å gjøre om grafen til et *minimumspennviddetre* (minimum spanning tree). På den måten er en garantert at grafen er asyklisk samt at en ikke ”dobbeltlagrer” for mye informasjon. Eksempelvis, hvis en kan komme til ”butikk” både via ”sortiment” og ”varelager”, vil all informasjon om butikken og dens ansatte komme i begge tilfeller. I dette eksemplet vil det være regelen og ikke unntaket at et produkt er både i varelageret og i sortimentet til en og samme butikk. Ved å gjøre det om til et minimumspennviddetre ville en unngått problemstillinger som dette. Figuren under viser hvordan en slik analyse kunne sett ut:



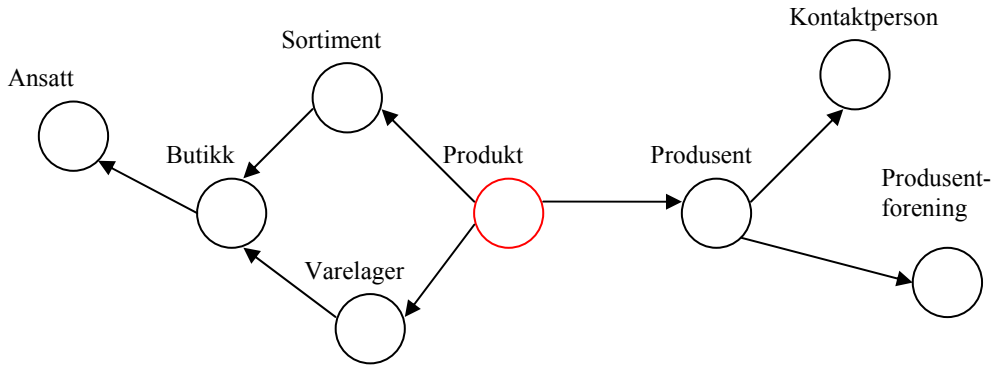
Figur 15: Eksempeldatabasen som en rettet minimumspennviddetre med produkt som rot.

Problemet med en slik analyse er at en kan risikere å miste mye data når en skal bruke den. Det enkleste eksemplet kan være at en butikk er tom for en bestemt vare. Ved utflating ville ikke informasjon om at denne butikken vanligvis fører varen komme med. Så jeg prøvde i stedet å finne korteste vei fra baserelasjonen til de andre relasjonene.

Bruk av korteste vei-algoritmen

Alternativet der grafen gjøres om til en rettet graf basert på korteste vei, er vist i Figur 16.I dette tilfelle ville en opprettholde kanten mellom sortiment og butikk med de fordeler og ulemper det medfører. Et eksempel kan være at hvis ansatte har mange fremmednøkler til andre relasjoner som adresser, lønnslister osv så kan ansattdata bli forholdsvis store og det er ikke nødvendigvis ønskelig å dobbeltlagre ansattinformasjon hvis en kom til en og samme ansatt via butikk fra både fra "sortiment" og "varelager".

Selv om bruk av korteste vei-algoritmen fører til at en legger til en kant til ser en at kantene fra produkt til produkt og fra kontaktperson til produsentforening forsvinner. Dette kan igjen medføre at en mister data. Det er heller ikke her gitt om en kontaktperson for en produsent er den samme for produsentforeningen. Det kan til og med ligge begrensninger i databasen som sier at en kontaktperson kun kan være kontakt for enten en produsent eller en forening. I så fall ville informasjon om produsentforeningkontaktpersonene ikke komme med.



Figur 16: Eksempeldatabasen som en rettet graf med korteste vei etter analyse med produkt som baserelasjon

Andre måter å avskjære på

Dybdebegrensning

Bruk av automatisk uttrekking kan være en teknikk, som ikke nødvendigvis henter ut all data i en database, men er en måte å hente ut relatert informasjon fra en baserelasjon. Men hva er så relatert informasjon? Hvor mange fremmednøkler utover skal en følge før det vil være interessant for å se på en utflatning av produktet? Vil informasjon om barn fra tidligere ekteskap som ektefellen til en ansatt som jobber i en butikk som har produktet i varelageret være interessant å ta med (gitt at slik informasjon lå i databasen)? Det er vel heller tvilsomt. Derfor la jeg inn denne kanskje mest interessante mulighet for avskjæring, nemlig mulighet til å avskjære på dybde – hvor mange fremmednøkler en skulle følge.

Manuell avskjæring/tilpassing

Kanskje den mest aktuelle bruken av en slik analyse måtte være å videreføre denne tankegangen inn i et hjelpeverktøy til bruk ved ekstraksjon av strukturert data. I så fall vil analysen kunne levere et forslag som en så manuelt kan tilpasse ved å legge til og fjerne kanter i grafen.

Løkkedeteksjon i data

Hittil har jeg kun sett på selve relasjonsskjemaet, først og fremst fordi dette er en enkel måte å tilnærme seg problemet på. Likevel har jeg sett at det også byr på en del problemer. Det er f.eks. ikke nødvendigvis et problem med sykler selv om det kanskje finnes i skjemaet. Det er jo strengt tatt ikke attributtet i en tabell som peker på et annet attributt – det er data lagret i et bestemt tuppel som peker på et annet. Det kan også finnes underliggende begrensninger slik at data i databasen ikke ligger i sykler selv om det kan se slik ut i skjemaet. En alternativ tilnærming kan derfor være å ikke gjøre begrensninger på skjemaet, men å gjøre bruk av korteste vei-algoritmen for å finne retningen en skal gå. En kan faktisk se på en kombinasjon av data og fremmednøkler som en graf der to forskjellige produkter vil være to forskjellige noder. Det vil da være umulig å detektere eventuelle sykler ved å se på skjemaet, men en blir nødt til å gjøre avskjæringer når en genererer XML for å forhindre uendelige løkker. Dette kan gjøres ved å registrere hvilke tupler en har vært innom og stoppe hvis en kommer til et tuppel der en har vært før.

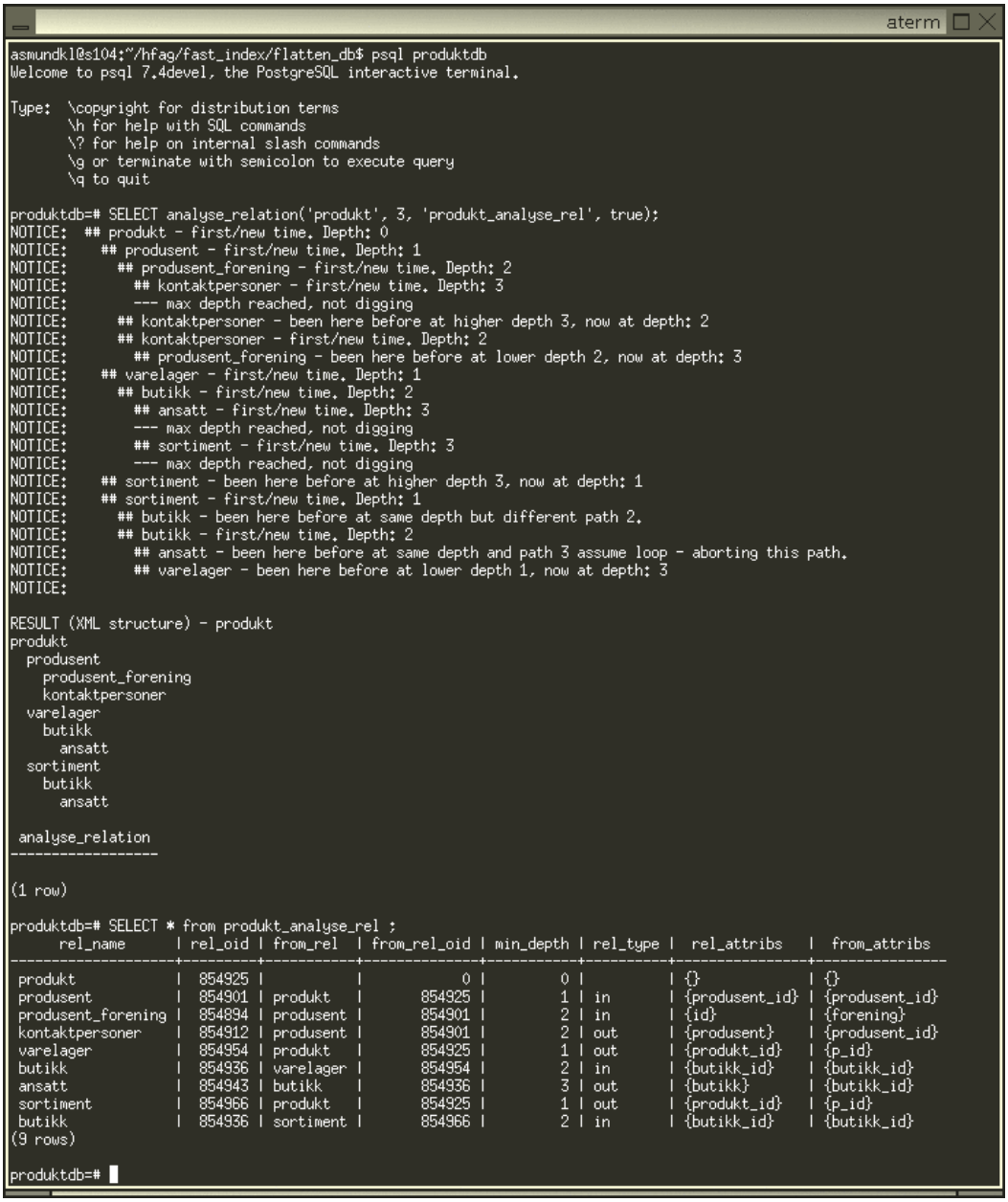
Analysefunksjonen

I min prototyp valgte jeg å kun analysere skjemaet for å forenkle oppgaven. Jeg valgte tankeganger fra korteste vei for bestemme hvordan data skulle genereres. Figur 16 viser hvordan en slik analyse så ut for eksempeldatabasen med produkt som baserelasjon med dybdebegrensning lik tre.

Jeg laget en database funksjon skrevet i plpgsql som genererte opp en analysetabell. Dette for at jeg kunne kjøre analysen kun en gang for deretter å bruke resultatet når jeg skulle generere ut XML. Funksjonen kalte jeg *analyse_relation()* og den tok i mot fire parametre; navn på baserelasjonen, maks dybde, navn på analyserelasjonen og et boolsk parameter brukt til debugging da det skrives ut mer analyseinformasjon under generering.

I analysetabellen måtte jeg ikke bare vite hvilken vei jeg skulle gå når jeg skulle hente ut data, men jeg måtte også ta vare på hvordan fremmednøklene var i databasen samt hvilke attributter som denne nøkkelen gikk på.

Figur 17 viser eksempel på kjøring og resultatet i analysetabellen som jeg i eksemplet kalte ”produkt_analyse_rel”. Andre tuppel i tabellen leses som at relasjonen ”produsent” med OID 854901 har et attributt ”produsent_id” som blir pekt på av en fremmednøkkel fra attributtet ”produsent” i relasjonen ”produkt” med OID 854925. Relasjonsattributtene blir lagret som arrayer for å kunne følge fremmednøkler over flere attributter.



Figur 17: Kjøreeksempel på analyse av "produkt" og resultatet av kjøringen.

Andre problemstillinger

I tillegg til problemene jeg har nevnt med at kanter kan gå tapt i analyseprosessen er det en problemstilling knyttet til begrensninger/relasjoner som ikke ligger som fremmednøkler. Det kan f.eks. tenkes at det finnes forhold som databasedesigneren, av en eller annen grunn, ikke ønsker å overholde med fremmednøkler, men i stedet bruker triggere.

Uansett kommer en ikke forbi at ved denne tankegangen tar en alltid utgangspunkt i en gitt baserelasjon. Selv om en ikke har gjort en begrensing på dybde, risikerer en derfor at ikke alle relasjonene i databasen kan nås fra baserelasjonen - grafen er ikke nødvendigvis *sammenhengende* (connected).

7.1.3 Generering av XML basert på analysen

Når jeg så har analysert relasjonene og lagret denne informasjonen i en tabell kan jeg begynne å bygge XML basert på denne. Jeg lagde en funksjon, *flatten_to_xml_from_analyse()*, som tok imot navnet på baserelasjonen og en oid til et tuppell fra denne og navnet på analysetabellen. Dermed kunne jeg generere opp data for dette tuppelet. Et eksempel på spørring mot produktdatabasen kan være:

```
SELECT flatten_to_xml_from_analyse(
      (SELECT oid FROM pg_class WHERE relname='produkt'),
      (SELECT oid FROM produkt WHERE p_id=4),
      'produkt_analyse_rel'
);
```

Nedenfor er et eksempel på deler av data som ble generert fra testdatabasen.

Figur 18: Generering av XML over "produkt" tabellen basert på analysen.

XML'en jeg genererte var et veldig enkelt velformatert dokument der navnet på baserelasjonen utgjorde rot-taggen og hvor alle relasjoner som den måtte ha en fremmednøkkel til eller fra også ble tagger som igjen hadde tagger under seg. Attributter ble oppført som enkle terminerte tagger på formen '</attributtnavn value="verdi">'. Utformingen av selve dokumentet er forholdsvis enkelt å forandre, og jeg så også på muligheter for å eksportere data som rowset xml. Imidlertid fant jeg ingen god måte å representere nøsting på i og med at den er beregnet først og fremst for å ta ut data fra en og en relasjon.

En måte å overkomme problemet med at mye data kan komme til å lagres mange ganger (f.eks. butikk og dens ansatte) kan være å benytte seg av Xpath, evt bruk av id'er med referanser, for å kun lagre denne informasjonen ett sted i dokumentet for så referere til den. Det vil ikke nødvendigvis øke lesbarheten og ikke nødvendigvis gå fortere for en datamaskin å lese, men vil kunne begrense betraktelig mengden XML som produseres. På den annen side kan kanskje nettopp det at noe forekommer flere ganger i ett dokument enn i et annet være informasjon som en ønsker å ta vare på fordi dette kan gi nyttig informasjon ved rangering av dokumenter.

Jeg har med dette vist at prototypen er i stand til å hente ut strukturert data som et XML dokument basert på data i systemtabeller. Det er imidlertid et problem at relasjoner som skulle være med potensielt ikke trekkes ut på grunn av de ikke er laget som fremmednøkler. Bruk av korteste-vei-algoritmen kan også føre til at en velger bort noen relasjoner som burde være med. Trolig er det ikke mulig å gjøre en optimal utflating uten manuell tilpassing, men en tilsvarende metode kan brukes som et hjelpeverktøy.

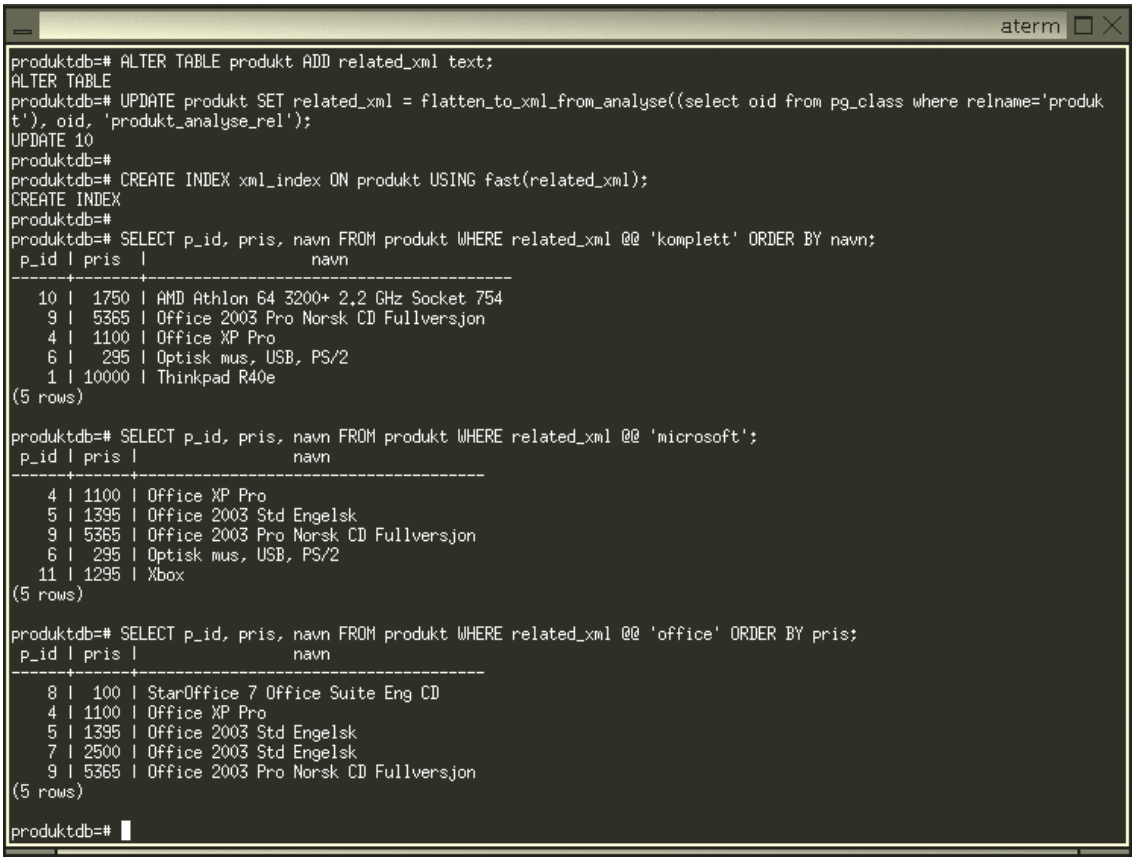
7.1.4 Hva kan en bruke automatisk utflating til?

Det fremgår av det ovenstående at dette ikke egner seg som en form for "dump" av en database. Til det blir datamengden for stor og usikker i og med at en risikerer at visse data ikke kommer med samtidig som at veldig mye data potensielt kommer ut. Gitt f.eks. at en produsent hadde veldig mye data lagret om seg, så ville alle produkter som hadde de samme produsentene dobbeltlagret dette – ikke akkurat tredje normalform, men samtidig er jo ikke det som er målet her. Likevel ser jeg for meg mange andre spennende bruksområder for denne teknikken og nedenfor følger noen av dem.

Relatert søk

En av fordelene kan være mulighet for "relatert" søk. I en database i dag, også med vår løsning på fulltekst søk, er det kun muligheter for å kunne gjøre veldig eksakte søk, dvs en må vite eksakt hva en er ute etter. Hvilken relasjon det ligger i, hvilket attributt osv. For å tillate relatert søk kunne en legge til et attributt i produkt relasjonen som en så la til denne genererte XML informasjonen. Dette attributtet kunne så indekseres med vår fulltekstindeks og så kunne en søke ved hjelp av indeksen. Dette ville muliggjøre relatert søk slik at en søker i store deler av databasen på en gang. Eksempel på bruksområdet kan være hvis en ønsker å søke på et bestemt produkt, men husker ikke akkurat hva det heter, og så hadde mulighet til å søke ved hjelp av fulltekstsøk både på produsentnavn, produktnavn og beskrivelse av produktet. Med et slikt system kan en da søke over disse i ett

og samme søkefelt (f.eks. i en webapplikasjon). Dette er en form for forenkling som en bl.a. ser på de nye websidene til Gule Sider der en ikke må bestemme om det er et bestemt firma en søker på, eller en bransje og en trenger ikke skrive stedsbegrensninger i et eget felt – alt kan skrives inn i det samme feltet. Dette forenkler oppgaven for brukere, og gjør startterskelen enda lavere for å begynne å bruke det.



Figur 19: Bruk av relatert søk basert på utflating til XML.

Figuren over viser et eksempel der jeg utvider produkttabellen til å inneholde ett attributt til kalt ”related_xml”. Dette fyller jeg så med data som blir generert ved automatisk utflating basert på analysen. Deretter lager jeg en indeks over dette attributtet som muliggjør fulltekstsøk over denne XML’en. I eksemplet viser jeg tre spørringer som viser noe av styrken ved dette. Den første gir en liste over de produkter som befinner seg i varelageret og/eller sortimentet til ”Komplett”. Det andre eksemplet søker i relatert-informasjon-attributtet etter ”Microsoft” og får da opp alle produkter som bedriften har i produktdatabasen. I det siste eksempelet kan en tenke seg at brukeren er ute etter kontorverktøy, men ikke nødvendigvis en bestemt versjon og spørringen vil derfor gi svar på de som finnes i systemet. Her kunne en selvfølgelig laget en egen indeks over produktnavnet, men styrken i dette eksemplet er at alt dette får en ut ved å kun søke i ett felt!

Ved å legge den genererte XML’en i et attributt i databasen, åpner en ikke bare for ny funksjonalitet, men samtidig også for oppdateringsproblemer. Hvordan

kan en sørge for at den utflatede informasjonen en har til enhver tid vil være overens med slik data faktisk er i databasen?. En måte å sikre dette på er å lage en liten (forholdsvis enkel) funksjon som også tar utgangspunkt i analysetabellen, men i stedet for å hente ut data, vil den automatisk generere opp triggere som har til ansvar å generere opp ny XML i de tupler som henter/hentet ut informasjon fra tuppelet. Eksempelvis måtte det vært en trigger på produsent som gjorde at når den ble oppdatert,måtte den regenerere XML til alle de produkter den hadde. Problemet med en slik tilnærming ville vært at det potensielt kunne medført mye oppdateringer. Mye av styrken til relasjonsdatabaser med god modellering er nettopp at en kan unngå denne typen oppdateringer for å minimalisere risikoen for oppdateringsanomalier som kan oppstå hvis en gruppeoppdatering feiler. Hvis det viste seg at databasen hadde en slik struktur at noen få relasjoner ble oppdatert hele tiden og disse igjen medførte at en måtte oppdatere XML'en til store deler av databasen ville denne løsningen vært helt uholdbar. Har en derimot antagelsen om at systemets oppførsel ligner mer på informasjonsgjenfinningsystemer med få oppdateringer, vil dette være forsvarlig. Har en også applikasjoner der det ikke er kristisk med en oppdatert versjon til envher tid, kan en også tenke seg gruppeoppdateringer når systemet har liten last, f.eks. om natten.

Automatisk ekstraksjon av en enkelt relasjon

Hvis en bestemmer at analysen kun skal gå ned til dybde 0 vil en kun få en dump av data fra baserelasjonen, men samtidig har en laget et generisk eksportverktøy som kan hente ut data fra en relasjon f.eks. i rowset xml ut til annen software som forstår data på dette formatet. Dette skjer uten at en behøver å skrive så mye som en linje kode - noe som vil gjøre integrering mellom database og eksterne applikasjoner som ikke skal ha "live" tilgang enklere. Dette er imidlertid bare et sideresultat av det jeg har gjort og er noe som allerede i dag tilbys fra de fleste store RDBMS leverandørene.

Søk uten å gå via databasen

Siden resultatet som kommer ut er velformatert XML, betyr det at en kan lese ut mye data uten å nødvendigvis å gå via databasen. Eksempelvis kan en bare ved hjelp av enkel XSL gjøre om data til HTML, noe som muliggjør at kunder og ansatte kan søke i produktdatabasen uten å belaste databasen. I og med at en søkemotor som oftest har mye bedre skaleringsmuligheter enn en database vil dette kunne avlaste og tilby raskere, bedre og mer skalerbart søk uten bruk av mye penger og resurser på å skalere opp en database.

7.2 Mulige utvidelser av vår implementasjon

I kapittel fem viste jeg hvordan vi implementerte en prototyp der vi bygde en bro mellom FDS og PostgreSQL ved å tilby en ny indeksaksessmetode. Dette muliggjorde fulltekstsøk over et enkelt attributt i en relasjon. Jeg vil her omtale noen av de problemstillingene vi ikke løste, og skissere løsningsforslag og andre utvidelser for å utnytte begge systemene bedre.

7.2.1 Implementasjon av CONTAINS-operatoren

Et av hovedproblemene med vår prototype-implementasjon var begrensningen vi la på spørreplanleggeren ved å kreve obligatorisk bruk av indeksmetoden vår. Normalt vil den, på bakgrunn av statistisk oppsamlet informasjon, kunne gjøre et valg mellom flere mulige måter å hente opp data. Eksempelvis må den ta avgjørelser om det skal være bruk av indeks, sekvensielt søk og i tilfeller der det finnes flere indekser og/eller relasjoner hvilken som skal brukes først. Grunnen til at vi gjorde det på den måten er at PostgreSQL ikke har mulighet for å evaluere en FDS-spørring. Problemet illustreres veldig godt når en gjør spørringer som er en blanding av IR og tradisjonell databasespørring. Eksemplet nedenfor viser ekstremvarianten:

```
SELECT * FROM philosophers
WHERE data CONTAINS 'Kant OR "das Ding an sich"' AND oid=1234;
```

I en relasjonsdatabase vil en oid være en unik id for ethvert attributt (så fremt tabellen har oid'er). Spørringen over ville normalt bli evaluert ved at en fant frem tuppelet, hvis det eksisterte, med oid=1234 og så evaluert om tuppelet tilfredstilte den andre betingelsen. Siden det her er snakk om en IR-betingelse, kan ikke databasen evaluere det, og måtte isteden gjøre spørringen ved hjelp av fast indeksen, noe som potensielt kunne returnere veldig mange tupler. Endelig måtte det gjøres et filter på den rette oid'en. Det sier seg selv at den første måten å utføre spørringen på vil være mye mer effektiv.

En måte å løse dette problemet på ville være å implementere CONTAINS-operatoren slik at den også kan evaluere ett og ett uttrykk. Jeg har tidligere nevnt at databasen ikke har mulighet for å kunne evaluere et IR uttrykk opp mot en tekst, men hvis en kunne få FDS til å evaluere det, kunne man implementere operatoren. For å gjøre dette må en imidlertid utvide enten PostgreSQL eller FDS. Nye operatorer i PostgreSQL lages ved å oppgi en funksjon som mottar de to argumentene. I dette tilfellet vil det være tekstdata (dokumentet) og en spørring. Siden vi kun mottar data og ikke noen form for id på hvilket tuppel/dokument det er snakk om, måtte det i så fall vært en mulighet i FDS for å gjøre en slik evaluering. Dette ligger imidlertid ikke i FDS som kun kan fortelle om hvilke indekserte dokumenter som tilfredstiller en spørring, ikke svare om en gitt tekst gjør det.

Det er meget gode argumenter på begge sider for dette. Hvis en skulle kunne benytte en operator kun på tupler, kunne en ikke benyttet operatoren andre steder, for eksempel i en SELECT del. Samtidig ligger det i konseptet at FDS ikke har noe grunnlag for å gjøre en kjapp evaluering om et dokument tilfredstiller en IR spørring uten å først indeksere og analysere det. Vi kunne "hacket" til en løsning der vi først lagde et midlertidig dokument i FDS med dataene for så å spørre om dokumentet tilfredstilte spørringen. Dette hadde tatt uforholdsmessig lang tid og er helt urealistisk.

En annen løsning hadde vært å implementere en funksjon i stedet for en operator. På den måten kunne vi tatt i mot informasjon som identifiserte tuppelet, slik at vi kunne gjort en spørring mot FDS som igjen evaluerte IR spørringen mot et gitt

dokument. Spørringen over kunne i så fall bli løst på en bedre måte eksempelvis ved:

```
SELECT * FROM table WHERE contains_tuple('table', 'data', oid,
'Kant OR "das Ding an sich"') AND oid=1234;
```

Årsaken til at den må skrives om som en funksjon er at vi er avhengig av å vite hvilket dokument i FDS det er snakk om, og for å identifisere det trenger vi informasjon som identifiserer hvilket tuppel det er snakk om. Hver gang funksjonen blir kalt vil den hente ut informasjon om hvilket dokument det gjelder (basert på relasjon, attributtnavn og tuppel for å finne block og offset) for så å spørre FDS om det gir treff med den gitte spørringen. Forhåpentligvis vil da spørreplanleggeren ta beslutningen som gjør at den finner riktig tuppel med oid=1234 for så å gjøre et filter på disse ved hjelp av contains_tuple funksjonen. I og med at hver gang denne funksjonen kalles vil en gjøre et søk i FDS, må brukeren analysere spørringen og være sikker på at det er slik den vil bli kjørt hvis ikke vil det ikke lønne seg. Dette er imidlertid ingen god løsning av to årsaker. For det første er ikke oid strengt tatt unik i PostgreSQL. Siden den bare består av en positiv fire bytes heltall, garanterer ikke PostgreSQL for at den vil være unik innen en tabell en gang. Dermed kan en komme i situasjoner der denne funksjonen ikke kan garantere korrekt resultat. For det andre overkjører fremdeles brukeren spørreplanleggeren.

For å løse dette på en integrert måte, dvs slik at en kun lager en operator og så blir det opp til spørreplanleggeren om den vil evaluere ett og ett tuppel eller om den skal bruke indeks metoden, måtte en utvide PostgreSQL. Det måtte innføres en ny type "operator" som i stedet for kun å motta to tekst argumenter vil motta tuppelidentifikasjon som det ene argumentet og tekst som det andre argumentet. På den måten kunne operatorfunksjonen evaluere uttrykket mot et dokument i FDS ved å bygge opp en dokumentid. I tillegg måtte en utvide funksjonen amcostestimate slik at denne ikke lenger returnerer 0 for å indikere at indeksen alltid er best, men returnerer et mer realistisk kostnadsoverslag. Dette ville i så fall utvidet operatorbegrepet slik at en ikke kunne bruke den i alle sammenhenger og kun mot atributter som var indeksert. En bedre løsning vil trolig være å indeksere flere attributter i en og samme indeks for så la aksessmetoden evaluere begge betingelsene. Se 7.2.3 for mulig utvidelse for å tillate dette.

7.2.2 Indeksering av andre datatyper enn tekst

Å utvide vår implementasjon til å kunne ta andre datatyper vil ikke være noen stor operasjon. En måtte lage en ny operator for hver datatype en ville legge til, dessuten måtte en utvide indeksprofilen til å ha med et felt til for hver type. I tillegg måtte en utvide funksjonene som brukes i aksessmetoden for å takle flere typer data. Grunnen til at vi i vår prototypeimplementasjon bare har sett på tekstdata, er begrenset tid til rådighet, og dessuten at tekstdata er mest interessant ved en integrasjon av et IR system og en database. FDS er imidlertid også veldig rask til å løse andre typer spørringer basert på andre datatyper som for eksempel tall. Tilsvarende som ved indeksering av tekstdata bygger den opp spesialiserte indekser slik at søk som større enn og mindre enn kan løses raskt. Normalt sett vil en ikke tro at bruk av ekstern indeks kan løse spørringer spesielt raskere enn

f.eks. en B-tre indeks i PostgreSQL siden en må hente opp alle tuplene i databasen etter at søket er gjort i FDS. Imidlertid kan det vise seg å være en fordel hvis en kan gjøre mer av spørringen i FDS f.eks. i form av EXISTS og aggregerings-spørringer. For å gjøre det, må en imidlertid også ta andre hensyn. Dette vil bli diskutert nærmere i neste avsnitt.

7.2.3 Multikolonne-indeksering

Multikolonneindekser brukes tradisjonelt hvis en ønsker å overholde en begrensning om at to eller flere attributter til sammen skal være unike eller fordi en ønsker å gjøre raskere oppslag på et sett av betingelser. Eksempelvis kan det øke hastigheten betraktelig på søk hvis en vil ha ut alle biler av typen Lada som er gule hvis det er en indeks på de to attributtene. Multikolonne indekser implementeres ofte ved hjelp av en nestet trestruktur. Rekkefølgen av attributtene en indekserer kan ha stor betydning for hvor lønnsomt det vil være å bruke en slik indeks. I vår implementasjon mot FDS benyttet vi oss kun av ett felt i indeksprofilen og det ble dermed kun bygget indekser over dette feltet. Hvis en legger til flere felter i indeksprofilen, vil hver av disse bli indeksert. Multikolonne indekser i FDS har derfor en litt annen natur enn i tradisjonell databaseverdenen. Egentlig kan en se på dem som et sett med single indekser. Det er likevel ønskelig å se på muligheten for å bruk av multikolonne indeks mot FDS. Eksempelvis kan en se på spørringen:

```
SELECT title, abstract FROM philosophers
WHERE body CONTAINS '"The meaning of life"' AND
      conclusion CONTAINS '42 AND Adams';
```

Ved bruk av vår implementasjon kunne en indeksert attributtene body og conclusion, men i spørringen over ville PostgreSQL, ved hjelp av FDS hentet ut alle dokumentene som handlet om meningen med livet, dernest alle som handlet om 42 og Adams. Hver av disse del-resultatene kan potensielt være veldig store, men i og med at en kun er ute etter de som finnes i begge settene må PostgreSQL ta snittet av de to for å komme frem til det endelige resultatet.

Hvis vi hadde implementert CONTAINS operatoren som diskutert i avsnitt 7.2.1, ville antakeligvis PostgreSQL valgt kun å bruke en av indeksene (den som den statistisk ville anta skulle gi færrest treff) for så å bruke denne operatoren som et filter på delresultatet. Men som beskrevet, er det vanskelig å få til, og operasjonen vil sannsynligvis være forholdsvis tidkrevende.

Kunne en imidlertid indeksert begge disse atributtene som et dokument i FDS kunne en skrevet om spørringen slik at PostgreSQL kun mottok det endelige resultatet. Dette forutsetter selvfølgelig at FDS vil kunne løse oppgaven vel så raskt som PostgreSQL, noe man må tro er tilfelle, ettersom FDS kan distribuere sine oppgaver. Som nevnt kan også FDS brukes til å indeksere annen data enn tekst, noe som betyr at flere attributter, om ikke alle, i en relasjon kan indekseres. Det åpner for at flere spørringer kan utføres der, men som jeg vil diskutere i neste avsnitt, medfører det også en rekke problemstillinger som må løses.

Utvidelse på indeksprofilen

I vår implementasjon hadde vi kun ett dataattributt i indeksprofilen som vi kalte ”data”. Dette betyr at vi kun har ett ”attributt” per rad/dokument. En mulig utvidelse består i å la indeksprofilen ha et sett med ”generiske” attributter i stedet for kun det ene dataattributtet,. F.eks. text1, ... textN og num1, ... numM. Dette vil muliggjøre spørringer i FDS som eksempelvis

```
text2:’”The meaning of life”’ AND text7:’42 and Adams’
```

som vil kunne gjøre det vi ønsker. *text2* refererer her til feltnavnet i indeksprofilen. Å skrive *attributtnavnet:søkebetingelsen* er i dag vanlig å bruke i søkemotorer for å angi hvilken indeks en ønsker å søke i (eksempelvis *site:domene* og *intitle:overskrift* begrensningen i Google). Vi trenger ikke skrive spørringer slik i prototypen fordi vi definerte *content* (som igjen inneholt datafeltet) som default feltet å søke i.

Utvidelser i PostgreSQL

Ny systemtabell

Med en utvidelse av indeksprofilen vet en ikke lenger uten videre hvor data for et gitt attributt i databasen ligger. I vår implementasjon visste vi at det til enhver tid ville ligge i ”data”. For å holde rede på hvilke felter i indeksprofilen som holder informasjon om hvilke attributter i en gitt relasjon trenger vi en form for systemtabell. Den kan holde informasjon som at i tabellen philosophers er body i text2, conclusion i text7, no_words i num3 osv. Denne systemtabellen kan en så benytte seg av når en skal skrive om et sett med betingelser til en spørring mot FDS. Systemtabellen burde bli automatisk generert når en laget en ny indeks. En slik systemtabell kunne sett ut som:

indexOID	**AttributeNo**	**FastField**
1234	8412	text1

Grunnen til at indeksoid bør brukes og ikke relasjonsoid er at en kan lage flere indekser over samme attributt hvis en måtte ønske det. Fra indeksoid’en kan en finne frem til hvilken relasjon den er på.

Omskriving av spørringer

Hovedproblemet med en slik utvidelse blir hvordan en skal integrere det på databasesiden. Som med CONTAINS operatoren, kan det løses på flere måter avhengig av hvor mye jobb en er villig til å gjøre, og hvor god integrasjon en ønsker.

Den beste, men desidert mest kompliserte løsningen for å få til en full integrasjon ville være å skrive om spørringsparseren. Da kunne en analysere spørringen for å se om det var flere betingelsesledd som kan bindes sammen ved at spørringen skrives om før den blir sendt videre til backend. Dette ville medføre forholdsvis mye arbeid og det er ikke sikkert at det kunne bli løst på en god måte. Nedenfor skisseres to enklere, men ikke fullt så gode løsninger.

Bruk av dummyindekser

Som nevnt under implementasjonskapittelet måtte vi tvinge PostgreSQL til alltid å bruke Fast-indeksen fordi den ellers ikke ville kunne evaluere betingelsen. Hvis vi utvider vår implementasjon av indeksmetoden til å kunne indeksere over flere atributter, vil PostgreSQL tro at det snakk om en multikolonne-indeks. Men selv om det i FDS vil ligge flere kolonner med data, vil det ikke være snakk om en ”tradisjonell” multikolonne indeks, men heller en samling av enkeltindekser som det kan spørres fra ved søk. Hvis en så lager en multikolonneindeks over to attributter, attr1 og attr2 i databasen, vil ikke PostgreSQL forstå at den faktisk kan bruke denne indeksen til også å svare på spørsmål angående kun attr2.

En måte å løse det på vil være å generere opp en form for dummyindekser. En måtte i så fall lage en ny indeks aksessmetode, fastdummy som ikke gjorde annet en å slå opp i data som allerede er blitt indeksert av en omsluttende multikolonneindeks. Denne indeksmetoden måtte i så fall ikke gjøre noe ved oppdateringer i datagrunnlaget, siden dette vil bli tatt hånd om av den ”egentlige” indeksen (vår implementasjon er jo heller ikke noen indeks i seg selv siden den kun videreformidler oppdateringer og spørringer til FDS). Dummyindeksens eneste oppgave vil være å opptre som en indeks slik at spørreplanleggeren kan velge denne. Ved søk vil den benytte seg av de metodene vi allerede har implementert for å utføre spørringen til FDS og hente opp riktig data.

Eksemplet jeg har brukt så langt var ved indeksering av to attributter, men hva om en ønsker å indeksere flere attributter som et dokument i FDS? Hvis en for eksempel ønsker tre attributter, må en ikke bare lage en dummyindeks på hver av attributtene, men også på alle mulige kombinasjoner av to og to. Antall dummyindekser m for n attributter blir: $m = \sum_{i=1}^{n-1} \binom{n}{i} = \sum_{i=1}^{n-1} \frac{n!}{i!(n-i)!}$ Altså summen av alle uordnede kombinasjoner fra 1 til n-1. Riktignok blir det totale antall indekser summen fra 1 til n, men i og med at antall kombinasjoner av n over n alltid gir 1 så kan en trekke fra den i og med at en også vil ha den ”ekte” indeksen.

Som en ser vokser antall dummyindekser eksponentielt. For en relasjon med fem attributter betyr det 30 dummyindekser, og for en med 10 attributter betyr det 1022 dummyindekser. Hvis bruken av databasen tilsier at en sjelden gjør begrensninger på flere enn tre attributter av gangen, er det kanskje bare interessant å generere opp dummyindekser for kombinasjoner av tre attributter og mindre (ved kun å lage dummyindekser over kombinasjoner av tre attributter blir det 175 dummy indekser ved 10 attributter). Indeksaksesshjelpemetoden aminsert vil bli kalt for hver av dummyindeksene ved innsetting, men siden den vil returnere med en gang ville dette trolig ikke forsinket innsettinger nevneverdig. Det vil derfor ikke nødvendigvis medføre noe overhead under kjøring.

Å lage alle disse dummyindeksene bør ikke være en manuell jobb siden det fort er snakk om ganske mange. En måte å håndtere det på vil være å sørge for at når en generere en multikolonne indeks, vil den automatisk generere opp de

nødvendige dummyindeksene. Siden dette kanskje ikke alltid er ønskelig, kan en lage en funksjon fast_optimize(rel_name) som genererer opp en indeks over hele relasjonen, samt lager de nødvendige dummyindeksene.

Spørreplanleggeren i PostgreSQL vil da se at enhver spørring som har spørrebetingelse som går på et ukjent antall kombinasjoner av attributter i tabellen har en indeks på akkurat de attributtene, og kan velge om den ønsker å bruke den. Inneholder spørringen et fulltekstuttrykk er PostgreSQL tvunget til å bruke indeksaksessmetoden, men kan samtidig også foreta eventuell annen filtrering i FDS. Hvordan en skal lage amcostestimate på disse dummyindeksene må derfor bestemmes ut fra hva slags type attributter det er snakk om. Dette betyr også at FDS kan bli brukt som indeksaksessmetode for andre typer søk enn fulltekst. Dette vil kunne avlaste databaseserveren, men vil aldri kunne utkonkurrere den hvis det er snakk om søk som f.eks. skal hente opp 90% av data på disken. Sekvensielt søk vil da være mye raskere enn å hente opp ett og ett tuppel.

Utvidelse av contains_table funksjonen

En annen løsning, som er enda enklere men mindre integrert mot SQL, vil være å utvide funksjonen contains_table til å behandle mer komplekse spørringer av typen :

```
SELECT title, abstract
FROM contains_table('philosophers',
      'body CONTAINS ""The meaning of life"" AND
      conclusion CONTAINS "42 AND Adams"');
```

I så fall måtte en analysere spørringen og skrive den om i funksjonen basert på data fra systemtabellen for at FDS kunne evaluere den. Dette er en enklere måte enn den som ble beskrevet i forrige avsnitt, siden vi har full kontroll på hva som skjer. Ulempen er igjen at det er brukeren som må bestemme hvordan databasen best skal foreta søket.

7.2.4 Flytte mer kompleks betingelsesevaluering til FDS

Til nå har jeg sett på mulige utvidelser for å kunne indeksere data i FDS og så benytte meg av dens indekser for å gjøre enkle begrensninger. Kunne en også flytte over mer kompleks betingelsesevaluering, eksempelvis EXISTS-spørringer og aggregasjoner? Dette er ønskelig først og fremst for å avlaste databaseserveren. Altså ett skritt nærmere å gjøre skriv til databasen, mens les fra søkemotoren som er billigere og bedre på distribusjon og skalering. Spørringen nedenfor er et eksempel på en enkel spørring med bruk av EXISTS som kun spør fra en tabell, produkt, og ønsker å finne de produkter som blir referert til via "se også" attributtet.

```
SELECT * FROM product main WHERE EXISTS (
      SELECT * FROM product ref WHERE ref.see_also =
main.product_id
);
```

En måte denne spørringen kunne løses på er om databasen gjenkjente en EXISTS-betingelse for så å analysere om den kan evalueres eksternt. I første omgang se at det ikke er join i spørringen, så sant ikke også dette er implementert, sjekke at relasjonen med de aktuelle attributtene er indeksert osv. En kunne i så fall evaluere hver enkelt EXISTS-betingelse for seg og velge ut de tupler som tilfredstilte den. Dette vil imidlertid resultere i veldig mange kall til søkemotoren (ett per tuppel), så det ville trolig ikke lønne seg. Det beste alternativet ville selvsagt vært å utvide spørrespråket i søkemotoren slik at den kunne håndtere en del av SQL.

Å flytte mer kompleks betingelsesevaluering til FDS, ville imidlertid ikke vært forsvarlig før en hadde løst problemene knyttet til transaksjoner og gyldighet av data. Eksempelvis har vi nevnt at data ikke blir slettet i søkemotoren før en VACUUM kommando kjøres, men at det ikke er noe problem da databasen vil sjekke om tuppelet er gyldig før det returneres til brukeren. Hvis en benytter seg av EXISTS-betingelser vil denne sjekken ikke lenger utføres. Søkemotoren kunne da svart at et gitt produkt var referert, mens det i virkeligheten ikke var det da det refererende produkt egentlig var slettet. Det samme problemet vil oppstå ved aggregeringer.

KAPITTEL 8: KONKLUSJON

Vi har laget en prototyp for integrasjon mellom PostgreSQL og FDS. FDS benyttes som en ekstern fulltekstindeks ved at vi innførte en ny indeksaksessmetode i PostgreSQL. Vi viser at en slik integrasjon er mulig uten å endre direkte i kildekoden til noen av systemene. Den nye indeksaksessmetoden tilfører PostgreSQL ny funksjonalitet, og tilbyr aktiv oppdatering av søkemotoren fra databasen. Bruken av denne typen ekstern indeks medfører imidlertid at systemet ikke tilbyr transaksjonsstøtte fordi ACID-kravene ikke lenger er garantert. Transaksjonsstøtte kan ikke oppfylles uten utvidelser i ett eller begge systemene. Likevel blir konklusjonen at denne integrasjonen tilbyr et mer avansert fulltekstsøk enn det som er vanlig å finne i en database. En enkel ytelsestest indikerer at ytelsen er meget skalerbar og viser at indeksaksessmetoden ikke bare tilfører ny funksjonalitet, men kan også tenkes å hjelpe å avlaste en database.

Videre presenterer jeg en prototyp for automatisk utflating av data fra PostgreSQL. Denne henter ut strukturert data som et XML dokument basert på en analyse av data i systemtabeller, men kan imidlertid ikke garantere at all relevant data kommer med. Ved å legge den genererte XML'en i et attributt i databasen for så å indeksere det ved fulltekstindeksen åpner en for søk i databasen uavhengig av det underliggende relasjonsskjemaet. Dette vil forenkle søk for uerfarne brukere. XML'en kan også brukes direkte for å presentere resultatet uten å gå via databasen. Ved å ha automatisk utflatet data i databasen må en imidlertid håndtere masseoppdatering av data.

Til slutt drøfter jeg mulige utvidelser av aksessmetoden og viser at det ligger store muligheter, men også store utfordringer i en slikt integrert system. Utvidelse for å indeksere annen data enn tekst kan gjøres forholdsvis enkelt, men vil trolig ikke ha stor verdi i seg selv. Innfører en imidlertid multikolonneindeksering, betyr dette at FDS kan indeksere hele relasjoner og evaluere flere begrensninger i søkemotoren. Den vil da kun returnere de dokumenter som tilfredsstiller alle betingelsene. Skal mer komplekse spørringer, slik som agregeringer evalueres i den eksterne indeksen, må det integrerte systemet tilby transaksjonsstøtte.

På tross av begrensninger i de to prototypene; manglende transaksjonsstøtte ved ekstern indeks og oppdateringsproblemer ved automatisk utflating, har de potensiell anvendbarhet i mange systemer. Dette kan for eksempel være systemer der man ikke har strenge krav til at oppdaterte data alltid skal være søkbare.

REFERANSER

[1] Knuth, E. D., *The art of computer programming*, 2 ed. Addison-Wesley, 1973.

[2] Espinoza, B. M. Jeg googler, altså er jeg - Finnes du ikke på Google, ja, da spørs det om du finnes. 22-10-2004. Oslo, Dagbladet. http://www.dagbladet.no/nyheter/2004/10/22/412092.html.

[3] Google Inc. 2004. http://www.google.com.

[4] Datamengden eksploderer i moderne bedrifter. 6-9-2004. www.digi.no. http://www.digi.no/php/art.php?id=108983

[5] Mohan, K. and Krithi, R. Efficient Transaction Support for Dynamic Information Retrieval Systems. 147-155. 1996. ACM Press. Proceedings of the 19th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval.

[6] Samuel, D., Amjad, D., Lisa, A. S., and Jagannathan, S. Integrating IR and RDBMS using Cooperative Indexing. 84-92. 1995. ACM Press. Proceedings of the 18th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval.

[7] Surajit, C., Umeshwar, D., and Tak, W. Y. Join Queries with External Text Sources: Execution and Optimization Techniques. 410-422. 1995. ACM Press. Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data.

[8] Tak, W. Y. and Jurgen, A. Integrating a Structured-Text Retrieval System with an Object-Oriented Database System. 740-749. 1994. Morgan Kaufmann Publishers Inc. Proceedings of the 20th International Conference on Very Large Data Bases.

[9] International Organization for Standardization. ISO/IEC 13249-1 Information Technology - Database Languages - SQL Multimedia and Application Packages - Part 1:Framework. 2002.

[10] International Organization for Standardization. ISO/IEC 13249-2 Information Technology - Database Languages - SQL Multimedia and Application Packages - Part 2: Full-Text . 2002.

[11] Oracle Text: Features Overview. 2002. Oracle Corporation.

[12] Microsoft Corporation. Textual Searches on Database Data Using Microsoft SQL Server 7.0. 1999. MSDN.

http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnsql7/html/textsearch.asp .

[13] Under-the-hood of FAST Enterprise Search Solutions. 2002. Fast Search & Transfer ASA.

[14] Fast Search & Transfer ASA. 2004. http://www.fast.no.

[15] Gule Sider. 2004. http://www.gulesider.no.

[16] PostgreSQL 7.3.2 User's Guide. 2003. PostgreSQL Global Development Group.

[17] PostgreSQL 7.3.2 Reference Manual. 2003. PostgreSQL Global Development Group.

[18] Clausen, Håkon, "FAST søkemotor som indeksaksessmetode i en relasjonsdatabase." Institutt for Informatikk, Universitetet i Oslo, 2004.

[19] Elmasri, R. and Navathe, B. S., *Fundamentals of Database Systems*, 3 ed. Addison Wesley, 2000.

[20] Garcia-Molina, H., Ullman, D. J., and Widom, J., *Database Systems: The Complete Book* Prentice Hall, 2002.

[21] Codd, E. F., "A Relational Model of Data for Large Shared Data Banks," *Commun.ACM*, vol. 13, no. 6, pp. 377-387, 1970.

[22] Lane, T. Transaction Processing in PostgreSQL. 2000.

[23] PostgreSQL. 2004. http://www.postgresql.org/.

[24] Stonebraker, M. and A.Rowe, L. The design of POSTGRES. 340-355. 1986. ACM Press. Proceedings of the 1986 ACM SIGMOD International Conference on Management of Data.

[25] Stonebraker, M., Rowe, A. L., and Hirohama, M., "The Implementation of POSTGRES," *IEEE Transactions on knowlege and data engineering*, vol. 2, no. 1, pp. 125-142, 1990.

[26] PostgreSQL 7.3.2 Administrator's Guide. 2003. PostgreSQL Global Development Group.

[27] Korfhage, R. R., *Information Storage and Retrieval* Wiley, 1997.

[28] Belew, K. R., *Finding out about : A Cognitive Perspective on Search Engine Technology and the WWW* Cambridge University Press, 2000.

[29] Berry, W. M. and Browne, M., *Understanding Search Engines : Mathematical Modeling and Text Retrieval* Society for Industrial and Applied Mathematics, 1999.

[30] Search Engine Sizes. 2-9-2003. http://searchenginewatch.com. http://searchenginewatch.com/reports/article.php/2156481

[31] Inktomi. 2004. http://www.inktomi.com/.

[32] AllTheWeb. 2004. http://www.alltheweb.com/.

[33] Stochasto. 2004. http://www.stochasto.com/.

[34] Risvik, M. K., Mikolajewski, T., and Boros, P. Query Segmentation for Web Search. 2003. Proceedings of the 12th International World Wide Web Conference. http://research.yahoo.com/publications/16.pdf

[35] Yahoo! 2004. http://www.yahoo.com.

[36] Startsiden. 2004. http://www.startsiden.no/.

[37] Vivísimo Clustering Engine. 2004. http://vivisimo.com/.

[38] Nelson M.Mattos. SQL99, SQL/MM, and SQLJ: An Overview of the SQL Standards. 2000. IBM Database Common Technology.

[39] Melton, J. and Eienberg, A., "SQL Multimedia and Application Packages (SQL/MM)," *SIGMOD Rec.*, vol. 30, no. 4, pp. 97-102, 2001.

[40] Oracle Text, Oracle Technical White Paper. 2002. Oracle Corporation.

[41] Hellerstein, M. J. GiST: A Generalized Search Tree for Database Systems. 1996.

[42] Hellerstein, M. J., Naughton, F. J., and Pfeffer, A. Generalized Search Trees for Database Systems. 1995. 21st International Conference on Very Large Data Bases.

[43] GiST for PostgreSQL. 2004. http://www.sai.msu.su/~megera/postgres/gist/.

[44] Barunov, O., Demetriou, N., Sigaev, T., and Wickstrom, D. OpenFTS Primer. 2002.

[45] OpenFTS. 2004. http://openfts.sourceforge.net/.

[46] Paul, M. A., "Implementation of Extended Indexes in POSTGRES," *SIGIR Forum*, vol. 25, no. 1, pp. 2-9, 1991.

[47] PostgreSQL 7.3.2 Programmer's Guide. 2003. PostgreSQL Global Development Group.

[48] PostgreSQL 7.3.2 Developer's Guide. 2003. PostgreSQL Global Development Group.

[49] FAST Data Search Integration Guide. 2003. Fast Search & Transfer ASA.

[50] Indexing Database Content and XML. 2003. Fast Search & Transfer ASA.

[51] DeWitt, D. and Gray, J., "Parallel Database Systems: The Future of High Performance Database Systems," *Commun.ACM*, vol. 35, no. 6, pp. 85-98, 1992.