UNIVERSITETET I OSLO
Institutt for Informatikk

FAST søkemotor som indeksaksess-metode i en relasjonsdatabase

Hovedoppgave

Håkon Clausen

Februar 2004

Abstract

Integrating information retrieval in relational databases has been a long recognized goal. Still, databases lack support for advanced free text search and information retrieval functionality. At the same time, search engines are some of the most advanced and most rapidly developing information retrieval systems.

In this thesis we use one of the most advanced search engines available today, FAST Data Search, to add information retrieval support to a relational database, namely PostgreSQL. Implemented as a simple index access method, we explore different possibilities and discuss future solutions for doing this type of integration.

The rest of this thesis is written in Norwegian.

Innholdsfortegnelse

Kapittel 1. Innledning

Store mengder av den informasjonen vi har tilgjengelig i dag er digitalisert og lagret på datamaskiner rundt om i verden. Spesielt tilgangen til raskere og billigere kommunikasjonsteknologi fører til at informasjon blir stadig lettere tilgjengelig. Dagens digitaliserte informasjon finnes ikke bare som websider på Internett, men er også lagret i store databasesystemer. Informasjonen i disse systemene brukes sjelden til annet enn sin primæroppgave, som er persistent lager for programsystemer. Nyttegraden av slik lagret informasjon som ikke kan brukes til annet enn sin primærapplikasjon er lav, og muligheten til å finne informasjon i slike systemer er derfor et ettertraktet mål.

> *I denne oppgaven gjør vi bruk av "state of the art" søketeknologi for å tilføre relasjonsdatabasesystemer egenskaper til et informasjonsgjenfinningssystem.*

Oppgaven fokuserer på følgende konkrete problemstillinger:

1. Jeg ønsker å se om en søkemotor kan tilføre en relasjonsdatabase fritekstsøkemuligheter og i hvilken grad dette lar seg integrere. Dette er en mulighet som tradisjonelt er meget begrenset, men som det i de senere år har vært stor interesse for. Flere av de store databaseleverandørene tilbyr dette, men det kan se ut som om søkemotorer har kommet lenger i utviklingen av denne typen systemer, spesielt med fokus på effektivitet og skalerbarhet. En integrasjon er derfor potensielt en god måte å tilføre relasjonsdatabaser fritekstsøk.

2. En søkemotor kan klassifiseres som et informasjonsgjenfinningssystem[1]. Likhetene mellom et databasesystem og et IR-system er flere, og systemene har klar nytte av funksjonaliteten til hverandre. Jeg ønsker derfor å bidra til å kartlegge sammenhengen mellom klassiske IR-systemer, søkemotorer og databasesystemer. Hvorvidt en integrasjon er vellykket eller ikke, avhenger av at søkemotoren kan tilføre oss den ønskede funksjonaliteten.

3. Søkemotorer tilbyr ulike "ad-hoc" spørrespråk for å gjøre fritekstsøk, mens relasjonsdatabaser har et veletablert spørrespråk, nemlig SQL. I et integrert system må også disse spørrespråkene integreres. Jeg ønsker å se på hvordan dette skal gjøres.

For å belyse problemstillingene har vi implementert en prototyp der vi bruker søkemotoren FAST Data Search (FDS) som en indeksaksessmetode i relasjonsdatabasen PostgreSQL.

Integrering av IR-funksjonalitet i relasjonsdatabaser har vært gjort både i forskning og kommersielt. I denne oppgaven gjør vi imidlertid dette på en ny måte, ved å benytte oss av kommersiell søketeknologi til integrasjonen. Vi ønsker å belyse hva dette gir oss i forhold til andre implementasjoner.

[1] Information Retrieval System (IR-system)

1.1. Dokumentets oppbygning

Resten av oppgaven inneholder følgende:

Kapittel 2

Gir en introduksjon og motivasjon til tema og ideen om et integrert system. Det følger også en oversikt over tidligere arbeider.

Kapittel 3

Gir en kort introduksjon til databasesystemer generelt og relasjonsdatabaser spesielt, som er nødvendig for å forstå integrasjonen.

Kapittel 4

Dette kapittelet gir en grundig innføring i fritekstsøk med hensyn på funksjonalitet og oppbygning til et IR-system. Denne introduksjonen er lang, men dette er nødvendig for å forstå forskjellen mellom fritekstsøk og databasesøk, og dermed hensikten med å integrere den i en relasjonsdatabase. Enda viktigere er den for sammenligningen mellom IR-systemer og søkemotorer som følger i neste kapittel, for å kunne vurdere om en søkemotor er hensiktsmessig å bruke i en integrasjon.

Kapittel 5

Her gis en introduksjon til FDS og en analyse av funksjonaliteten i denne. Videre gjennomgås eksisterende fritekstløsninger fra de store kommersielle databaseleverandørene, samt standarden SQL. Deretter sammenlignes disse systemene for å avgjøre om søkemotoren kan brukes til en integrasjon.

Kapittel 6

Her gjennomgås implementasjonen av prototypen der søkemotoren FDS integreres som en ekstern indeksaksessmetode i relasjonsdatabasen PostgreSQL. Resultat fra en test av implementasjonen presenteres også.

Kapittel 7

I dette kapittelet diskuterer jeg hvorvidt integrasjonen var vellykket, samt andre sider ved problemstillingen. Jeg vil også peke på begrensninger i søkemotoren og implementasjonen som gjør at integrasjonen ikke er fullstendig, samt foreslå mulige metoder for å supplere disse manglene.

Kapittel 8

Konkluderer oppgaven og trekker opp noen retninger det ville være interessant å jobbe videre med.

1.2. Arbeid med prototyp

Implementasjon av prototypen har vært et samarbeid mellom meg og Åsmund Kveim Lie. Dette implementeringsarbeidet har vært såpass omfattende at det ville ha vært mye å gjennomføre alene. Kompleksiteten ligger egentlig ikke i omfanget av resultatet, men at det er to omfattende systemer som skal integreres med hverandre. FDS er et meget stort system som vi har måttet tilegne oss detaljkunnskap om for å kunne bruke. PostgreSQL er et stort databasesystem bestående av en halv million kodelinjer som ikke er meget godt dokumentert, og det er innenfor dette systemet det meste av implementasjonen har vært gjort.

Resultatet av denne implementasjonen har bidratt til forståelse av problemstillingene og til diskusjonen som kommer senere i denne oppgaven.

1.3. Takk til

Jeg vil gjerne takke veilederen min, Knut Omang, for all hjelp under arbeidet med denne oppgaven.

Jeg vil også rette en spesiell takke til Åsmund Kveim Lie, som har vært min samarbeidspartneren under det praktiske arbeidet. Takk også til alle andre som har lest oppgaven og kommet med innspill og rettelser.

Kapittel 2. Bakgrunn og motivasjon

2.1. Motivasjon

2.1.1. Viktigheten av søketeknologi

"A wealth of information creates a poverty of attention"

Nobelprisvinner i økonomi, Herbert Simon.

I dagens samfunn er tilgangen på informasjon stor og stadig økende. Utviklingen av Internett, personlige datamaskiner, tilgang til kommunikasjonsteknologi og en økende mengde digitalisert informasjon er med på å påvirke denne utviklingen. Som en følge av at mengden informasjon øker, blir det stadig vanskeligere å finne frem til den informasjonen man har behov for. Problemet i dag er ikke tilgang på informasjon, men mengden informasjon. Det er derfor viktig å kunne lokalisere og filtrere ut den relevante informasjonen, og det er derfor ikke tilfeldig at de mest populære sidene på Internett tilhører søkemotorer som har som oppgave å hjelpe folk med å finne den relevante informasjonen og unngå resten[1]. Søketeknologi blir stadig viktigere både for privatpersoner og selskaper. Fokus på søketeknologi og informasjonssystemer har økt kraftig de siste årene, men det har også vært et aktivt forskningsfelt i over 40 år. Det er imidlertid i de siste årene at slik teknologi har blitt gjenstand for særlig kommersiell interesse.

Det er flere grunner til at søketeknologi er viktig for selskaper. For det første er det viktig at de blir funnet av de kundene som selskapet retter sin virksomhet mot. Som sitatet over uttrykker, skaper informasjonsmengden vanskeligheter for selskaper når det gjelder å få oppmerksomhet hos kundene. For kundene er søketeknologi også viktig, for at de skal kunne finne den informasjonen hos selskapet som de er på jakt etter. Dette kan være kritisk for et selskap som selger produkter over Internett. La oss for eksempel si at man gjør et søk på ordet 'laptop' på IBM's hjemmesider. Hvis man ikke finner informasjon om deres bærbare PC "ThinkPad", vil man sannsynligvis vurdere å gå til en konkurrent.

For selskaper er det imidlertid vel så viktig å kunne utnytte den store mengden informasjon de har internt. En rapport fra "The Yankee Group" i desember 2002[1] viser at 75% av amerikanske bedrifter med mer enn 100 ansatte benytter seg av en eller annen form for søketeknologi. Dette tallet er i tillegg sterkt økende, og samme rapport anslår at tallet vil være over 90% i 2005. Fremgangen til teknologiselskaper som utvikler denne type systemer, slik som FAST[2] og Autonomy[3], bekrefter denne utviklingen. Viktige egenskaper ved søketeknologi er ytelse, skalerbarhet og raske oppdateringer slik at informasjon raskt blir tilgjengelig.

En annen grunn til at søketeknologi er viktig, er spredningen av informasjon. Søkebehovet går på tvers av både formater og systemer. Vanligvis vil informasjon være

[1] ComputerWeekly.com, 16.01.2003
[2] Fast Search & Transfer ASA, www.fast.no
[3] Autonomy Systems, www.autonomy.com

lagret både i databaser, filsystemer og på Internett eller intranett. Det er derfor vanskelig å holde oversikten over informasjonen, samt å lokalisere den uten søketeknologi.

2.1.2. Databasens behov for fritekstsøk

Tradisjonelle relasjonsdatabasesystemer (RDBMS) er beregnet på lagring av små, mange og strukturerte informasjonselementer (entiteter). Typiske bruksområder er lagring av persondata, forretningsdata og en mengde andre områder. Populariteten til relasjonsdatabaser har vært kraftig stigende de siste 20 årene. Til tross for at nye modeller og teknologier for datalagring har blitt utviklet, har relasjonsmodellen slik den ble definert av Tedd Codd i 1970 (se 3.1), forsatt å være den mest benyttede måten å lagre data på. I nær tilknytning til relasjonsdatabaser finner vi spørrespråket SQL som nær sagt alle relasjonsdatabaser implementerer.

Bruksområdet til relasjonsdatabaser har ekspandert fra det opprinnelige, til nær sagt alle mulige typer datalagring. Typiske eksempler på dette er CAD/CAM[1] systemer, geografiske systemer og informasjonssystemer. Det er flere grunner til dette:

- Det kan være hensiktmessig å lagre strukturert informasjon tilknyttet den informasjonen systemet primært skal lagre, i det samme system. La oss f.eks. ta et digitalt bibliotek. I tillegg til selve informasjonsobjektet, for eksempel en bok, ønsker man å lagre informasjon om forfatteren av boken. Dette er data som vanligvis vil være normalisert i en relasjonsdatabase. Grunnen til å lagre dette i samme system kan både være økonomisk, fordi man kun trenger å drifte ett system, men kan også være praktisk, fordi man kun trenger å kjenne til ett applikasjonsgrensesnitt.
- Ettersom man får nye informasjonslagringsbehov, er det enklere å integrere og utvide et allerede eksisterende databasesystem, istedenfor å sette opp et nytt.
- Relasjonsdatabaser har vært og er et forskningsfelt som utvikler seg stadig. Dagens databaser har mange funksjoner som er ønskelige for alle typer lagringssystemer, som for eksempel samtidighetskontroll, transaksjonsstøtte, og gjenoppretting. Relasjonsdatabaser har dessuten et vel fundamentert abstraksjonslag, datauavhengighet og spørrespråk.

Alle disse punktene gjør at relasjonsdatabaser også brukes til å lagre tekst, uten at de tradisjonelt er egnet for dette. Interessen og behovet for å integrere fritekstsøk i databaser har i likhet med interessen for søketeknologi vært sterkt økende. Forskning fra tidlig på 90-tallet har resultert i flere kommersielle implementasjoner i de største relasjonsdatabasene[2]. Også SQL-standarden har blitt utvidet med støtte for tekstsøk gjennom SQL/MM: FullText[3], men denne er foreløpig ikke støttet av noen av de kommersielle aktørene.

2.1.3. Informasjonssystemers behov for databaser

Et informasjonssystem er et system som kan lagre, hente, søke og vedlikeholde informasjon i form av f.eks. lyd, bilde eller tekst[4]. I denne oppgaven fokuserer jeg på tekst, og det er også her utviklingen av disse systemene har kommet lengst. Hensikten med et informasjonssystem er å minimere den tiden man bruker på å lokalisere den

[1] Computer Aided Design / Computer Aided Manufacturing. Programvare brukt i kunst, arkitektur, ingeniørfag og til produksjon.

informasjonen man er på jakt etter. Det vil si at man finner mest mulig av den relevante informasjonen i systemet, uten å måtte lete gjennom irrelevant informasjon. Dette er et betydelig fagfelt som har vært særlig viktig de siste årene. Jeg vil introdusere dette i Kapittel 4.

Et informasjonssystem skiller seg fra et databasesystem fordi databasesystemer er optimalisert for strukturerte data, mens tekst, lyd og bilder i hovedsak er ustrukturert. Det er imidlertid i prinsippet ingen ting i veien for at et informasjonssystem skulle kunne implementeres ved hjelp av et databasesystem[5].

Det finnes en rekke grunner til at en integrasjon med databaser er ønskelig fra de klassiske IR-systemenes synspunkt:

- Mange av dagens kommersielle IR-systemer er dårlig fundamentert i datamodeller, og har som oftest et spørrespråk som er lite uttrykkskraftig. Når brukere ber om mer funksjonalitet, løses dette som oftest med ”ad-hoc”-utvidelser som ikke har noe teoretisk fundament[6]. Databaser derimot, tilbyr et anerkjent og kraftig høynivå-spørrespråk, og en solid teoretisk fundamentert datamodell. På områder som sikkerhet, har dagens rene IR-systemer vanligvis liten eller ingen systemstøtte.
- Bruksområdet til et IR-system er smalt, og mange av de applikasjonene som involverer tekst, har også behov for typisk RDBMS-funksjonalitet. Et digitalt bibliotek, for eksempel, har i tillegg til å lagre og søke i store mengder informasjon, også behov for å holde oversikt over låntakere og utlån. Det siste er et område hvor man vil måtte benytte en vanlig database, fordi det er snakk om strukturerte data som krever samtidighetskontroll og transaksjonsstøtte. Mange av egenskapene i en relasjonsdatabase er også av interesse i systemer som kun inneholder ustrukturerte data, slik som samtidighet og gjenopprettbarhet. Ved en fullstendig integrasjon slipper man oppdateringsproblemer og push/pull-problematikk[1] som man ville hatt hvis man valgte å holde dem separat. Det ville også vært mye vanskeligere å gjennomføre de egenskapene som er nevnt over.
- Mange av dagens IT-systemer er bygd på en relasjonsdatabase. De fleste selskaper har behov for å utvide anvendelsen av søk, både for sin egen del og for å kunne tilby kunder bedre og mer informasjon, men de ønsker ofte ikke å gjøre noe med disse gamle systemene. Ved å integrere et IR-system i disse relasjonsdatabasene, vil man kunne tilby søk i applikasjoner man ikke ønsker å endre på. I mange tilfeller tilbyr slike systemer søk allerede, men disse skalerer sjelden godt. Ettersom databaser stadig vokser, ønsker man å benytte seg av IR-teknologi for å øke hastigheten på søk.

[1] Push/pull problematikk er et ”klassisk” problem hvis man ønsker å holde to separate systemer oppdatert i forhold til hverandre. I dette tilfelle kunne man gjøre innholdet i en database søkbart enten ved å ”pushe” nytt innhold fra databasen til søkemotoren, eller ved at søkemotoren ”puller” databasen med jevne mellomrom for å kontrollere om det er ny tekst som må gjøres søkbar.

- For en applikasjonsprogrammerer som har behov for IR-funksjonalitet, har databaser en klar fordel fremfor tradisjonelle IR-systemer ved at de er *datauavhengige*. Datauavhengighet vil si at systemet skjuler de fysiske lagringsdetaljene fra brukeren og at det er systemet som optimaliserer uthenting av data. Brukeren, det vil si applikasjonsprogrammereren, vil kunne bruke et grensesnitt og et spørrespråk for å hente ut dataene. I et tradisjonelt IR-system vil man som regel trenge inngående kjennskap til hvordan data fysisk er lagret for optimalisering av søk og uthenting av data.

Alle disse momentene er med på å gjøre at en integrasjon mellom IR- og databasesystemer ønskelig. Objektorienterte databaser (OODB) var lenge ansett som den databasetypen som skulle brukes til applikasjonsområder som de relasjonelle databasene ikke var beregnet for. Utviklingen av disse har imidlertid stagnert de siste årene, og effektiviteten er ikke god nok for å støtte fritekstsøk. Det mest naturlige er derfor å integrere med en relasjonsdatabase, som er det klart mest utbredte databasesystemet i dag.

2.1.4. Argumentasjon for integrasjon

Informasjonssystemer og databasesystemer har lenge vært to atskilte felt, både forskningsmessig og kommersielt. De deler allikevel en god del egenskaper rent praktisk. Mange IR-applikasjoner, slik som digitale biblioteker, er derfor ofte implementert ved hjelp av kommersielle databaser[7].

Det som ofte skiller et informasjonssøkesystem fra et databasesystem, er datamodellen. I et informasjonssystem finnes det kun en entitet, dokumentet. Med en relasjonsmodell kan man fritt definere mange typer entiteter og man kan dynamisk opprette relasjoner mellom disse. Det at en relasjonsdatabase tilbyr en mer avansert datamodell, skulle ikke være til hinder for at et informasjonssystem skulle kunne representeres i en relasjonsdatabase. Spørsmålet er imidlertid om det er egnet og om det er effektivt nok. Et informasjonssystem bygger som oftest på inverterte filer (se avsnitt 4.7.1) for å oppnå effektive søk, og denne type datastrukturer mangler ofte relasjonsdatabaser. Det er imidlertid ingen grunn til at en relasjonsdatabase skal mangle dette. Det er heller ingen grunn til at en relasjonsdatabase med en velfungerende spørreoptimaliserer skal ha dårligere yteevne enn et rent invertert filsystem. Et databasesystem (DBMS) vil faktisk kunne optimalisere bruken av inverterte filer bedre enn man ofte vil kunne gjøre manuelt. Det er nettopp dette som er gevinsten av datauavhengighet.

Det finnes enkelte som mener at en slik integrasjon ikke er hensiktsmessig[8]. Argumentasjonen handler i hovedsak om at bruken av SQL for å gjøre fritekstspørringer mot en normalisert database vil gjøre spørringene alt for komplekse. Det å bruke SQL som et grensesnitt mot et integrert IR- og databasesystem betyr, imidlertid ikke at sluttbrukeren må bruke SQL for å gjøre søk i systemet. Et standardisert grensesnitt betyr derimot at man kan lette implementasjonen av brukervennlige grensesnitt som kan brukes på toppen av slike integrerte systemer[6].

Både tidligere forskning[2, 7, 9-11], standarder[3] og kommersielle implementasjoner[12-14] bekrefter at en integrering av IR- og databasesystemer er svært interessant. Allikevel har mange informasjonssystemer, deriblant mange systemer som har til hensikt å gjøre effektive fritekstsøk, ikke vært implementert ved hjelp av databaser. Mye av forskningen rundt fritekstsøk har også vært atskilt fra databasesystemer, og dette har ført til at forskningen innen dette feltet har kommet mye kortere på databasesiden enn innenfor

tradisjonelle informasjonssøkesystemer. Spesielt når det kommer til skalerbarhet, noe som er svært viktig i et effektivt fritekstsøkesystem, har DBMS'ene kommet kortere.

Sett fra de kommersielle databaseselskapenes synspunkt er integrasjon av fritekstfunksjonalitet meget interessant, og dette vil utvide deres marked betraktelig. De ønsker med andre ord å tilby noe som søkemotorer har tilbudt i en årrekke.

Et eksempel på et interessant bruksområde er søk i strukturerte data. Hvis man skal søke i en database, krever det kjennskap til den underliggende strukturen, noe som ikke alltid er tilgjengelig for en bruker. Allikevel kan det være interessant å søke i databasen, nettopp fordi en del av dataene er ustrukturerte. Resultat fra søk i de strukturerte dataene uten kjennskap til den underliggende strukturen kan dessuten presenteres med tilhørende informasjon, nettopp fordi databasen kjenner den underliggende strukturen.

Hvis man ser på de forskjellige integrasjonsforsøkene som er gjort mellom IR og relasjonsdatabaser hittil, kan man i tillegg konkludere med at systemer som tar utgangspunkt i databaser, ikke tar nok hensyn til usikkerheten og vagheten i IR, mens systemer som gjør dette, har en meget svak datamodell og et begrenset spørrespråk[6].

I denne oppgaven vil jeg derfor forsøke å bruke en kommersiell søkemotor for gi en relasjonsdatabase fritekstsøkfunksjonalitet. Det vil være et forsøk på å ta det beste fra begge leire, noe som vil resultere i et system som har databasens fordel med en datamodell og et spørrespråk, mens den bevarer IR-systemets usikkerhet og vaghet når det gjelder fritekstsøk og som, på grunn av en søkemotors implementasjon, vil skalere godt.

Som antydet i problemstillingen er et av de første spørsmålene som melder seg ved en slik integrasjon, hvorvidt en søkemotor kan tilby den mest sentrale IR-funksjonaliteten. Dette bør ligge på et slags middels nivå i forhold til hva dagens informasjonssøkesystemer kan tilby. Jeg vil derfor i Kapittel 4 oppsummere en del av denne funksjonaliteten og i Kapittel 5 sammenligne den aktuelle søkemotoren, eksisterende IR-implementasjoner i databaser og den nye SQL/MM-standarden, for å avdekke om søkemotoren har vesentlige mangler i forhold til den IR-funksjonalitet som man vil forvente å finne i en database.

Deretter vil jeg presentere en implementasjon av et slik integrert system mellom søkemotoren FAST DS og relasjonsdatabasen PostgreSQL. Ut i fra forståelsen av denne implementasjonen, vil jeg så evaluere hvor fullstendig en slik integrasjon vil kunne være. Full integrasjon bør inkludere[2]:

1. Støtte for lagring, indeksering og oppdatering i dokumenter og søk i disse.
2. Transaksjonssemantikk, det vil si at operasjoner på dokumenter i systemet er atomiske, konsistente, isolerte og varige.
3. Samtidig oppdateringer og opphenting av dokumenter
4. Utvidelse av databasespørrespråk for å støtte rangering av dokumenter og søkeoperasjoner.
5. Skalerbar ytelse på dokumentindeksering og søk

Diskusjonen rundt disse punktene vil ligge til grunn for konklusjonen i Kapittel 8.

2.1.5. Andre applikasjonsområder for et integrert system

Fordi en integrasjon tilfører en relasjonsdatabase skalerbare og effektive tekstsøk, kan man tenkte seg flere applikasjonsområder. Jeg kommer ikke til å diskutere disse noe videre i oppgaven, men nevner dem her for å understreke at bruksområdene er mange.

Datavarehus er store databaser som gjerne er bygget opp med data fra flere mindre databaser. Hensikten med datavarehus er å samle store mengder data som man deretter kan gjøre analyser av. Systemer som gjør dette kalles gjerne OLAP[1]- og DSS[2]-applikasjoner. På grunn av mengden data er det et opplagt behov for skalerbare og effektive søk. I tilknytning til datavarehus finner vi også ”Data Mining”. Dette fagområdet dreier seg om å finne ukjente sammenhenger i store mengder data, i motsetning til OLAP- og DSS-systemer som benytter seg av den allerede kjente strukturen i databasen for å analysere data. Siden IR har en rekke teknikker for å finne frem til relevant informasjon, er det interessant for slike systemer å bruke IR-teknikker i databaser og datavarehus. I tillegg til å utføre ”Data Mining” på typiske relasjonelle data, kan man benytte seg av IR for å gjøre det på ustrukturerte data, det vil si tekstdokumenter.

Et annet område som har hatt mye fokus de siste årene, er digitale bibliotek. Konseptuelt er dette en analog til det tradisjonelle biblioteket, det vil si store samlinger med informasjon lagret på forskjellige typer media, for eksempel tekst og lyd som er digitalisert. Et digitalt bibliotek byr på en rekke utfordringer; effektive og skalerbare fritekstsøk er opplagt en av dem. Mulighet for søk i relasjonsdatabaser gir oss bedre anledning til å bruke nettopp dette som grunnlag for et digitalt bibliotek.

[1] On-Line Analytical Processing
[2] Decision-Support Systems

2.2. Tidligere arbeider

Jeg vil i dette avsnittet se på noen artikler og arbeider som har vært relevant i forhold til problemstillingen og spesielt i forhold til den implementasjonen som vi har gjort. Jeg vil i tillegg til å beskrive arbeidene også forklare hvilke elementer jeg har benyttet meg av i denne oppgaven og hvordan arbeidet har blitt brukt.

Det er skrevet mange artikler og gjort mye forskning på dette temaet; effektivisering av fritekstsøk i databaser. Tildels svært ulike fremgangsmåter og metoder har vært brukt. En liten oversikt over slike metoder (eller modeller) er beskrevet i 4.2. Jeg vil begrense meg til de mest relevante arbeidene som baserer seg på utvidelser av relasjonsdatabaser. Utvidelsene er innenfor eksisterende aksessmetoder, nye aksessmetoder, eller bruk av SQL og indekser til å effektivisere søk. Spesielt arbeid som er gjort i PostgreSQL har vært interessant, fordi vi ønsket å bruke denne relasjonsdatabasen som grunnlag for vår prototyp.

Implementation of extended indexes in Postgres[7]

> Artikkelen presenterer en implementasjon av en modifisert aksessmetode i PostgreSQL for å støtte fritekstsøk.

Integrating IR and RDBMS Using Cooperative Indexing[2]

> Artikkelen foreslår og presenterer en implementasjon for å integrere IR-funksjonalitet i en relasjonsdatabase. Det foretas endringer i en Oracle database og aksessmetoder for å vise dette.

Tsearch[15]

> Dette er en indeksaksessmetode i PostgreSQL for effektive fulltekstsøk som følger med distribusjonen til databasen.

Efficient Transaction support for Dynamic Information Retrieval Systems[9]

> Artikkelen ser på transaksjonshåndtering for et integrert database- og IR-system og foreslår modifikasjoner i den tradisjonelle transaksjonshåndteringen til relasjonsdatabaser for å kunne håndtere IR-data.

Oracle Text m.fl.[14, 16, 17]

> Flere kommersielle databaseleverandører har implementert fritekstsøk i sine produkter, deriblant Oracle, IBM og Microsoft. Disse systemene vil jeg forklare under 5.2.2, der jeg sammenligner disse systemene med FDS og SQL/MM.

SQL/MM[3, 18]

> Dette er et tillegg til SQL-standarden som blant annet definerer et fulltekstsøkespråk til SQL. Dette vil jeg gå nærmer inn på i 5.3.

De fire første arbeidene vil jeg nevne allerede her, men Kapittel 3 og Kapittel 4 kan med fordel leses først hvis man er ukjent med en del begreper og teori om relasjonsdatabaser og IR.

2.2.1. Implementation of extended indexes in Postgres

I denne artikkelen viser forfatteren for det første hvordan man implementerer nye aksessmetoder i Postgres. Det argumenteres for at fordi databasesystemer som Postgres kan utvides med nye aksessmetoder, er det ingen grunn til at en relasjonsdatabase ikke skulle kunne brukes til å implementere et informasjonssøkesystem, og at det heller ikke er noen grunn til at et slikt system skal ha betydelig dårligere ytelse enn et IR-system basert på inverterte filer. Fordelene ved å bruke en relasjonsdatabase er blant annet datauavhengighet, noe som gir programmereren mulighet for å skille mellom den fysiske og den logiske lagringsmåten.

For å muliggjøre effektive fritekstsøk benytter forfatteren seg i all hovedsak av en B-tre-indeks, men i stedet for å indeksere hele teksten som en enhet, det vil si å legge hele teksten inn som nøkkelen i treet, indekserer han alle ord hver for seg. Dette gjøres ved å implementere en egen aksessmetode, (se aksessmetoder i 3.3) kalt FB-tre (funksjonelt B-tre), som er en påbygning på den eksisterende B-tre implementasjonen, eller egentlig et mellomlag mellom aksessmetoden og databasen. Når aksessmetoden mottar en tekst som skal settes inn, brytes denne teksten ned til ord som settes inn i treet. På denne måten gjøres ingen endringer i PostgreSQL eller i B-tre implementasjonen. Spørringen kan deretter foregå på denne måten:

```
SELECT title FROM reports r
WHERE extract(r.abstract) @= "database"
```

Her er '@= ' en ny operator hvor venstre argument er en ny type; nøkkelord, som blir generert av funksjonen extract. I de tilfeller der det finnes et FB-tre indeks på *abstract*, vil denne aksessmetoden velges. Et FB-søk vil foregå på nøyaktig samme måte som oppslag i et vanlig B-tre, fordi søket vil skje etter nøkkelord generert av *extract*, så ingen ekstra modifikasjoner er nødvendige.

Forfatteren viser hvordan det er mulig å lage en ny aksessmetode i Postgres, og i stor grad benytter vi oss av de samme prinsippene når vi lager en ny aksessmetode med vår eksterne indeks, selv om denne artikkelen bruker en langt eldre versjon av Postgres der blant annet spørrespråket PostQUEL ble benyttet istedenfor SQL. Selve løsningen er også svært primitiv med svært dårlige muligheter for fritekstfunksjonalitet, avanserte spørringer og skalerbar IR-ytelse.

2.2.2. Integrating IR and RDBMS Using Cooperative Indexing

Forfatterne av denne artikkelen fokuserer på fordeler ved å integrere et informasjonssøkesystem med en relasjonsdatabase, og mange av de fordeler jeg har nevnt i avsnitt 2.1.4 brukes i argumentasjonen. Forfatterne viser deretter to måter å implementere fulltekstsøk i en database på. Den første baserer seg på eksisterende funksjonalitet i Oracle DBMS, som er deres testdatabase, og den andre på modifikasjoner de gjør basert på tester av den første metoden.

For å karakterisere flaskehalsene i et IR-system basert på en database, gjør de noen innledende tester som baserer seg på bruk av en invertert indeks (se 4.7.1). Dette gjør de i databasen ved å definere to tabeller, en for dokumentene som skal indekseres, og en med den inverterte indeksen som ser slik ut:

```
DOCINDEX(keyword text, DocumentID int, frequency int, occlist
text)
```

Denne inneholder da nøkkelordet det skal søkes etter, en peker til dokumentet som inneholder dette, et tall på hvor mange ganger nøkkelordet forekommer innenfor dokumentet og opplysninger om hvor i dokumentet ordet forekommer, slik at man kan gi dokumenter der ordene er nær hverandre høyere rangering. Tabellen med den inverterte indeksen ble indeksert med et vanlig B-tre for hurtig aksess. Forfatterne gjør tidsmålinger både på innsetting og søk, og måler tiden på hvert av søkene med forskjellig mengde data lastet inn. De ønsker å kartlegge hvorvidt tidsøkningen er lineær, fordi dette avgjør om transaksjonen er skalerbar eller ikke. Resultatene av deres innledende tester ble at for en multisegmentindeks[1] var tiden det tok å laste inn data ikke-linær ettersom data økte. Det var også klart at en multisegmentindeks var den beste løsning å gjennomføre spørringer på for å kunne beregne rangering av dokumentene, fordi man da kunne klare seg med data fra indeksen hvis man indekserte både atributtene *text* og *frequency*. Hentet de derimot opp *occlist* fra disk, økte tiden ganske betraktelig.

For å overkomme disse svakhetene implementerte de en indeks som kunne lagre også forekomstlisten innefor den fysiske indeksen, slik at det ikke var nødvendig med diskoppslag. I tillegg implementerte de en INDEX ONLY-konstruksjon som gjorde at de kunne lage en tabell hvor lagringer forgegikk kun i den fysiske indeksen. En vanlig relasjon vil være unødvendig når man har alle data i indeksen.

Videre argumenterer de for det de kaller ”cooperative” indeksering, det vil si at på grunn av forskjellige typer data som potensielt bør kunne indekseres, det være seg bilde, lyd, video og så videre, bør applikasjonen og databasen være kollektivt ansvarlig for indeksering samt tolking av dets innhold. Denne delen er mindre interessant for vårt vedkommende fordi vårt ønske er å implementere en teknikk som er helt uavhengig av applikasjon og som er transparent for brukeren av en relasjonsdatabase. Deres funn og betraktninger omkring problemer ved å representere en invertert indeks fysisk i en database er imidlertid interessante, noe som resulterer i implementasjon av et modifisert B-tre og en mulighet for å lage en INDEX ONLY tabell. Her vil databasen ikke lagre dokumentene i en tabell, men data vil ligge fysisk i en indeks.

2.2.3. PostgreSQL, tsearch2

Tsearch er en modul til PostgreSQL som distribueres sammen med databasen. Her benyttes flere av PostgreSQL-utvidbarhetsteknikker, blant annet mulighet for nye datatyper. Modulen tilbyr en ny datatype, *tsvector*, som er søkbar med effektiv indeksstøtte. Vektoren inneholder et sett med unike ord, sammen med informasjon om posisjon i teksten, og eventuell vekting av ordene. Vektoren er optimalisert for søk og oppdateringer. Alle indekserbare tabeller må ha denne søkbare typen som et attributt:

```
Document(document text, vector tsvector)
```

Oppdatering av denne vektoren skjer ved bruk av triggere og funksjoner som kan gjøre om en tekststreng til en tsvector. Vektoren kan indekseres med aksessmetoden GiST[19], som er en implementasjon av et generelt søketre. Denne kan blant annet brukes som et B-tre. I dette tilfellet representeres hver av tsvectorene som et sett bitsignaturer av fast

[1] En multisegmentindeks er en indeks over flere atributter.

lengde, og det er dette som lagres i søketreet slik at vektorene kan søkes i raskt ved hjelp av indeksstøtte.

Tsearch inneholder også funksjoner for blant annet å fjerne stoppord fra vektorene, utvide med forskjellige bøyninger av ord og rangering av dokumenter. I tillegg er det mulighet for dokumentsammendrag og bruk av synonymer.

Tsearch utgjør søkemotoren for OpenFTS[20], et åpent kildekodeprosjekt som har som mål å implementere en søkemotor. OpenFTS er en mellomvare som bruker PostgreSQL og tsearch for lagring og søking i en fulltekstindeks. Parsering av dokumenter, spørreprosessering og generering, samt lingvistiske operasjoner foregår derimot i mellomvaren. Det finnes også muligheter for et dokument "repository" slik at systemet utgjør en fullverdig IR-applikasjon.

Som nevnt bruker denne modulen flere av de utvidelsesmulighetene som finnes i PostgreSQL, muligheter som også vi benytter oss av i implementasjon. Dette programmet var derfor interessant å studere, samt at vi bruker denne i en sammenligning mellom vår egen implementasjon og andre teknikker.

2.2.4. Efficient Transaction Support…

Transaksjonsstøtte i et kombinert IR- og databasesystem er et felt som har fått liten oppmerksomhet. I denne artikkelen foreslås det en effektiv transaksjonsteknikk for IR-systemer som fokuserer spesielt på håndtering av indekser. Først analyseres IR-transaksjoner og hvilke egenskaper som er spesielle for disse. På bakgrunn av dette forslås det en transaksjonsmodell som baserer seg på korte låser på de inverterte filene for å øke samtidigheten, samt en omsortering på operasjoner for å kunne ta hensyn til de siste oppdateringene. Oppdateringstransaksjoner skjer i en mini-batch[21] som tillater å gjenoppta en uferdig transaksjon på et gitt punkt, slik at man ikke trenger å utføre en lang oppdateringstransaksjon på nytt. Gjenoppretting skjer ved en foroverlog (redo-log). Mellomresultatene i mini-batchen lagres på et temporært sted på disk, samt at informasjon om hvilke dokumenter som har blitt prosessert, lagres i en logg. Uferdige transaksjoner gjenopptas fra siste mellomresultat, og når transaksjonen er ferdig, legges de temporære inverterte filene til i databasens inverterte filer.

Siden en oppdateringstransaksjon kan være lang og oppdateringen ikke skrives til databasen før hele er ferdig, vil mange av de siste dokumentene ikke komme med i resultatet av spørringer som utføres samtidig med oppdateringen. For å bedre dette foreslåes det en teknikk der operasjonene kan omsorteres. En oppdateringstransaksjon forteller hvilke inverterte filer som vil bli oppdatert når en pågående transaksjon er ferdig. Hvis det kommer en spørring som er interessert i de samme inverterte filene, omsorterer transaksjonshåndtereren operasjonene og lar oppdateringen skrive til den inverterte filen før lesetransaksjonen. På denne måten øker mulighetene betraktelig for at de siste dokumentene kommer med i spørringen.

Artikkelen gjør imidlertid flere antagelser for å få denne transaksjonsmodellen til, og det er usikkert om alle disse vil kunne gjelde i et kombinert IR-/RDBMS-system. Blant annet antas det at det ikke eksisterer noen logiske avhengigheter mellom dokumenter i systemet. Dette fører til at det ikke kan oppstå noen form for logiske feil, og en transaksjon trenger derfor ikke å måtte avbrytes.

Videre presenterer artikkelen en implementasjon av en prototyp på dette transaksjonssystemet, samt tester med forskjellige typer låser.

Artikkelen har flere meget interessante synspunkter og forslag til transaksjonsmodeller for et integrert system. Flere av disse bruker jeg som grunnlag når jeg i 7.2 diskuterer vår implementasjon i forhold til transaksjoner og utvidelser til denne.

Kapittel 3. Relasjonsdatabaser

3.1. Innledning og historie

Databasesystemer er, i tillegg til operativsystemer, noen av de mest kompliserte og største generelle dataprogrammene vi har i dag. En database er en samling data som man ønsker å bevare over lengre tid, og et databasesystem er et verktøy for å forvalte disse dataene. Et databasesystem skal tillate oss å definere en struktur for de data vi ønsker å lagre, ofte kalt et databaseskjema, samt gi oss mulighet til å legge til, endre og hente ut data fra dette skjemaet. Et databasesystem skal også gi flere brukere tilgang til de samme dataene i databasen samtidig, uten at operasjonene den enkelte bruker utfører kommer i konflikt med hverandre. Til slutt må en database kunne lagre store mengder data, hindre at data går tapt som følge av for eksempel strømbrudd og systemfeil, samt hindre at uvedkommende får tilgang til databasen.

De tidligste databasesystemene kom på 1960-tallet som en følge av at et filsystem, som man da tradisjonelt brukte til å lagre data i, ikke kunne tilby løsninger på de kravene som nevnt over. I disse første databasesystemene ble det brukt ulike modeller for å representere data. Blant de mest brukte var den hierarkiske modellen, som er basert på en trestruktur, og nettverksmodellen, som er en grafbasert modell. Problemet med disse modellene var blant annet at spørrespråket var vanskelig å bruke, selv for enkle oppgaver.

Ted Codd presenterte i 1970 [22], i en meget berømt artikkel, en datamodell der data ble organisert i *relasjoner*. Dette er et logisk abstraksjonslag som skiller mellom hvordan data fysisk og logisk er lagret. Den fysiske lagringsstrukturen kan være meget kompleks, slik at operasjoner på data kan være meget raske. Brukeren av et databasesystem trenger derimot ikke kjenne til dette. Det eneste man trenger å forholde seg til, er relasjoner. Codd definerte også en relasjonsalgebra. Denne algebraen inneholder operasjoner som kan lage nye relasjoner ut i fra eksisterende relasjoner for å besvare en gitt spørring. Operasjonene fungerer på sett av tupler, og inkluderer mengdeoperasjonene union, snitt og differanse, samt de spesielle projeksjon-, seleksjon- og join-operasjonene. Med noen få tillegg kan disse operasjonene utføre det aller meste vi kan tenkte oss i en database, og det viste seg også at de kan implementeres svært effektivt i en datamaskin. Algebraen gjør det dessuten svært enkelt å formulere komplekse spørringer uten noen form for navigering i data. Algebraen består også av et sett regler som gjør at en mengdeoperasjon kan forenkles og effektiviseres på en helt bestemt måte, og dette lar seg også implementere i en datamaskin. Dette har tilsammen ført til at de aller fleste databasesystemer i dag bygger på relasjonsmodellen.

Omtrent alle relasjonsdatabaser implementerer et spørrespråk kalt SQL[1]. Dette er et deklarativt spørrespråk som tilbyr det meste av relasjonsalgebraen, i tillegg til et sett operasjoner for å manipulere data i databasen, samt definere og manipulere et databaseskjema.

[1] Structured Query Language, også kalt SEQUEL

3.2. Transaksjoner

I et databasesystem er det vanlig å gruppere en eller flere operasjoner i en *transaksjon*. Grunnen til dette er ønsket om å utføre en operasjon bestående av flere mindre databaseoperasjoner, som en helhet. Transaksjonen må derfor utføres *atomisk*. I et databasesystem kan det i tillegg foregå flere samtidige transaksjoner, og vi ønsker at en transaksjon skal utføres isolert fra de andre, slik at konflikter som kan oppstå som følge av at flere transaksjoner pågår samtidig, blir tatt hensyn til. I en DBMS er det vanlig å si at en transaksjon må oppfylle ACID egenskapene, som er som følger[23]:

Atomitet (Atomicity)

> Atomitet betyr at en transaksjon skal utføre ”alt eller ingenting”, også i tilfeller der databasen feiler midt i en transaksjon som følge av for eksempel strømbrudd. I disse tilfellene må databasen avbryte transaksjonen og gjøre om alle endringer som hittil er utført slik at tilstanden er det den var før transaksjonen begynte.

Konsistens (Consistency)

> Når en transaksjon er avsluttet, skal databasen være i en lovlig/konsistent tilstand, i henhold til de regler som er definert i databaseskjemaet.

Isolasjon (Isolation)

> Samtidige transaksjoner skal eksekvere som om de var alene. Dette betyr blant annet at en transaksjon ikke skal se verdier fra andre transaksjoner som ikke er ferdige enda. Operasjoner som er i konflikt med hverandre og som er utført av forskjellige transaksjoner, må oppdages og løses.

Varighet (Durability)

> Varighet betyr at effekten av avsluttede transaksjoner aldri må bli borte, selv om databasen feiler eller går ned.

Atomitet må garanteres fordi systemet eller brukeren når som helst kan avbryte en transaksjon, for eksempel fordi en operasjon gjør at databasen kommer i en ulovlig tilstand, eller at brukeren ikke ønsker å fullføre. Et annet eksempel kan være at systemet går ned midt under en transaksjon. Når systemet har kommet opp igjen, sjekkes det for transaksjoner som ikke ble fullført, og operasjonene som tilhørte denne transaksjonen blir gjort ugyldige. Den mest vanlige måten å implementere dette på er ved hjelp av logger. Etter hvert som systemet utfører lese- og skriveoperasjoner, skrives endringene til en logg sammen med informasjon om hvilke transaksjoner som utførte de. Loggen kan enten brukes til å gjøre om operasjoner som har blitt utført (undo-log) eller utføre operasjoner som skulle ha vært utført, men som ikke har blitt skrevet til disk enda (redo-log). Begge teknikkene har fordeler og ulemper som jeg ikke skal gå nærmer inn på, og noen systemer bruker dem begge. I tillegg til å sikre atomitet, oppfylles også varighet ved hjelp av logger, dette fordi vi ved hjelp av loggen kan gjøre alle operasjoner som ikke har vært ferdig utført.

Når flere transaksjoner eksekverer samtidig på et databasesystem kan det oppstå konflikter fordi flere transaksjoner for eksempel skriver og leser til samme data. Vi har flere kategorier med slike konflikter, og de to viktigste er ”skitten les” og ”tapt oppdatering”. Skitten les oppstår når man leser en verdi som er skrevet av en transaksjon som enda ikke er fullført og som derfor kanskje må avbryte. Tapt oppdatering skjer når to

transaksjoner skriver til samme element. Den som kommer sist, vil få sin endring varig, selv om de hadde samme utgangsverdi. På grunn av slike konflikter ønsker vi at transaksjonene skal være isolert fra hverandre. Graden av isolasjon er avhengig av hva slags type transaksjoner som skal kjøre på systemet og hvor stor samtidighet vi ønsker. Et særlig strengt isolasjonsnivå som ofte brukes i databaser er *serialiserbarhet*. Dette betyr at selv om operasjoner til forskjellige transaksjoner overlapper i tid, skal transaksjonene ha den samme effekten som om de hadde vært utført etter hverandre, *serielt* i tid. Vi ønsker nemlig ikke å utføre transaksjoner som *ikke* er i konflikt med hverandre, serielt etter hverandre. Operasjonene kan for eksempel lese og skrive til helt forskjellige elementer i databasen, og en seriell eksekvering av slike vil redusere samtidigheten. Et annet isolasjonsnivå, READ-COMMITED, som blant annet PostgreSQL benytter, tillater at en transaksjon kan se resultatet av andre fullførte transaksjoner, selv om dette i utgangspunktet bryter med serialiserbarhet, fordi disse transaksjonene kan ha startet etter den transaksjonen som leser resultatet.

Serialiserbarhet blir ofte implementert ved hjelp av låser[1]. Hvis ingen nye låser blir tatt etter at man har sluppet andre låser, vil dette garantere serialiserbarhet. Låsene som trengs av en transaksjon blir derfor ikke sluppet før mot slutten av transaksjonen. Ulempen med denne måten å gjøre det på er at det å holde låser lenge, fører til mindre samtidighet. En annen måte å garantere serialiserbarhet på er ved hjelp av tidsstempling, der et tidsstempel på hver enkeltoperasjon sammenlignes med tidstempelet på transaksjonen. På denne måten kontrolleres det at verdien er gyldig for en bestemt transaksjon. En viktig variant av denne teknikken er der man også tar vare på tidligere versjoner av databaseelementer, så lenge det er de korrekte verdiene for en pågående transaksjon. Fordelen med dette er at en transaksjon ikke trenger å avbryte eller vente på grunn av at det har kommet en ny verdi som ikke er gyldig for en gitt transaksjon, den kan bare lese den gamle. Dette fører til høyere grad av samtidighet. PostgreSQL, som er databasesystemet som er brukt i denne oppgaven, benytter seg av denne teknikken.

3.3. Indeksaksessmetoder

For å kunne forstå hvordan man kan integrere et IR-system i en database, er det viktig å vite hvordan en RDBMS lagrer og henter opp data. I en relasjonsdatabase lagres alle data i relasjoner, som igjen lagres fysisk som en fil på disk i et eller annet format. Det finnes mange forskjellige måter å gjøre dette på, og hvilken teknikk man velger, er avhengig av typen data som skal lagres. De fleste databaser har imidlertid en eller flere bestemte måter de gjør dette på. Måten data aksesseres på, det vil si leses inn og søkes i, er imidlertid vel så viktig for ytelsen.

En database har flere typer aksessmetoder for lese inn data fra disk. En aksessmetode er en mekanisme for å lokalisere spesifikke data et bestemt sted på et lagringsmedium, for deretter å lese eller skrive på disse dataene. En aksessmetode beskriver også måten man lokaliserer bestemte data innenfor en større enhet av data eller et datasett, slik som f.eks. en fil. For at en aksessmetode skal være effektiv bør den finne de dataene man er på jakt etter med færrest mulig disk- (I/O) operasjoner, det vil si lese- og skriveoperasjoner på lagringsmediet. Sekundærlageret er betydelig tregere enn CPU og minne, og når man ser på effektiviteten av aksessmetoder og sammenligner dem med hverandre, utgjør andre faktorene en så liten del av den totale tiden at vi nesten kan se helt bort fra annet enn diskoperasjoner.

[1] Låser er en mekanisme som har til hensikt å sikre at kun en prosess jobber på et dataelement av gangen. Andre prosessert må vente for å få låsen til denne blir ledig.

Den enkleste aksessmetoden er sekvensielt søk, det vil si å lese gjennom hele relasjonen til man finner det man leter etter. Gjør man en spørring av typen ”*SELECT * FROM table”* vil man uansett måtte lese inn hele relasjonen. Hvis man derimot har en spørring av typen ”*SELECT * FROM table WHERE a=10”,* og det kun er 1% av tuplene som tilfredsstiller denne betingelsen, vil det være uhensiktsmessig å lese inn hele relasjonen.

For å effektivisere søk bruker man indeksaksessmetoder. En indeks er en fysisk datastruktur som bygges over data fra databasen. Databasesystemet bruker så indeksen for å lokalisere data istedenfor å søke i den opprinnelige relasjonen. De mest vanlige datastrukturene som benyttes til indekser er trær, og da spesielt B+ trær, samt hasher. Med spørringen over vil man f.eks. kunne lokalisere tuplene med kun tre eller fire diskblokkaksesser, mot potensielt hundrevis hvis relasjonen er stor. I indeksen ligger det kun pekere til hvor data er plassert på disk, så diskblokkene som inneholder de dataene man er på jakt etter, må også leses inn. Man behøver imidlertid ikke å lese inn flere enn man er interessert i.

Indekser opprettes på bakgrunn av kunnskap om hva slags type spørringer som gjøres, hva som trenger raskere aksess, og hva slags type system det er. Hvilken type datastruktur man velger er også avhengig av hva slags type data som skal indekseres. I et system med mange transaksjoner kan man ikke opprette indekser på alt fordi indekser er plasskrevende, samt at indeksen må vedlikeholdes. En indeks på en relasjon fører til at søk går mye raskere, mens oppdateringer på relasjonen vil ta litt mer tid fordi databasen også skal oppdatere indeksen. I et OLAP system derimot, der man gjør mange leseoperasjoner, vil man lage mange indekser for å få rask aksess til data.

Forholdet mellom aksessmetoden, samtidighet og gjenopprettbarhet er også meget komplisert. Det har tatt årevis å skjønne interaksjonen mellom disse og det kun for et begrenset antall datastrukturer[2].

3.4. PostgreSQL

Vi har i denne oppgaven benyttet databasesystem PostgreSQL[24] som basis for vårt integrerte system. PostgreSQL er et objektrelasjonsdatabasesystem basert på POSTGRES som ble utviklet ved Berkley-universitetet i California. POSTGRES var et forskningsprosjekt ledet av Professor Michael Stonebraker, og ble presentert gjennom en serie artikler[25-27]. Den første kjørende prototyp kom i 1987, og prosjektet varte frem til 1993. PostgreSQL videreførte dette prosjektet som en åpen kildekodedatabase, og er den mest avanserte av dette slaget i dag. Den tilbyr SQL92/SQL99, samt en rekke annen funksjonalitet som man vil forvente å finne i en moderne kommersiell relasjonsdatabase. Spesielt i de siste tre årene har utviklingen av PostgreSQL skutt fart og antall kodelinjer er nær fordoblet, det vil i dag si over en halv million kodelinjer. Blant utvidelser de siste årene har vi f.eks. MVCC[1] og WAL[2] som gjør databasen ACID kompatibel. Dette vil jeg komme tilbake til senere i oppgaven.

En annen viktig egenskap med PostgreSQL er utvidbarhet. Databasens brukere kan selv utvide systemet med blant annet nye datatyper[28] og funksjoner for å nevne noe. For oss var dette en avgjørende egenskap fordi vi skulle gjøre flere slike utvidelser. Vi kunne heller ikke gjort dette uten tilgang til kildekode. I avsnitt 6.2 vil jeg gå litt nærmere inn på disse egenskapene.

[1] Multi Version Concurrency Control
[2] Write Ahead Log

Kapittel 4. Fritekstsøk

4.1. Karakteristikk av fritekstsøk

Fritekstsøk skiller seg fra tradisjonelle databasesøk på flere områder. Fritekstsøk har egentlig få likhetstrekk med søk i en relasjonsdatabase. Prosessen for å sammenligne søkekriteriene med en bestemt verdi er ikke lenger en enkel prosess. I en tradisjonell database vil de fleste sammenligningsmetodene enten basere seg på *eksakt sammenligning*, der et tall eller en streng enten er lik eller ikke lik det det ble søkt etter, eller *intervall-sammenligning,* der man med en gitt skala kan avgjøre om verdien ligger innenfor den oppgitte avstanden av det du søker etter, f.eks. om et tall er høyere eller lavere enn et annet[29]. Fritekstsøk derimot, og da snakker vi om fritekstsøk i kontekst av et informasjonssystem eller IR-system, dreier seg om å finne de dokumentene som *handler* om det brukeren søker etter. Det er ikke lenger en enkel mønstergjenkjenning, slik som SQL LIKE-funksjonen, men dokumentets mening vi leter etter. Det dreier seg altså om å utføre en *tilnærmet sammenligning*, fordi det ikke er snakk om et absolutt kriterium. Hvis man i et IR-system får treff på dokumenter som ikke er relevante, er ikke det i seg selv galt. I databasesøk vil det være fatalt hvis man fikk gale data i en eksakt eller intervall- sammenligning. For å kunne abstrahere ut handlingen fra en tekst kreves det menneskelige evner, noe som gjør at IR til en viss grad er beslektet med kunstig intelligens (AI), men ingen slike teknikker brukes i noen større grad i dag[4]. Man bruker imidlertid en rekke andre metoder for å hente ut dokumentets handling, og det er dette jeg vil beskrive i dette kapittelet.

Ofte ser man på selve spørringen som et dokument, man forsøker å finne ut hva spørringen egentlig handler om, for så å finne dokumenter som ligner på denne. Denne søkeprosessen kan gjøres på mange forskjellige måter, men alle har en ting til felles. Man forsøker å skille ut de dokumentene som er relevante for brukeren fra dem som ikke er det. I denne sammenheng, og til stor forskjell fra tradisjonell søking, er den samlingen[1] av dokumenter det søkes i, ikke uvesentlig. Hvis alle dokumentene det søkes i handler om det samme, må man tillegge denne likheten mindre relevans, og det som virkelig skiller dokumentene, mer. Dette leder oss også inn på en annen karakteristikk av fritekstsøk som ofte er et problem, nemlig at man potensielt kan få veldig mange treff på et søk, men at man sjelden er interessert i så mange. Dette utgjør en stor forskjell fra vanlige databasesystemer der resultatene fra et søk er likverdige og det at man får med alle er meget viktig. I et IR-system er man som oftest kun ute etter de N beste resultatene. Til forskjell fra tradisjonelle søkesystemer er altså rangering og tilbakemelding om dokumentets relevans i forhold til spørringen en vesentlig egenskap ved et IR-system.

I et databasesøk trenger man dessuten god kjennskap til det man skal søke i. I en relasjonsdatabases tilfelle trenger man kunnskap om relasjonene og deres sammenheng for å kunne søke opp den informasjonen man trenger. I IR søker man som ofte kun i en stor mengde tekst, uten kjennskap til hvordan den er strukturert. Kravet til presist formulerte spørringer er også større i databaser, slik som i spørrespråket SQL. I mange IR-systemer kan spørringer formuleres mye mer generelt siden man vet lite om det man søker i.

[1] Det er også vanlig å kalle en mengde dokumenter man utfører tekstanalyse på for et *korpus*.

4.2. Modeller for sammenligning

Fritekstsøk handler altså om å finne dokumenter eller informasjon som omhandler det man søker etter. Vi skal med andre ord skille relevante dokumenter fra de som ikke er relevante ved å sammenligne dem med en spørring. Det finnes en lang rekke metoder og modeller for å gjøre dette, og jeg skal gi en kort introduksjon til disse sammenligningsmodellene.

Boolsk sammenligning

Ved boolsk sammenligning skilles dokumentene ved hjelp av en logisk funksjon. Spørringen er en funksjon som spesifiserer hvilke nøkkelord som skal være med i dokumentet og hvilke som ikke skal. I et slikt system vil ikke spørringen være et dokument, fordi dokumentet og spørringen har helt forskjellig konstruksjon. Et rent boolskbasert system tar ikke hensyn til grader av relevans, og gir dermed ikke mulighet for noen form for rangering mellom dokumenter. Enten oppfyller dokumentet kravene til spørringen eller ikke. Systemer som baserer seg på boolsk sammenligning bruker derfor andre teknikker for å gjøre dette. Eksempler kan være nærhet mellom forskjellige nøkkelord eller at dokumentet inneholder flere av nøkkelordene det spørres etter. Ta for eksempel en spørring 'A OR B OR C'. Her vil man kunne rangere dokumenter som har alle termene, både A, B og C, høyere enn de som bare inneholder ett eller to.

Vektorbasert sammenligning

I en vektorbasert modell er likheten mellom dokumenter enten definert ved at like dokumenter har vektorer som ligger nær hverandre i et vektorrom eller ved at vinkelen mellom dem peker i samme retning. Begge teknikkene baserer seg på at dokumenter kan representeres med en dokumentvektor på eksempelvis denne formen:

$$D = <t_1, t_2, ..., t_n>$$

Her er komponentene i vektoren, alle ord som finnes i korpus, ordnet i en bestemt rekkefølge. Hvis ordet ikke forekommer, settes *t* til 0, og i de andre tilfellene kan den enten settes til 1 eller til antall forekomster av det bestemte ordet. Hvert ord tilsvarer en dimensjon i vektorrommet og med et normalt vokabularforbruk kan dette rommet få over 10^5 dimensjoner. Dokumentvektorene regnes ofte ut under indeksering av dokumentene.

Vektorrommodellen har en fordel over den boolske modellen fordi den tar høyde for grader av relevans. Allikevel deler også denne teknikken dokumentmengden i to, de dokumentene som ligger innenfor terskelverdien og de som ikke gjør det.

Sannsynlighetsbasert sammenligning

I denne modellen tar man i enda større grad enn den foregående hensyn til den usikkerheten og vagheten vi har i IR. Vektorrommodellen kan ikke oppfylle dette fordi den opererer med en terskelverdi. I denne modellen regner man ut usikkerheten direkte i form av at man beregner *sannsynligheten* for at et dokument er likt et annet, eller er lik en spørring. Detaljene rundt denne modellen er for omfattende til å forklare i sin fulle dybde her, men enkelt fortalt dreier det seg om følgende: Hvis man plukker ut et tilfeldig dokument, er det en bestemt

sannsynlighet, basert på antall forekomster av ordet totalt sett, for at dokumentet inneholder dette ordet. Videre er det en bestemt sannsynlighet, basert på forekomster av dette, for at et bestemt ord følger ett annet, slik som for eksempel ”political” og ”science”. Basert på nøkkelordene i spørringen, regnes det ut en sannsynlighet for hvert dokument for hvorvidt den er lik eller ikke lik spørringen.

”Data Fusion”

Fordi ingen av de metodene som er nevnt over er perfekte i alle situasjoner, kombineres de gjerne. Dette blir kalt ”Data Fusion”. De fleste IR-systemer gjør dette, og resten av dette kapittelet handler i stor grad om dette og hvordan forskjellige teknikker brukes for å øke relevansen til de dokumentene som hentes.

4.3. Indeksering

Fritekstsøk handler altså om å finne dokumenter eller informasjon som omhandler det man søker etter. I dag bruker man først og fremst nøkkelord for å beskrive hva et dokument handler om, og det er den viktigste egenskapen som knytter et dokument til en spørring. Brukeren av et IR-system oppgir gjerne et eller flere nøkkelord som man assosierer med den informasjonen man er på jakt etter. Nøkkelord er derfor en karakteristikk av hva et dokument handler om, og indeksering er en prosess for å tildele dokumenter nøkkelord.

En indeks kan sees på som en matematisk relasjon mellom et dokument og et sett nøkkelord[30]:

$$index : doc_1 \xrightarrow{about} \{kw_j\}$$

Den inverse relasjonen blir da slik:

$$index^{-1} : \{kw_j\} \xrightarrow{describes} doc_1$$

Herav kommer begrepene inverterte filer eller inverterte indekser, som er en av de mest brukte indeksstrukturene i dagens IR-systemer. Disse er nærmere beskrevet under avsnitt 4.7.1.

Indeksering, definert som den prosessen der en velger ut ord som skal representere dokumentet, kan enten gjøres manuelt eller automatisk. I mange tilfeller gjøres dette manuelt, bakerst i fagbøker for eksempel, finner man som oftest en indeks med nøkkelord og hvilke sider i boken som omtaler disse begrepene. Disse er plukket ut manuelt av forfatteren. Et annet eksempel er artikkeldatabaser, der forfatterne av en artikkel plukker ut noen nøkkelord som beskriver innholdet i artikkelen. Manuell indeksering har en åpenbar fordel ved at dokumentene blir tildelt nøkkelord som stemmer med hva dokumentet handler om. Ved hjelp av et menneskes assosiative evner har man mulighet til å tildele dokumenter nøkkelord som ikke en gang er nevnt i den teksten man skal indeksere, men som allikevel beskriver den. Manuell indeksering fører imidlertid til at nøkkelordene blir subjektivt tildelt. Sett at vi lar to forskjellige personer indeksere det samme dokumentet. Vi vil neppe få de samme nøkkelordene fra begge. I et IR-system med store mengder dokumenter, noe som ofte vil være tilfelle i slike systemer, vil det dessuten bli en alt for tidkrevende jobb å indeksere alle dokumentene manuelt. Derfor er de fleste IR-systemer i dag basert på automatisk indeksering.

Når en indekserer, kan man enten bruke et lukket sett med nøkkelord eller man kan bruke alle mulige ord som nøkkelord. I et system med automatisk indeksering, indekserer vi gjerne med et åpent eller ukontrollert vokabular.

Utfordringen er da å plukke ut de ordene som beskriver hva dokumentet handler om. Hvilke nøkkelord som er relevante for et dokument, baserer seg i dag stort sett på følgende antagelse[4]:

> *Hvis et ord forekommer flere ganger enn det man ville forventet i en samling dokumenter, beskriver dette ordet hva dokumentet handler om.*

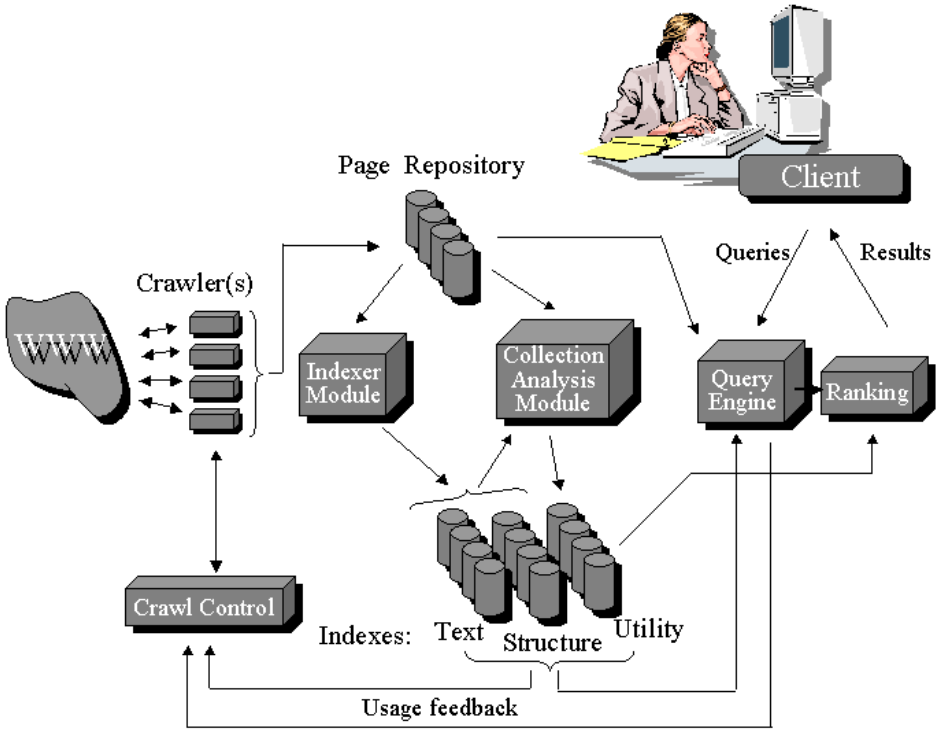
Dette gjør at vi indekserer stort sett alle ord i dokumentet, for senere å se på antall forekomster for å bedømme om ordet har noe med dokumentets innhold å gjøre. Det er likevel vanlig å fjerne en del ord samt endelser før man indekserer dokumentet. Prosessen med å dele opp dokumentet i ord og prosessere ordene slik at de blir til de ordene vi ønsker å indeksere, kalles normalisering og er nærmere beskrevet senere. Jeg vil også diskutere andre teknikker i tillegg til antall ordforekomster for å finne relevant dokumenter.

4.4. Oversikt over et IR-/søkemotorsystem

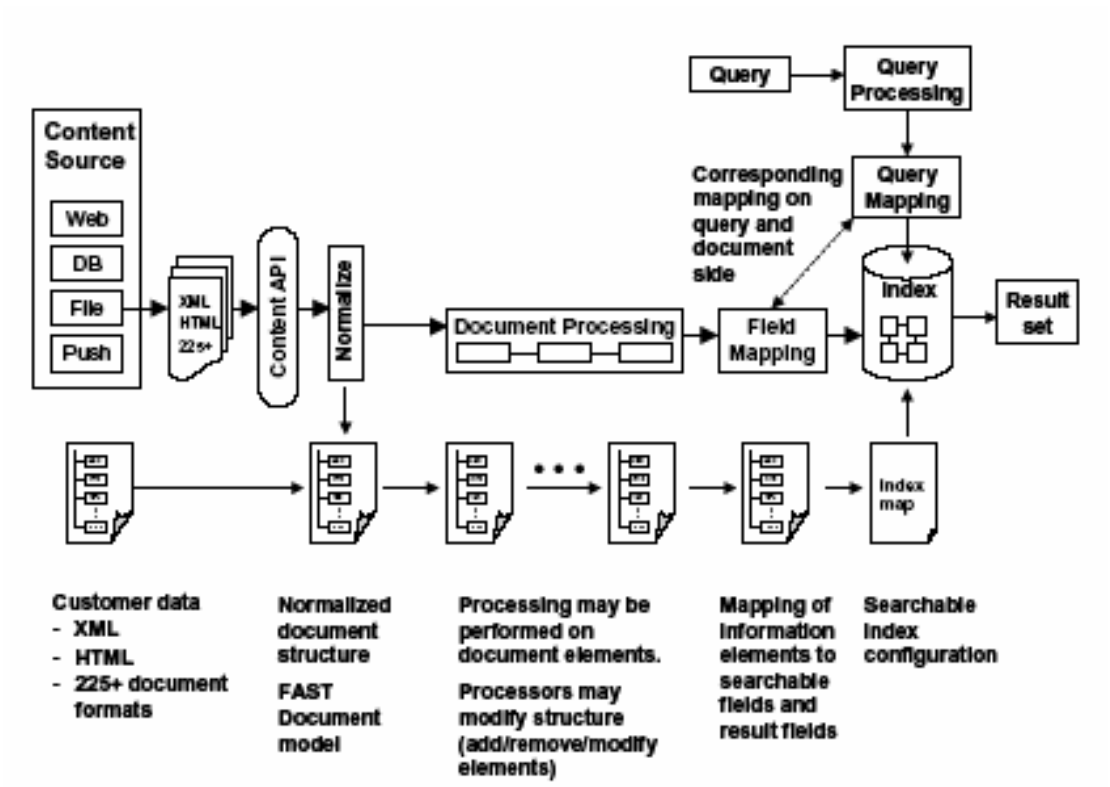
Et IR-system har vanligvis en arkitektur som bygger på følgende prinsipper [31, 32]:

Dokumenter, hentet for eksempel ved hjelp av en crawler[1], blir sendt inn i en prosesseringspipeline som består av flere steg der dokumentet bli normalisert. Dette resulterer i en indeks som brukes av søkemotoren for å hente ut relevante dokumenter. Brukeren sender en spørring til søkemotoren som henter ut de relevante dokumentene og rangerer dem i forhold til brukerens spørring og presenterer dem for brukeren. Ofte vil IR-systemet også ta vare på en kopi av dokumentene som indekseres slik at de kan hentes ut av systemet i sin helhet.

[1] En crawler er en prosess/dataprogram som leter etter informasjon som den sender videre til et IR-system. Et eksempel på dette kan være en fil-crawler som leter gjennom filområder eller en webcrawler, der nye websider blir funnet ved å følge hyperlinker.



Figur 4-1: Denne figuren, som er hentet fra [31], viser en de vanligste komponentene i et IR-system med fokus på websøkemotorer.



Figur 4-2: Denne figuren viser hvordan dokumenter i Fast Data Search (FDS) blir prosessert. Figuren er hentet fra [32].

4.5. Spørrespråk

De fleste IR-systemer er basert på boolsk logikk for å kombinere søkeord. De tre mest vanlige boolske operatorene er AND, OR og NOT. Operatorene utfører henholdsvis mendgeoperasjonene snitt, union og differanse på det settet av dokumenter som blir hentet. Noen systemer tilbyr også XOR-operatoren, men denne kan også uttrykkes ved hjelp av de tre andre operatorene, og er mindre nødvendig, siden de fleste brukere ikke har så god kjennskap til boolsk logikk. Ved å plassere de boolske uttrykkene i parenteser kan man styre presedensen på operatorene, ellers vil presedensen vanligvis være NOT, AND og OR. En av ulempene med boolske spørrespråk er at mange brukere ikke kan boolsk logikk, og derfor ikke klarer å gruppere utrykkene riktig. Resultatet kan dermed bli noe helt annet enn det brukeren egentlig mente å uttrykke. De fleste brukere vil imidlertid ikke lage spesielt komplekse spørringer, derfor er boolsk språk veldig enkelt og intuitivt å forstå og kan brukes nesten uten forkunnskaper.

En annen variant av det boolske spørrespråket er vektede boolske spørringer. Et stadig problem med vanlige boolske spørringer er at de ofte gir veldig mange treff, særlig ved bruk av OR-operatoren, uten noen måte å angi hvordan man vil rangere nøkkelordene man søker etter. Med en vektet boolsk spørring kan man angi at det ene nøkkelordet skal ha høyere relevans enn det andre, slik som for eksempel:

$$\texttt{Query: } \texttt{cat}_{0.2} \texttt{ OR } \texttt{dog}_{0.8}$$

der ’dog’ er vektet 80% og ’cat’ 20%.

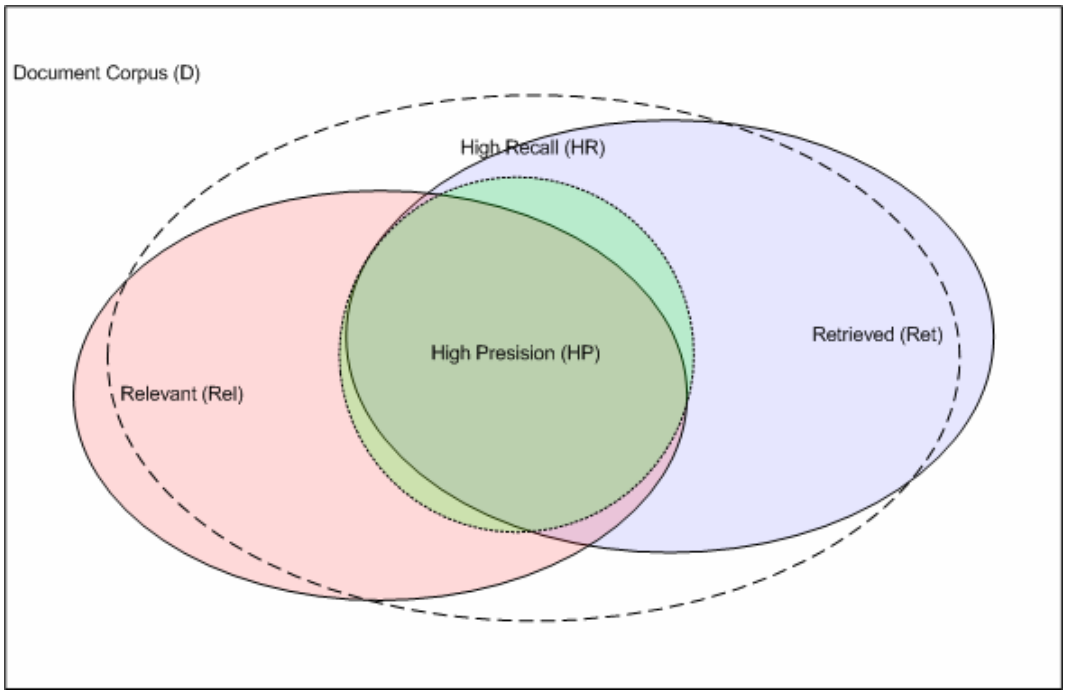
Fra brukerens ståsted vil det ofte være enklere og mer komfortabelt om man kunne skrive spørringene i et naturlig språk. Slike spørringer er lette å formulere, men ofte upresise og tvetydige. Det er dessuten svært vanskelig å automatisk tyde semantikk, oppbygning og meningen i slike spørringer, som ikke følger en forhåndsdefinert syntaktisk oppbygning. En enkel måte å prosessere naturlige spørsmål på er f.eks. ved å fjerne de ordene som ikke er informasjonsbærende i spørringen, såkalte *stoppord*. Det vil imidlertid også fjerne selve styrken i den naturlige spørringen, slik som syntaktisk og pragmatisk forståelse. Det er dessuten vanskelig å forstå negasjon, slik som ’Gi meg alle de dokumentene som handler om hunder, men ikke de som handler om pudler’. Interessen for å kunne prosessere naturlige spørringer er stor, men mulighetene i dagens IR-systemer for å forstå slike spørringer er svært begrensede.

I noen spørrespråk kan man gi mer nøyaktig beskrivelse av egenskaper man søker etter i et dokument. Eksempler på dette kan være søk etter alle dokumenter der to bestemte ord finnes i samme setning eller avsnitt, eller det kan være dokumenter som enten inneholder et ord som er synonymt med et annet eller som er på et bestemt språk. Disse tingene vil jeg beskrive nærmere ut over i kapittelet.

4.6. Funksjonalitet

I et IR-system benyttes gjerne flere forskjellige teknikker for å assosiere (”matche”) en spørring med et sett dokumenter (se ”Data Fusion” 4.2). Det vi ønsker å oppnå med dette er å hente ut de *relevante* dokumentene. Relevans i IR betyr at dokumentet handler om det samme som spørringen, og det på en slik måte at et menneske vil være enig. Relevans er heller ikke noe absolutt. Dokumenter kan være mer eller mindre relevante i forhold til en spørring. Relevans er med andre ord et temmelig uklart begrep.

For å gjøre dette litt mer konkret, la oss si vi har en mengde dokumenter i et IR-system; D. Ut i fra en bestemt spørring har vi en delmengde, *Rel*, som utgjør de relevante dokumentene som brukeren ønsker å få tak i. Et ideelt IR-system vil hente ut alle, men ikke flere dokumenter enn det er i mengden *Rel*. Mengden *Ret* består av de dokumentene som faktisk blir funnet relevante av systemet. Venn-diagrammet på Figur 4-3 viser mengden relevante dokumenter og mengden som faktisk hentes.



Figur 4-3: Et Venn-diagram som viser en mengde dokumenter D (korpus), og forholdet mellom de relevante dokumentene (Rel) og de gjenfunnede dokumentene (Ret). Diagrammet illustrerer videre begrepene høy presisjon (HP) som omfatter dokumentene innefor stiplede sirkelen i midten, og høy gjenfinningsgrad (HR) som er dokumentene innefor den store stiplede sirkelen. Gjengitt fra [30].

For at et IR-system skal fungere bra må snittet mellom *Ret* og *Rel* være størst mulig, men det er to måter å oppnå dette på. Hvis vi er opptatt av at systemet skal returnere absolutt alle relevante dokumenter, snakker vi om at systemet skal ha høy gjenfinningsgrad (recall). Det er definert slik:

$$\text{Recall} = \frac{|\,\text{Ret} \cap \text{Rel}\,|}{\text{Rel}}$$

Et system med høy gjenfinningsgrad (HR) vil returnere bortimot alle relevante dokumenter, samt en del ikke-relevante dokumenter. Er vi interessert i at flest mulig av de dokumentene vi henter skal være relevante, må vi ha et system med høy presisjon. Det er definert slik:

$$\text{Precision} = \frac{|\,\text{Ret} \cap \text{Rel}\,|}{\text{Ret}}$$

Et system som har høy presisjon (HP), vil returnere en mengde dokumenter der omtrent alle er relevante, men utelater gjerne noen.

Antall dokumenter i et IR-system er som regel veldig stort, og hvordan man kan øke systemets gjenfinningsgrad og presesjon er ikke entydig. Noen IR-systemer bruker teknikker som andre igjen mener er med på å minske disse faktorene. Det vil også ofte være slik at høyere gjenfinning vil gi lavere presisjon og omvendt, men dette er heller ikke entydig. Disse to målene er imidlertid ikke enkle å måle nøyaktig. Antall relevante dokumenter i forhold til en spørring er vanskelig å telle og hvilke som er relevante og hvilke som ikke er det er dessuten svært subjektivt.

Det er ikke definert noen standard for hva slags teknikker man kan bruke. Jeg vil allikevel i de neste avsnittene gi en oversikt over de mest vanlige metodene som brukes i dagens forskjellige IR-systemer.

4.6.1. Tekstanalyse og lingvistikk

Stemming og lemmatisering

I et ukontrollerbart vokabular vil ord opptre i mange forskjellige former, slik som flertalls-, entalls- og tidsbøyninger. Ta for eksempel de engelske ordene ’computer’, ’computers’, ’computing’, ’compute’, ’computed’, ’computable’. Alle disse ordene stammer egentlig fra det samme basisordet ’compute’, og er i mange sammenhenger sterkt relatert til hverandre. Et søk på ordet ’computers’ vil ikke treffe alle de relevante dokumentene hvis det ikke omfatter alle former av ordet. For å oppnå høyere gjenfinning ønsker man derfor at dokumenter med alle disse formene av ordet ’comput’ blir sett på som relevante hvis man søker på en av formene.

Under tekstprosesseringen ønsker man følgelig å finne frem til rotordet og indeksere dette, slik at søk i indeksen på ordets basisform vil gi treff i alle dokumenter som inneholder en form av dette ordet. Denne prosessen kalles ofte å *normalisere* teksten, men ofte brukes dette begrepet i en videre betydning til å omfatte all type analyse og behandling av ordene som skal indekseres. Begrepet *stemming* brukes derfor spesifikt om denne prosessen med å redusere ord til dets basisform. Stemming består i stor grad i å fjerne endelser og prefikser. Det mest vanlige er å fjerne kun endelsene, siden det er vanskeligere å bedømme om en bokstavsekvens virkelig er et prefiks eller ikke. Et åpenbart problem som en stemmer må kunne ta høyde for, er ord som skifter form, slik som være/var/er. Stemming kan være en noe tidkrevende prosess. I de mest omstendelige teknikkene blir ordene sendt gjennom en stemmingalgoritme flere ganger, helt til ordet har nådd sin basisform. På hvilket tidspunkt man ønsker å gjøre normaliseringen, varierer også ettersom hvor i systemet man ønsker den beste ytelsen. Det vanligste er å gjøre det under dokumentprosesseringen, før ordene går til indeksering, slik at man kun indekserer og lagrer basisformen av ordene. Tilsvarende må man da gjøre med spørringen før oppslag i indeksen. Dette har den fordel at det er plassbesparende i forhold til å indeksere alle ordene i sin opprinnelige form, spesielt hvis man bruker inverterte indekser.

Et annet alternativ er først å redusere ordet til dets basisform for så å utvide ordet til alle mulige former. Dette kalles *lemmatisering,* og er den omvendte prosessen av stemming. Etter at alle ordene er lemmatisert, kan man indeksere alle formene slik at de peker på dokumentet der en av formene forekommer. Den åpenbare negative siden ved å gjøre det slik er at størrelsen på indeksen blir meget stor. En fordel er derimot at man slipper å lemmatisere eller normalisere spørringen. Hvis man slipper dette, trenger man ikke vite

hvilket språk spørringen er skrevet i, noe som ofte er vanskelig, om ikke umulig å oppdage automatisk, på grunn av spørringenes korte lengde. Språket trenger man for å kunne lemmatisere riktig. I mange spørrespråk er det mulig å definere dette, men i spørrespråk som skal være enkle å bruke, må man da lemmatisere for alle støttede språk. En tredje løsning er å indeksere ordene i sin opprinnelige form og lemmatisere spørringen, noe som vil gjøre at spørretiden går opp og indekseringstiden ned.

Stoppord

Indeksering med et ukontrollert vokabular fører til flere problemer selv om man tar i bruk normaliseringsteknikker som beskrevet over. Et at dem knytter seg til det faktum at ikke alle ord i et dokument er med på å beskrive dokumentet like godt. Mange ord er viktige for en setnings syntaktiske oppbygning, men beskriver ikke dets mening på noen måte. Mange av disse ordene finnes også i nær sagt alle dokumenter. Flere studier av de mest vanlige ordene på engelsk viser at så mye som 50% av ordene i en gitt tekst er fra de mest vanlige 250 til 300 ordene. En annen viser at de to mest bruke ordene 'the' og 'of' står for 10% av teksten, og at 18 forskjellige ord står for 30% av teksten. På grunn av dette er det vanlig at IR-systemer har en "a priori" definert liste, ofte med 250-300 ord, som blir utelukket fra videre prosessering når teksten indekseres. Denne listen kalles en stoppordliste eller en negativ ordliste.

En kan imidlertid ikke bruke denne teknikken helt ukritisk. Ved fjerning av stoppord fra en frase vil man ofte miste helt vesentlige deler av meningen. Ta for eksempel den engelske frasen 'to be or not to be' som kun består av stoppord. Denne vil ikke være søkbar i et system der stoppord blir fjernet. Noen søkemotorer, deriblant FDS, velger derfor å ikke fjerne disse ordene, men reduserer isteden vekten av disse forekomstene til fordel for ord som ikke er definert som stoppord. Stoppord har på den måten liten innvirkning på rekkefølgen av resulatene, men har innvirkning på hvilke resultater som blir med. FDS er en av de få søkemotorene som klarer å inkludere stoppord i søk uten å få store ytelsestap.

Språk og språkdeteksjon

Hvilket språk teksten er skrevet på, er en viktig egenskap ved et dokument. I et IR-system vil man i flere av prosesseringsstegene være avhengig av å vite hvilket dette, for eksempel for å velge den riktige stoppordlisten og normalisere ord riktig. Ikke-europeiske språk introduserer en annen problemstillinger for IR-systemer, som i all hovedsak er utviklet med fokus på det engelske språk. Det faktum at for eksempel japansk tekst ofte ikke skiller ord med blanke tegn, blir derfor et problem. Mange IR-systemer støtter flere språk, og de fleste har derfor også automatisk deteksjon av hvilket språk teksten som prosesseres er skrevet på.

4.6.2. Ordrelasjoner og kryssreferansen

Ord kan relateres til hverandre på mange måter. For å kunne øke gjenfinningen i et IR-system benyttes det ofte teknikker som utvider ord med andre relaterte ord, slik at flere dokumenter kommer med i resultatet. Den mest brukte relasjonen er likhetsrelasjonen, altså synonymer. Enten automatisk eller spesifisert i spørringen, kan et nøkkelord utvides med dets synonymer. Søk på f.eks. 'PC' vil utvides slik at systemet søker både på 'PC' og 'datamaskin'. Dette vil føre til økt gjenfinning.

En annen type relasjon er hierarkiske relasjoner mellom ord. Hvis vi ser på ordet 'datamaskin', vil 'bærbar PC' og 'lomme-PC' være to mer spesifikke betegnelser. Omvendt vil ordet 'datamaskin' være en mer generell betegnelse på 'bærbar PC'. 'Lomme-PC' vil også være relatert til 'bærbar PC'. De vanligste relasjonene er:

- Videre form av ord/uttrykk (BT[1]), det vil si et ord med mer generell mening.
- Videste form av ord/uttrykk (TT[2]), det vil si det ordet med den mest generelle meningen.
- Smalere form av ord/uttrykk (NT[3]), det vil si et ord med en mer spesifikk mening.
- Foretrukket form av ord/uttrykk (PT[4]), et foretrukket synonym.
- Relatert ord/uttrykk (RT[5]), ord som relaterer seg til et annet ord, slik som f.eks. synonymer.

Hvis man søker på et mer generelt ord, vil man øke gjenfinningen og minske presisjonen i søket, mens hvis man søker på et mer spesifikt ord, vil man kunne øke presisjonen, men minke gjenfinningen.

Homonymer, det vil si ord som skrives likt, men som har en helt ulik betydning, kan imidlertid føre til at et ord kan bli ekspandert feil og dermed minker presisjonen av et søk betydelig. Et synonym til et homonym vil kunne ha en helt annen betydning enn det brukeren mente. Man kan enten la brukeren interagere med systemet og avgjøre hvilken betydning som er den korrekte, eller man kan analysere de andre nøkkelordene i søket for å finne den rette betydningen. Homonymer utgjør uansett et problem for IR-systemer, noe som ofte fører til lav presisjon.

Ord kan relatere seg til hverandre også på helt andre måter. Fuzzy-søking er en søkeform der man kan søke etter ord som har lignende stavemåte. Dette er spesielt effektivt når en søker i tekst der det kan forekomme mye stavefeil, slik som i OCR[6]-prosessert tekst, men også der brukeren er i tvil om stavemåten.

Noen IR-systemer tillater også søke etter ord som uttales ellers høres ut som andre ord, såkalte 'soundex'- eller 'soundslike'-spørringer. Dette er også en form for relasjon mellom ord. En siste variant som tilbys av noen systemer, er 'is about'-søk, som skal avgjøre om et dokument handler om noe. Man kan spørre seg om det ikke er det man hele tiden ønsker å oppnå, men denne måten å søke på stiller strengere relevanskrav til dokumentet for å bli inkludert i resultatet enn om man hadde søkt direkte på nøkkelordet. Det er altså en måte å instruere IR-systemet i at man ønsker en høyere grad av presisjon.

4.6.3. Tekstformatering, syntaktisk oppbygning

En *frase* er to eller flere ord satt sammen til en semantisk enhet. Den spesifiserer både ordene og hvilken orden og nærhet de må ha i forhold til hverandre. Et søk på en slik frase vil kunne returene dokumenter der frasen er nøyaktig slik den er oppgitt i

[1] Broader term
[2] Top term
[3] Narrow term
[4] Preferred term
[5] Related term
[6] Optical Character Recognition

spørringen. Bruk av fraser øker dermed presisjonen, men reduserer samtidig gjenfinningen i spørringer der man bruker dette i stedet for å søke på ordene separat.

Søk på frase kan sees på som et spesialtilfelle av *nærhet-/kontekstsøk*. Dokumenter er strukturelt oppbygd med avsnitt, setninger, overskrifter, ord og bokstaver, og denne syntaktiske oppbygningen er ofte interessant, fordi den er med på å formidle innholdet av dokumentet. IR-systemer kan gjøre nytte av denne oppbygningen for å øke presisjonen i spørringene på flere måter. Ved søk på for eksempel to nøkkelord kan dokumenter der de står nærmere hverandre få en høyere rangering enn andre dokumenter (se under). Det vil også være nyttig å kunne spesifisere denne avstanden i en spørring slik som:

$$\texttt{Query: } kw_1 \texttt{ within X UNITS OF } kw_2$$

der X er et tall som definerer avstanden og UNITS er en enhet slik som ord, avsnitt osv. Et spesialtilfellet av dette er der X f.eks. er satt til null, det vil si at ordene må være innenfor samme avsnitt eller setning, eller at ordene må stå i en bestemt orden.

En annen funksjonalitet som er inkludert i de fleste IR-systemer, er muligheten for å søke etter spesielle mønstre ved å bruke jokertegn. Effekten av dette er stor i systemer der det ikke gjøres lemmatisering/stemming slik at man kan legge til vilkårlige prefikser og suffikser selv. Det finnes gjerne to typer jokertegn, en som erstatter et vilkårlig tegn og en annen som erstatter flere vilkårlige tegn. I mange systemer brukes slike prefiks-jokertegn automatisk uten at brukeren trenger å spesifisere det i spørringene.

4.6.4. Analyse av spørringer

Analyse og omskrivning av spørringen etter at brukeren har formulert den ferdig og sendt den til IR-systemet, kan i mange tilfeller føre til bedre presisjon. Det finnes flere teknikker som særlig brukes av web-søkemotorer der spørrespråkene er enkle. I IR-systemer der man har mulighet for å skrive mer presise spørringer, er ikke disse teknikkene like effektive.

Stavekontroll på ord og fraser i spørringen kan være en god støtte for brukeren, slik at resultatet av en spørring blir som forventet. Dette kan enten gjøres ved at feilstavede ord endres automatisk i spørringen eller ved at systemet gjør oppmerksom på stavefeil og eventuelt foreslår andre stavemåter. Mest interessant er det kanskje når det gjelder egennavn, der stavemåten ikke er så åpenbar. En forhåndsdefinert liste med egennavn som systemet kan sjekke mot, kan være til hjelp for å rette slike feil. En slik liste kan også brukes til å gjenkjenne egennavn. Da kan man utelate disse fra videre prosessering, slik som for eksempel stemming eller lemmatisering, samt at egennavnet kan behandles som en frase, selv om dette ikke var oppgitt i spørringen.

Antifrasing er en teknikk for å fjerne eller redusere vekten av fraser som ikke er informasjonsbærende, det vil si typiske stoppord. Hvis man for eksempel formulerer et spørsmål i naturlig språk, vil ord som ikke er relevante for spørringen bli fjernet, slik at ”Hvem er X” blir redusert til ”X”.

Segmentering[33] av spørringer er en annen teknikk som baserer seg enten på kunnskap fra de indekserte dokumentene, eller på tidligere spørringer, for å gruppere ord i fraser slik at presisjonen i spørringen blir bedre. Spørringen ”large search engine systems” vil for eksempel kunne bli segmenter til ”large ’search engine’ systems”.

Ofte kan flere etterfølgende ord sees i sammenheng og det vil føre til bedre presisjon hvis de behandles som en enhet. Egennavngjenkjenning for eksempel, kan sees på som en del av en mer generell prosess kalt terminologiekstraksjon. Hensikten er å gjenkjenne viktige biter av teksten. Dette kan for eksempel være gjenkjenning av forkortelse og substantivfraser slik som ”political science”.

4.6.5. Kategorisering

Det finnes en rekke teknikker som bygger på klassifisering og kategorisering av dokumenter, og som kan brukes til å øke presisjon men også gjenfinning i IR-systemer.

Det finnes generelt to måter å kategorisere dokumenter på: Forhåndsdefinert (overvåket) eller ikke-forhåndsdefinert (uovervåket) kategorier. Ved en overvåket kategorisering er kategoriene allerede definert, og man kan trene systemet til automatisk å plassere dokumentene inn i riktig kategori. Dette gjøres ved å plassere dokumenter med manuelt definerte kategorier inn i systemet. Ved en uovervåket kategorisering lager systemet selv kategorier basert på likheter mellom dokumenter. Likhet og avstand mellom dokumenter må så defineres. Den vanligste modellen for dette er å bruke vektorrom, men også andre sammenligningsmodeller kan brukes (se avsnitt 4.2). Kategorier er da definert ved at en dokumentvektor befinner seg innenfor en viss terskelverdi, slik som innenfor en bestemt vinkel, peker i en bestemt retning eller er i en gitt avstand fra kategoriens vektor.

I et IR-system kan disse dokumentkategoriene brukes til flere forskjellige formål. Hvis vi tenker oss et system med faste kategorier (overvåket), kan man la brukerne grave seg ned i kategorier (drill-down), nærmest et slags alternativ til å søke ved hjelp av nøkkelord. Eksempler på slike systemer er Yahoo![1] og Gule Sider[2]. Man kan også gi brukeren mulighet til å begrense søket innefor en bestemt kategori, slik at søket kun blir utført på en delmengde av dokumentene i systemet.

Når en bruker har funnet et dokument som er relevant, vil det ofte være ønskelig å kunne hente opp lignende dokumenter, såkalte ”find similar”-søk. Et annet bruksområde er gruppering av lignende dokumenter som har blitt hentet ut ved et søk, slik at brukeren ikke skal måtte behøve å bla gjennom mange dokumenter med tilnærmet likt innhold hvis han ikke finner dem relevante.

4.6.6. Relevans og rangering

Som tidligere nevnt, er det viktig at et IR-system kan rangere dokumenter etter hvor relevante de er i forhold til spørringen, og presentere dem i en rangert orden der de mest relevante dokumentene blir presentert først. Jeg har hittil diskutert forskjellige teknikker for å kunne øke både presisjon og gjenfinning i et IR-system, det vil si selve utvelgelsen av dokumentene. Hvordan de skal rangeres, det vil si hvordan vi skal finne dokumenter som er relevante i forhold til en spørring, gjenstår.

Et rent boolsk basert sammenligningssystem avgjør som tidligere nevnt kun om dokumentet er relevant i forhold til spørringen eller ikke. Det finnes imidlertid mange teknikker for å gjøre dette bedre. Det mest åpenbare er å ta hensyn til antall ordforekomster i dokumentet, slik at dokumenter med et høyt antall forekomster av et søkeord blir rangert høyere enn de som har få forekomster. Dette kalles *”absolute term*

[1] www.yahoo.com
[2] www.gulesider.no

frequency". Hvis man kun ser på det absolutte antallet, diskriminerer man lett korte dokumenter som kan være vel så relevante som de lange. Det er derfor vanlig å bruke et relativt antall ordforekomster, *"relative term frequency"*, der man tar høyde for dokumentets lengde, for eksempel ved å dele på antall ord totalt i dokumentet. Det er også vanlig å ta hensyn til antall forekomster globalt sett, det vil si antall forekomster i alle dokumenter som finnes i IR-systemet, og normalisere dette i forhold til antall ord globalt. Det globale forekomstantallet brukes på forskjellige måter til å justere relevansen. Hvis et nøkkelord forekommer i en stor del av dokumentene i systemet, er også ordet mindre egnet til å skille dokumenter fra hverandre. Høy frekvens av et slikt ord i et dokument blir derfor tillagt mindre vekt enn et ord som forekommer sjelden globalt sett. Dette kalles *"inverse document frequency*[1] *weight",* fordi ord som forekommer ofte tillegges mindre vekt enn de som forekommer sjelden i dokumentsamlingen.

Et boolsk basert søk på *A OR B OR C* vil i utgangspunktet ta med alle dokumenter som enten inneholder A, B eller C, uten noen videre rangering mellom disse. Det synes imidlertid ganske selvfølgelig at dokumenter som inneholder mer enn ett av dem, er mer relevante en de andre. En annen teknikk som ofte brukes, er nærhet mellom nøkkelordene ("term proximity"), der dokumenter som inneholder ordene A og B som står tett sammen, f.eks. i samme setning, blir rangert høyere enn dokumenter der A og B står lenger fra hverandre.

Det finnes også helt andre teknikker, slik som for eksempel de som tar hensyn til bruksområdet for IR-systemet. Et eksempel på dette er PageRank-algoritmen til Google[31]. Her sees mengden av alle dokumenter, altså websidene i systemet, på som en graf, der alle dokumentene er forbundet med hverandre gjennom hyperlenker. En lenke fra A til B kan sees på som en anbefaling av dokument B fra A. Enkelt fortalt rangeres dokumenter etter hvor mange innkommende lenker de har, men det tas også hensyn til hvor viktig siden som lenkene peker fra er, det vil si hvor mange innkommende lenker denne siden har. På denne måten blir en side rangert høyere hvis det er en viktig side som peker på den, til forskjell fra hvis en helt ubetydelig side peker på den. En annen teknikk som brukes i søkemotorer, er indeksering av lenketeksten. I mange tilfeller kan teksten som står på den lenken som peker til et bestemt dokument, si mer om innholdet enn selve dokumentet.

Som jeg har nevnt tidligere, er bruk av vektorbasert sammenligning en annen teknikk for å avgjøre hvor relevante dokumentene er i forhold til en spørring. Et dokument som ligger nærmere spørringen i vektorrommet, kan sies å være mer relevant enn dokumenter som ligger lengre unna.

4.6.7. Dokumentsammendrag

En annen type tilbakemelding til brukeren om dokumentene som hentes opp, ved siden av relevansrangering, er et slags sammendrag eller "teaser", som brukeren raskt kan lese gjennom for å danne seg et bilde av dokumentet, og se om det er relevant for brukeren eller ikke. Slike sammendrag kan være statiske, det vil si at et dokument har et predefinert sammendrag, som enten er generert opp automatisk, eller lagt til manuelt og som presenteres hver gang. Dynamiske sammendrag er sammendrag som lages på bakgrunn av spørringen, normalt setninger rundt de nøkkelordene det ble søkt etter.

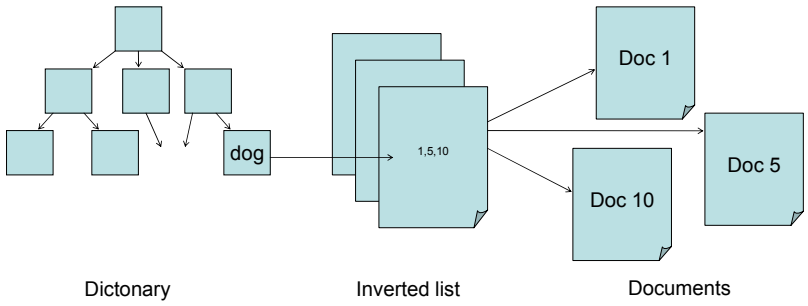
[1] IDF

4.7. Effektivitet.

En av de største utfordringene til et IR-system er effektivitet. IR-systemer brukes til å indeksere og søke i store mengde dokumenter. Typiske bruksområder er digitale biblioteker, nyhets- og artikkeldatabaser og websøkemotorer. I det siste tilfellet, som også har en meget hurtig vekst, indekserer søkemotorer som AllTheWeb og Google flere milliarder websider. Disse systemene bør kunne takle flere tusen forespørsler i sekundet og hvert søk skal helst besvares på under ett sekund. Løsningen for å få til dette er effektive datastrukturer og massiv parallellisering.

4.7.1. Datastrukturer

De fleste IR-systemer i dag er basert på inverterte filer eller inverterte indekser som datastruktur for søk. En invertert filstruktur består i hovedsak av tre komponenter: Dokumenter, en oppslagsfil og invertertingslister. Navnet *invertert* kommer av den underliggende metoden med å lagre alle nøkkelordene i en liste, og for hvert slikt nøkkelord en liste med dokumenter der ordet forekommer. Enkelt fortalt gis hvert dokument en unik identifikator. I oppslagsfilen ligger alle unike ord i sortert rekkefølge, for eksempel alfabetisk, eller den kan implementeres som et B-tre slik som vist på figuren under. For å få tak i inverteringslisten må man gå gjennom oppslagsfilen som inneholder en peker til den aktuelle inverteringsfilen. I denne er alle numrene på dokumenter som ordet forekommer i, samt tilleggsinformasjon, slik som for eksempel hvor i dokumentet ordet forekommer, for å støtte nærhet- og kontekstsøk. Når brukeren gjør et boolsk søk, vil de aktuelle inverteringslistene bli hentet frem og resultatet bygget ved å utføre mengdeberegninger på disse. Til slutt blir dokumentene hentet opp fra dokumentlageret og sendt tilbake til brukeren. Fordelen med de inverterte listene er at kun den informasjonen man trenger, blir lest opp fra sekundærlagre, noe som medfører få dyre diskoperasjoner. Den eneste letingen man behøver å gjøre, er i oppslagsfilen, men et B-tre gir ofte en optimal løsning med hensyn på å gjøre få IO-operasjoner.



Figur 4-4: For å finne dokumenter som inneholder ordet "dog" slår man først opp i treet (dictonary) der trenoden peker på en inverteringsliste. Denne inneholder dokument id 1,5 og 10 som er de dokumentene som inneholder ordet "dog".

Inverteringslister er som sagt den mest brukte søkbare datastrukturen for IR-systemer. En alternativ datastruktur er N-gram, som bryter ned ord i mindre fragmenter for søk. Denne teknikken har vist seg å være mer effektiv enn inverteringslister med fulle ord i enkelte tilfeller. Et annet alternativ er signaturfiler. Denne teknikken bygger på en antagelse om at det lønner seg å raskt kunne skille mellom relevante og ikke-relevante dokumenter, for så å bruke andre søketeknikker for å finne sluttresultatet.

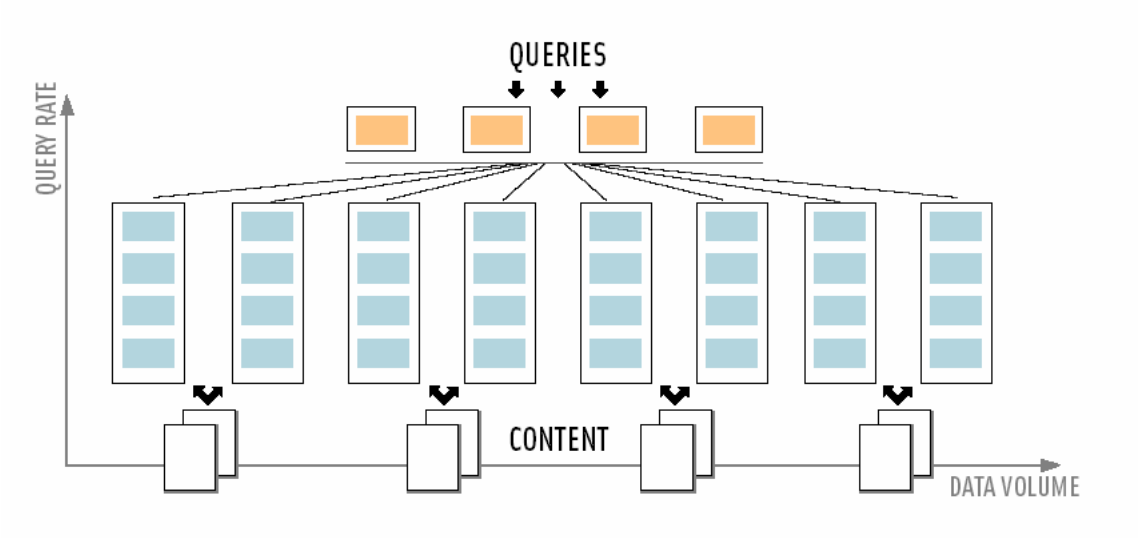
4.7.2. Parallellisering og skalerbarhet

Ved siden av å ha en effektiv datastruktur, er parallellisering og skalerbarhet utvilsomt de mest nødvendige egenskapene et IR-system kan ha for å lykkes. Søkemotorer, for eksempel Google, har tusenvis av maskiner for å crawle, indeksere og søke i web-dokumenter.

Som vist på Figur 4-1 og Figur 4-2, består et IR-system av flere distinkte prosesser som naturlig kan distribueres på forskjellige noder, slik at søkemotoren og dokumentprosesseringen foregår på forskjellige maskiner. Videre parallellisering og oppdeling av dokumentprosesseringen kan gjøres enten ved å la forskjellige noder utføre forskjellige steg i prosesseringspipelinen, slik at resultatet fra den ene noden går videre til den andre, eller ved å distribuere dokumentene som skal prosesseres over flere noder, og deretter sette sammen indeksen til slutt. Disse to teknikkene kan selvfølgelig også kombineres. De forskjellige stegene i en prosesseringspipeline er også resursskrevende på en måte som gjør dem egnet for parallellisering, også innenfor en node. Enkelte steg er svært CPU-intensive, mens andre deler av prosesseringen er IO-intensive.

Søkingen lar seg også dele opp på flere forskjellige måter. Som tidligere nevnt, er reduksjon av diskoperasjoner vesentlig for et IR-system, så vel som et databasesystem. Her er teknikker som bruker caching en klar utfordring. Videre er det vanlig, for ikke å si nødvendig, å distribuere indeksen over flere noder, fordi indeksen nødvendigvis blir stor. I eksempelet med en søkemotor vil indeksstørrelsen være flere terabyte, og man vil dessuten kunne gjøre diskoperasjoner parallelt. På grunn av dette er det derfor vanlig å duplisere indeksen flere ganger på forskjellige noder i systemer der søkebelastningen er stor.

Det er også vanlig å fragmentere indeksen over flere noder vertikalt, slik at f.eks. inverteringslistene fra ord som begynner med A og B er plassert på node X, C og D på node Y og så videre. En del av teknikkene jeg har nevnt i dette kapittelet, kan løses ved bruk av flere typer inverterte filer. Disse vil også kunne fragmenteres, slik at man får en type horisontal fragmentering. Søk i indeksfragmentene vil kunne utføres av en eller flere søkenoder. Selve søkeprosesseringen vil også kunne deles opp. Sortering og kombinering av de inverterte listene er problemer som enkelt lar seg parallellisere. Oppdeling, distribuering og kombinering av søkeresultater er nødvendig i en slik arkitektur, og man trenger derfor egne noder for å gjøre dette. Hvordan spørringen skal deles og distribueres, skjer både i henhold til hvilket fragment det søkes i og i henhold til lastbalansering over de dupliserte indeksene.



Figur 4-5: Denne figuren er hentet fra [32] og viser hvordan FDS parallelliserer spørringer. Langs X-aksen har vi mengden data som skal indekseres, og langs Y-aksen økning i antall spørringer. Hvis spørrvolumet er høyt, kan spørringer distribueres langs flere rader med søkenoder (boksene i midten av figuren) som hver prosesserer sitt subsett av datamengden. Man kan også legge til flere søkenoder som behandler det samme segmentet av data. Kontrollnodene (øverst) distribuerer lasten mellom nodene. Hver rad med spørrenoder kan bare behandle en viss andel av datamengden, så når denne øker, legges det til nye rader med noder.

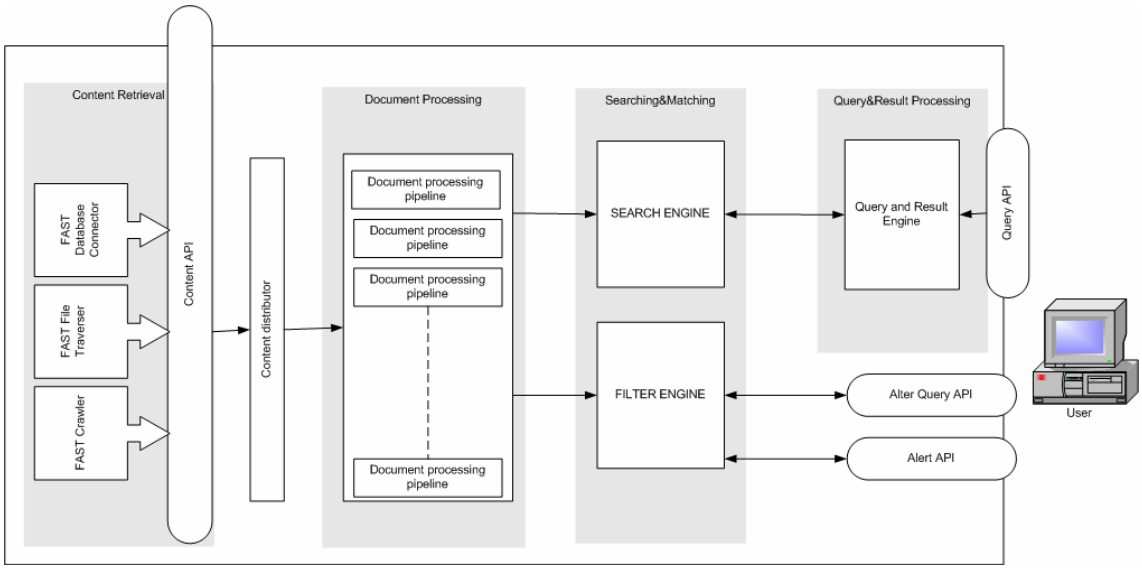
Kapittel 5. IR og databasesystemer

IR-systemer har i dag mange bruksområder og de blir tildels spesifikt tilpasset sitt applikasjonsområde. Databaser har et økende behov for å integrere IR-tjenester, og uviklingen i moderne kommersielle databasesystemer og databasestandarder er i ferd med å gjøre nettopp dette. I denne oppgaven fokuserer jeg på integrasjon mellom en relasjonsdatabase og en websøkemotor, som er en type IR-system som naturlig nok tilbyr kun den del av de mulighetene jeg har skissert i forrige kapittel. Jeg vil i dette avsnittet gå nærmere i dybden på FDS og funksjonaliteten som denne søkemotoren tilbyr, samt se på hva som eksisterer i databaser i dag, både kommersielle systemer som er i bruk, og standarden SQL. Spesielt vil jeg se på den nye standarden SQL/MM som vil legge føringen for både hva som kreves og hva som bør integreres av IR-funksjonalitet i en relasjonsdatabase.

5.1. FAST Data Search

5.1.1. Om FDS

FAST Data Search (FDS) er et IR-system utviklet av Fast Search & Transfer ASA. FDS er i dag søkemotoren bak nettsteder som AllTheWeb og Lycos, men FDS utvikles i stor grad som et IR-system for store bedrifter med behov for å integrere og aksessere store mengder informasjon. IBM og Dell er eksempler på bedrifter som bruker FDS.



Figur 5-1: Oversikt over FDS

5.1.2. Oversikt over FDS

FDS består av mange komponenter. Noen av dem er basiskomponenter i ethvert IR-system, slik som beskrevet i 4.4, mens andre er komponenter som er utviklet med tanke på å dekke dagens IR-behov hos store bedrifter og på Internett. Modulene er delt inn i fire

hovedområder, informasjonsinnhenting, dokumentprosessering, spørreoptimalisering / resultatprosessering, og informasjonssammenligning.

5.1.3. Informasjonsinnhenting.

Systemet kan indeksere data fra flere kilder, som for eksempel websider som hentes opp via en crawler, filer som ligger på en bedrifts intranett og filservere, eller tekst som kommer fra databaser. Informasjon kan hentes inn på to måter, enten ved at FDS leter etter informasjon (pull) eller at informasjon blir lagt inn i FDS fra en ekstern kilde (push)

FAST Web Crawler

> Lokaliserer og henter eksterne filer på webtjenere. Webcrawleren bruker hyperlinker ut i fra en gitt URL for å "grave" seg videre gjennom denne. Crawlingen skjer ved et gitt tidsintervall, slik at endringer detekteres og dokumenter enten legges til, slettes eller oppdateres. Dokumenter på over 225 forskjellige formater kan innhentes og oversettes til strukturerte data som kan sette inn i FDS.

FAST File traverser

> Traverserer lokale eller eksterne filområder for filer på angitte områder. Dokumenter av ulike formater sendes så videre inn i FDS.

FAST Database Connector

> Her hentes innhold fra en Oracle-, MS SQL- eller DB2-database via klient-API og sendes videre inn i FDS for å gjøre relasjonsdatabaser søkbare. Dette kan skje ved jevne intervaller. Hvis en krever en tettere integrasjon som gjør at søkemotoren alltid er oppdatert (push), må dette skje manuelt ved hjelp av innholds-API.

Content API

> Ved hjelp av innholdsgrensesnittet kan man fra Java, C++ eller COM lage egne rutiner for å "pushe" data inn til FDS. Det er denne teknikken vi har brukt i vår implementasjon for å dytte data fra databasen og inn i FDS.

5.1.4. Dokumentprosessering

Etter at informasjon er hentet inn ved hjelp av en av modulene over, blir dokumentene prosessert gjennom et prosesseringsrammeverk, *FAST Document Processing Framework*, som består av flere "pipelines". Hver pipeline består av ulike steg og hvilken pipeline som brukes, er avhengig av datatypen til dokumentet. Stegene kan tilpasses til hva slags data som blir sendt inn og hva slags resultat som er ønsket. Før dokumentprosesseringen blir alle dokumentene representert på et internt dokumentformat som består av flere attributter/elementer. Dette formatet defineres i en *indeksprofil* etter hva slags type informasjon som skal lagres. En typisk representasjon av et webdokument vil f.eks. inneholde 'title' og 'body'. Indeksprofilen definerer også hvilke felter som skal indekseres og hva som skal kunne returneres som resultat.

Stegene i dokumentprosesseringen tar i mot et slikt dokument, manipulerer det på en eller annen måte, for så å sende det videre til neste steg. Dette er illustrert på Figur 4-2 i forrige kapittel.

Dokumentprosesseringen inneholder gjerne steg for dokumentkonvertering, HTML parsering, klassifisering, språkdeteksjon og lingvistiske prosessering. Flere av disse operasjonene tilfører FDS-funksjonalitet som er naturlig å finne i et IR-system (se 4.6). Dette inkluderer:

Lemmatisering

> Lemmatisering utvider ord til alle mulige former av dette ordet, slik som flertallsformer og bøyninger. Det mest vanlige er å utvide substantiver med entalls- og flertallsform, men adjektiver og adverb kan også utvides. Utvidelser skjer kun innenfor ordklasser, slik at substantiver ikke vil bli lemmatisert til adverb. Under indeksering vil ord i dokumentet bli lemmatisert. Spørringen vil da gi treff på alle mulige former av et bestemt ord. Hvis man søker på ordet 'bil', vil dette gi treff i dokumenter som inneholder 'bil' og 'biler'. Lemmatisering er ikke det samme som bruk av jokertegn og utvidelser av ord. Lemmatisering vil derfor *ikke* føre til at søk på 'bil' kan gi treff på 'biller'. Lemmatisering øker sjansen for at de dokumentene du er på jakt etter kommer med i søkeresultatet, men samtidig er det naturlig at de dokumentene som inneholder akkurat den formen du søkte på, blir rangert høyere. Bruk av lemmatisering kan skrus av og på for hver spørring.

Språkdeteksjon

> En av de første analysene som skjer, er språkdeteksjon. FAST DS kan detektere 49 forskjellige språk, og dette legger grunnlaget for mange av de lingvistiske prosessene som kommer senere. I tillegg kan man søke innenfor bestemte språk ved å spesifisere dette i spørringen.

Synonymer

> Når et dokument indekseres, kan det utvides med en egen indeks av synonymer. Bruk av disse egne indeksene kan velges per spørring. For at dette skal føre til bedre relevans bør synonymlisten være domenespesifikk. Dette fordi mange språk, slik som engelsk, inneholder homonymer (ord med ulik betydning, men samme stavemåte), og utvidelse med alle synonymer vil derfor gi mange irrelevante treff. Dette fører til dårligere presisjon.

Egennavngjenkjenning

> En liste med egennavn kan defineres slik at disse gjenkjennes og blir utelatt fra videre prosessering i pipelinen, slik som for eksempel lemmatisering.

Klassifisering og kategorisering

> I FDS kan man benytte seg av både overvåket og uovervåket kategorisering, og dokumentene kan klassifiseres på grunnlag av predefinerte verdier eller sammenlignes dynamisk. Det er mulig å søke innefor bestemte kategorier for å øke presisjonen i spørringen. Dette øker sjansen for relevante treff. Man kan også grave seg ned og se alle dokumentene under en spesiell kategori. Kategorisering brukes også senere for å gruppere like treff i resultatet.

Resultatet av dokumentprosesseringen blir så sendt videre inn til søkemodulen og filtreringsmodulen for indeksering og sammenligning.

På samme måte som dokumenter, blir også spørringer prosessert før de sendes til søkemotoren. Vanlige steg i en spørreprosessering er for eksempel stavekontroll og anti-frasing.

5.1.5. Spørreoptimalisering og resultatprosessering

FAST Search engine er delt inn tre moduler

- **Dispatcher Module** som distribuerer, administrerer og setter sammen resultatet av spørringer over flere noder.
- **Search Module** som er komponenten som eksekverer en spørring mot en gitt del av indeksen.
- **Indexer Module** mottar dokumenter fra dokumentprosesseringen og konverterer dem til et binært indeksformat.

FDS baserer seg på store mengder inverterte indekser av forskjellige typer. Hovedhensikten med dette er å kunne gjøre raske søk i store mengder tekst ved hjelp av oppdeling og parallellisering av spørringen. Indeksene kan dupliseres over flere noder for å øke antall spørringer per sekund. Tilsvarende kan disse indeksene deles i mindre biter og fordeles utover flere noder for å øke oppslagshastigheten (redusere forsinkelsen) for enkeltsøk. I tillegg til at store mengde inverterte filer eller indekser gjør søk effektive, raske og skalerbare, ligger også mye av funksjonaliteten til FDS i disse inverterte filene, nettopp for å gjøre søk mer effektivt.

Når et dokument indekseres, vil det bli gjenstand for mange forskjellige lingvistiske analyser, og flere indekser vil bli generert opp på grunnlag av disse analysene. De fleste slike lingvistiske analyser av tekst foregår med andre ord når dokumentet prosesseres, og ikke under spørringer, noe som opplagt effektiviserer spørringene og reduserer spørretiden. En spørring er derfor i stor grad oppslag i de inverterte filene. Ofte blir dette en avveining mellom bruk av diskplass og prosessorkraft. Inverterte filer tar mye plass, i hvert fall når man snakker om å indeksere alle websider på Internett. Det kan derfor være bedre å løse dette etter at spørringen er kjent, men det vil da kreve mer prosessorkraft å gi svar på spørringen.

Gjennom *FAST Query API* kan spørringer sendes fra klientapplikasjoner gjennom C++, Java eller COM. Spørrespråket til FDS er et boolsk spørrespråk som ligner spørrespråket til andre søkemotorer. En viktig egenskap ved spørrespråket til en søkemotor er at det skal være enkelt, slik at alle kan bruke det. Selv uten kunnskap om boolsk algebra, skal det være mulig å lage enkle spørringer kun ved å oppgi et eller flere søkeord. Med litt mer kunnskap om spørrespråket og boolsk algebra skal har man muligheten til å uttrykke seg mer presist. Man kan karakterisere et slikt spørrespråk som ad-hoc, fordi det lar seg bruke bare ved å oppgi et søkeord, men det kan også beskrives som et spørrespråk, fordi det gjør det mulig å formulere mer presise spørringer:

Ord, fraser og numeriske verdier

> En spørring kan settes sammen av flere enkeltord eller fraser som skal søkes på i sin helhet. Anti-frasing, det vil si å fjerne vanlige stoppord og andre prefikser fra fraser, øker sjansen for at de relevante dokumentene kommer med. Man kan også søke på numeriske verdier ved bruk av operatorene <, >, <>, =.

Jokertegn

Jokertegn kan brukes for å betegne at en setning enten starter eller slutter med en eller flere forskjellige tegn.

Boolsk algebra

Boolske operatorer som AND, OR og ANDNOT kan brukes for å bygge opp et boolsk spørreuttrykk. Uttrykkene kan grupperes i parenteser for å gi dem den rette presedensen.

Som tidligere nevnt går også spørringene gjennom en slags dokumentprosessering. De viktigste ekstra stegene i denne i forhold til den vanlige pipelinen er:

Query analyse

Stavekontroll blir automatisk påført egennavn, og egennavnet blir automatisk behandlet som en frase.

Stavekontroll

Stavekontroll kan automatisk bli påført ord i spørringer. Ord blir erstattet av ord i en ordbok. Ord som opptrer ofte i ordboken, blir favorisert. Stavekontrollen gjelder også feilstavede fraser.

Den ferdig prosesserte spørringen blir sendt til *Dispatcher*-modulen som sender spørringen videre ned til søkemodulen på de forskjellige nodene. En av hovedstyrkene til FDS ligger nettopp i den skalerbare arkitekturen. Store datavolum er delt opp over flere noder, og spørringen blir deretter prosessert i parallell. Ved å legge til flere rader med noder, og distribuere spørringene over flere rader, sikres rask responstid uavhengig av belastningen (se Figur 4-5).

Resultatprosessering er siste steg før presentasjon for brukeren. Måten resultatet presenteres på og hva det skal innholde, kan variere.

Rangering

Rangeringen består av en statisk del, som alltid vil være den samme for et gitt dokument. Dette kan inkludere statiske rangeringsverdier for på forhånd å definere at et dokument er viktigere enn andre. Den dynamiske delen vil variere fra spørring til spørring, slik som for eksempel antall forekomster av søkeordene i dokumentet. FDS bruker også en rekke andre teknikker, slik som ordnærhet, hvilken form søkeordene og ordene i dokumentet har og så videre. En del av dette kan slås av og på for hver spørring.

Ved hjelp av nøkkelordet RANK kan man også angi ord man vil tillegge mer vekt enn andre. En spørring som for eksempel ’cat AND dog RANK dog’ vil gi treff på alle dokumenter med ordene katt og hund, men vil rangere dokumentene med vekt på hund. Man har også mulighet for å angi hvor mye de forskjellige ordene skal vektes f.eks. ved ’cat!30 AND dog!70’ vil gi 70% vekt på hund, eller ’cat! AND dog!!!’ vil gi ’hund’ tre ganger vekten til ’katt’.

Dokumentsammendrag

> Dokumentsammendrag gir ofte brukeren bedre oversikt over resultatet enn hvis man ble presentert for selve dokumentet. FDS kan lage enten et statisk eller et dynamisk sammendrag av dokumentet. Statiske sammendrag er sammendrag som er laget under dokumentprosesseringen for å gjenspeile dokumentets innhold. Et dynamisk sammendrag er relatert til søket, og gir ofte brukeren et mer relevant sammendrag. De viktigste tekstsegmentene i forhold til det man søker på, blir tatt med.

Gruppering og kategorisering av like dokumenter.

> Søkeresultater kan også bli gruppert og kategorisert on-the-fly basert på likheter mellom dokumentene. Dette kan være til hjelp dersom søkeresultatet blir stort og består av flere lignende dokumenter. FDS bruker vektorrommodellen for å sammenligne dokumenter. Denne klassefiseringen er uovervåket.
>
> I FDS har man også mulighet til å søke etter lignende dokumenter basert på kategorisering, såkalte 'Find similar'-søk.

5.1.6. FAST Filter Engine

FAST Filter Engine er den siste hovedkomponenten i FDS. Dette er et varslingssystem basert på real-time-filter-teknologi, der man har flere sett med filterbetingelser eller varslingsspørringer, og sammenligner dem med innkommende data som kommer fra dokumentprosesseringen. Disse filterbetingelsene kan settes ved hjelp av *FAST Alert API.* Resultatet kan være nyhetsartikler, aksjekurser eller andre dokumenter prosessert i FDS. Dataene kan konverteres og sendes til et stort antall brukere, applikasjoner og systemer. Applikasjonsområdet for slike filter kan for eksempel være overvåkning av aksjekurser, værvarsel, eller nyheter.

5.2. SQL

I relasjonsdatabaser er SQL det desidert mest brukte spørrespråket, og omtrent alle relasjonsdatabaser implementerer denne standarden i større eller mindre grad. SQL blir derfor gjerne omtalt som "the intergalactic data speech". SQL er foreløpig i sin fjerde versjon. Versjonene av SQL er kjent som SQL-86, SQL-89, SQL-92 og til sist SQL:1999. De fleste relasjonsdatabaser i dag implementerer det meste av SQL-92, og er i ferd med å implementere SQL:1999, men ingen databaser har foreløpig implementert denne helt.

SQL:1999 er delt opp i et sett kjernefunksjoner (Core SQL), og pakker med tilleggsfunksjonalitet. En av disse er SQL/MM (SQL MultiMedia) som igjen er oppdelt i flere deler, hvorav en del om fritekst. Denne beskriver jeg nærmere i 5.3.

Støtten for fritekstsøk i SQL-92 og Core SQL begrenser seg til bruk av LIKE/SIMILAR. Jeg vil først kikke litt på disse, for så å se videre på noen produktspesifikke utvidelser gjort for å gi bedre støtte for fritekstsøk i kommersielle SQL-databaser. Deretter vil jeg ta for meg SQL/MM del 2 som omhandler fritekst.

5.2.1. LIKE/SIMILAR

I Core SQL:1999/SQL-92 finnes det to operatorer for å gjøre mønstergjenkjenning i tekstatributter, LIKE og SIMILAR. I LIKE (SQL-92) bruker man tegnet '%' som jokertegn for en sekvens av tegn eller bokstaver og '_' er jokertegn for ett tegn. Man kan dermed bygge opp spørringer for enkle tekstsøk slik:

```
SELECT * FROM article WHERE text LIKE '%dog%' AND text NOT LIKE
'%cat_'
```

I SIMILAR TO (SQL:1999) har man muligheten til å bruke styrken i regulære uttrykk istedenfor de mer enkle jokertegnene i LIKE. Dette gjør det enklere å definere presise spørringer som gjenkjenner nøyaktig det tekstmønsteret man er på jakt etter.

LIKE/SIMILAR gjør altså at man i SQL har mulighet til å gjøre enkle søk i tekst ved å kunne definere presise mønstre på hva man vil søke etter. Man har ikke støtte for mange av de mulighetene vi kjenner igjen fra IR-systemer, slik som lingvistiske analyser, kontekstsøk, og rangering. Dette gjør at disse operatorene er uegnet når man vil søke i lengre tekst og fritekst, fordi de er først og fremst er ment å gi mulighetene for søk i korte tekststrenger, slik som datatypene char og varchar i en database. Fordi det ikke gjøres noe for å skille ut og rangere relevante dokumenter i forhold til den spørringen en gjør, kan man ikke definere dette som fritekstsøk. Et annet problem med disse operatorene er effektiviteten. Et slik mønstersøk er i de aller fleste tilfeller implementert ved et sekvensielt søk gjennom teksten for å finne det definerte mønsteret. Siden databaser tradisjonelt ikke er beregnet på lagring av større mengder tekst, tilbyr ikke disse noen passende datastrukturer for dette, slik som f.eks. inverterte filer. I en fritekstsammenheng vil selvfølgelig et søk uten indekser være helt utenkelig når man snakker om gigabytes eller terabytes med data.

5.2.2. Ikke-standardiserte løsninger.

Fordi SQL ikke tilbyr muligheter for fritekstsøk, har de største databaseprodusentene kommet med sine egne løsninger og utvidelser for å møte den stadige økende interessen for integrasjon av fritekstsøk og relasjonelle databaser. Jeg vil her kort introdusere løsningene til de aller mest utbredte databasene: Oracle 's Oracle9i, IBM's DB2 og Microsoft's SQLServer 7.

Oracle Text[16]

Oracle har i de siste årene hatt en meget sterk markedsposisjon innefor relasjonsdatabaser. Deres Oracle Text teknologi, som er en del av Oracle 9i databasen, integrerer mye av den funksjonaliteten som vi finner igjen i søkemotorer.

Selve indekseringsprosessen er bygd opp som en pipeline der resultatet er en invertert indeks. Som datakilder kan den indeksere alle tekstdatatyper i databasen (char, varchar, clob), men den kan også indeksere dokumenter som ligger på fil utenfor databasen eller på Internett. Pipelinen består blant annet av dokumentfiltrering som gjenkjenner diverse typer dokumentformater, og oversetter disse til et indekserbart format, og en prosess som deler opp teksten i seksjoner basert på dokumentets oppbygning slik at man kan støtte operasjoner som kontekst-/nærhetssøk. Så deles dokumentet opp i ord som til slutt blir indeksert. Sammenlignet med FDS, inneholder Oracles dokumentprosessering noen av de samme elementene men er mye enklere og inneholder færre steg.

Spørringer mot indeksen er integrert i SQL gjennom en CONTAINS-funksjon som man kan bruke i WHERE-delen. Syntaksmessig er denne veldig lik måten vi har valgt å gjøre det på, der man angir hvilket attributt man vil søke i og et søkeuttrykk:

```
SELECT * FROM article WHERE CONTAINS(text, 'dog NOT cat')
```

Som søkeuttrykk kan man bruke ord eller fraser sammen med en rekke operatorer. Noen av de viktigste er:

- Kontekstspørringer, slik som 'kw' NEAR 'kw', eller ord WITHIN SENTENCE/PARAGRAPH.
- Boolske operatorer, AND, OR og NOT.
- Ordrelasjoner slik som synonymer og BT, NT. (slik som beskrevet i 4.6.2)
- Fuzzy spørringer, soundex, og en spesiell ABOUT-operator som øker antall treff på en spørring, det vil si økt gjenfinning.

Fra resultatet kan man hente ut både rangering og sammendrag. Oracle Text kan også gjøre flere ting med resultatet slik som å markere hvilke ord i dokumentet som var årsaken til resultatet.

Når det gjelder effektivitet og skalerbarhet, er dette vanskelig å vurdere uten å teste løsningen. Databaser har tradisjonelt sett en meget monolittisk arkitektur, men Oracle hevder at den lett kan fordeles til flere noder. Til en viss grad er nok dette sant, men utfordringene med å skalere en database er mange, og det synes nokså klart at dette ikke vil fungere like godt som i en søkemotor som i mindre grad trenger å ta hensyn til ACID-egenskaper.

DB2 Text Information Extender[14]

DB2 Text Information Extender utvider IBM's DB2-database med fulltekstsøk muligheter. Løsningen ligner i stor grad på den som Oracle tilbyr, men er generelt litt enklere. Den har blant annet ingen dokumentfilterfunksjon, noe som gjør at man kun kan indeksere et fåtall typer dokumenter, (tekst, html, xml) og en indeks støtter bare en type av gangen. Mulighetene i spørrespråket er også mindre, men også her har en mulighet for kontekstspørringer, boolske operatorer, fuzzy spørringer og synonymer. DB2 har også muligheten til å hente ut en rangering av dokumentene, men ikke sammendrag.

```
SELECT * FROM article WHERE CONTAINS(text, "dog" & NOT "cat")
```

Microsoft Full Text Search[13]

Microsoft Full Text Search tilbyr mye av den samme funksjonaliteten som de to foregående, men løsningen er mye enklere enn i Oracle. Når det gjelder integrasjon mot SQL, har Microsoft i likhet med Oracle, en contains funksjon, men utvider den til også å kunne brukes i FROM-delen av spørringen. Ved bruk av containstable-/freetexttable-funksjonene returneres en tabell med to attributter: En rangering og en primærnøkkel for tuppelet, og en OID[1] for de tuplene som oppfylte fritekstsøket. Oracle returnerer også dette (bak i det skjulte), men ved å flytte funksjonen til WHERE-delen gir det brukere en langt enklere og penere måte å gjøre ting på. Grunnen til dette er at man da slipper å

[1] Object Identifier – I dette tilfelle en unik ID for tuppelet.

"trylle" frem attributter som egentlig ikke finnes i FROM-delen av spørringen. Denne muligheten har vi tilpasset til vår egen implementasjon som er beskrevet i 6.6.

Et eksempel på en slik spørring i MS SQL er slik:

```
SELECT *
FROM CONTAINSTABLE(article, text, 'dog NOT cat') c, article d
WHERE c.[KEY]=d.docNo ORDER BY K.RANK
```

I vår implementasjon av CONTAINSTABLE returnerer vi imidlertid også originaltuppelet, slik at denne eksplisitte joiningen med den originale tabellen er unødvendig.

Ingen av disse produktspesifikke løsningene bygger forløpig på SQL:1999 og ADT (beskrevet i neste avsnitt) slik som SQL/MM gjør, men bruker foreløpig SQL-92-syntaks med bruk av funksjoner i SELECT-, FROM- eller WHERE-delen av spørringene. Oracle har den klart mest avanserte løsningen, og implementerer det meste av den funksjonaliteten som er i SQL/MM-standarden. Microsoft og IBM har det viktigste man trenger for å gjøre fritekstsøk, og Microsoft har også etter min mening den beste integrasjonen med SQL.

5.3. SQL/MM

"SQL Multimedia and Application Packages"[3] er en ISO/IEC[1]-standard og et tillegg til SQL:1999. SQL/MM er delt opp i forskjellige deler som hver utgjør et slags klassebibliotek av SQL-objekter innenfor et bestemt domene. Hittil består SQL/MM av 5 deler:

- Part 1: Framework – Felles deler og definisjoner
- Part 2: Full-Text
- Part 3: Spatial
- Part 5: Still Image
- Part 6: Data Mining

Den planlagte fjerde pakken, som skulle inneholde generelle matematiske typer og operasjoner, ble skrinlagt tidlig i standardiseringsprosessen.

Grunnen til denne oppdelingen er at man tidlig så behovet for å unngå at standarder kom i konflikt med hverandre, slik som for eksempel ved bruk av nøkkelord. I 1992 kom det et tillegg til SQL kalt SFQL[2], som definerte en rekke fulltekstoperasjoner i SQL. Denne standarden brukte blant annet begrepet CONTAINS, for å betegne at en tekst inneholdt et ord eller en frase. Andre domener la imidlertid andre meninger i dette begrepet og implementasjonen ble da annerledes.

I SQL:1999 ble det derfor lagt til mulighet for å definere såkalte brukerdefinerte datatyper (ADT[3]). Til disse typene er det mulig å definere atributter og metoder. På denne måten kan en CONTAINS-metode til en fulltekst datatype ikke behøve å ha den samme

[1] International Organization for Standardization, International Engineering Consortium
[2] Structured Full-text Query Language
[3] Abstract Data Type

betydningen og implementasjonen som CONTAINS-metoden til en geografisk datatype. Som navnet på standarden tilsier, er SQL/MM en samling av slike datatyper, eller pakker med slike datatyper, beregnet for forskjellige applikasjonsområder. I standarden er typene og metodene definert i SQL eller med en definisjon av implementasjonen, det vil si hva slags resultat som er forventet. For å være konform med standarden må en implementasjon ha samme effekt som definert i denne, men ikke nødvendigvis være implementert på samme måte. Ingen har enda hevdet å være kompatibel med noen deler av SQL/MM, men flere av de store databaseleverandørene, slik som Oracle, planlegger dette. Jeg vil her se nærmere på del 2 om fulltekst.

De to viktigste datatypene i pakken er[18]:

- FullText
- FT_Pattern

Fulltext har to atributter: språk og innhold. Språket er viktig for den videre prosesseringen og tolkningen av innholdet, slik at denne alltid må defineres enten direkte eller indirekte. Innholdet er selve friteksten. På denne typen er det definert en rekke metoder, men de to viktigste er:

text.Contains(FT_Pattern search_expression)

> Denne metoden tar i mot et søkeuttrykk FT_Pattern type, og returnerer 1 hvis teksten inneholder dette søkemønsteret.

text.score(FT_Pattern search_expression)

> Denne returnerer en rangering av teksten i henhold til hvor godt søkemønsteret er inneholdt i teksten.

Disse to metodene tar altså i mot et søkemønster av typen FT_Pattern. Dette er rett og slett en definisjon av fulltekstspørrespråket til SQL/MM. BCNF-definisjonen av hvordan et slikt søkemønster kan være er over 3 sider langt, men beskriver til gjengjeld de mulighetene en har i spørrespråket. Kort oppsummert tilbyr SQL/MM disse mulighetene:

Boolsk søk etter ord og fraser

> Et søkeuttrykk kan bestå av ord eller fraser satt sammen til et boolsk uttrykk med operatorene AND(&), OR(|) og NOT. Man kan også benytte jokertegn både i ord og i fraser.

Lemmatisering

> Med nøkkelordene STEMMED FORM OF <word> kan man angi at man vil søke på alle mulige former av et ord.

Kontekstsøk

> Man kan angi om man vil søke etter ord som ligger i nærheten av hverandre eller innefor en viss kontekst. Mulige kontekster er bokstaver, ord, setninger og avsnitt. Man kan også angi hvor langt unna ordene skal være fra hverandre. Dette gjøres med <tokenlist1> NEAR <tokenlist2> WITHIN <distance> <unit>, der <unit> er avstanden angitt med et tall, og <distance> er konteksten. Kontekstsøk kan også angis på formen <tokenlist1> IN SAME <distance> AS <tokenlist2>.

Ordrelasjoner/synonymer

Dette omfatter søk etter ord som er relatert til et annet ord. Her kan ordrelasjonen være synonymer, BT, NT og så videre:

```
SELECT * FROM article WHERE text.CONTAINS(
'THESAURUS "computer science" EXPAND SYNONYM TERM OF "list"')
```

I dette eksempelet velger man å bruke en bestemt ordliste, nemlig "computer science", til å utvide ordet "liste" med synonymer. Dette kan for eksempel være sekvens, array osv. Grunnen til dette er at synonymer sjelden har noen nytte utenfor et spesifikt domene. Slik er det også i FDS, men der kan ikke ordlisten velges for hver spørring.

Spørringer for økt gjenfinning og relevans.

Man kan søke etter ord som staves eller høres like ut på denne måten: SOUNDS LIKE <word> og FUZZY FORM OF <word>. Man kan også søke etter tekst som handler om noe spesielt: IS ABOUT <word or phrase>

Et eksempel på en SQL/MM spørring kan være noe slikt:

```
SELECT * FROM articles WHERE text.CONTAINS(
'STEMMED FORM OF "cat" NEAR "dog" WHITIN 2 SENTENCES'
);
```

SQL/MM er foreløpig en papirstandard, og spørsmålet er om den noe gang kommer til å bli noe annet. I skrivende stund er det som sagt ingen databaser som har hevdet kompatibilitet til noen deler av standarden, og planer om det er kun nevnt i forsiktige ordelag. Oracle hevder at de planlegger å implementere del 3 av standarden, men nevner ingen ting om fritekstdelen. Standardiseringsarbeidet er imidlertid i stor grad drevet frem av kommersielle aktører. Oracle er blant annet med i komiteen som lager SQL/MM, og vi ser også at Oracle allerede tilbyr det meste av funksjonaliteten slik som den er definert i SQL/MM Part 2. Det kan derfor være forretningsmessige grunner til at de ikke har valgt å gjøre det slik som i standarden. Det bør egentlig være liten tvil om at det er planer om å gjøre dette til en reell standard i likhet med resten av SQL, og den er som kjent i aller høyeste grad en reell. Etter min mening vil SQL:1999 og SQL/MM, og da spesielt fritekstdelen, sannsynligvis bli like reell, tatt i betraktning den fokus som dette temaet har fått de siste årene.

5.4. Sammenligning

Jeg har hittil i dette kapittelet beskrevet IR-funksjonalitet i forskjellige SQL-databaser og i FDS. Det å skulle sammenligne forskjellige IR-systemer og deres funksjonalitet er vanskelig på flere måter Jeg vil allikevel gi en oppsummering av disse forskjellige systemene. Hensikten med dette er å drøfte hvorvidt en søkemotor, slik som FDS, er egnet for integrering i en relasjonsdatabase. Med det mener jeg at den må ha støtte for den vesentligste delen av funksjonaliteten som vi finner i SQL/MM-standarden og i eksisterende databaser. En søkemotor har andre behov enn det et IR-system i en relasjonsdatabase har. Samtidig er det nærliggende å spørre seg om mye av den samme teknologien kan brukes. Kan FDS brukes til å implementere SQL/MM, og hvis ikke, hva er det som konkret skiller disse to IR-områdene?

En av grunnene til at en sammenligning er vanskelig, er fordi at SQL/MM er en standard som ikke er implementert, og standarden overlater naturlig nok en del til implementasjonen. Vi kan likevel konkludere med følgende:

- FDS har et spørrespråk som er ”ad-hoc” i sin natur, med få krav til dets syntaktiske oppbygning. Den er rettet mot en type bruk der en sjelden formulerer vanskeligere spørringer enn to nøkkelord og en boolsk operator. Et annet typisk trekk som er knyttet til spørrespråket til FDS, er at mange valg og funksjonalitetsområder ikke spesifiseres i spørringen og dermed for en del av spørringen, men derimot *for hele spørringen*. Man mister mye av styrken i SQL/MM der man kan lage mer presise spørringer, men oppnår samtidig brukervennlighet. Noe av funksjonaliteten til FDS er også laget for å rette opp feil fra brukerens side, slik som skrivefeil. Dette er områder som SQL/MM ikke spesifiserer nærmere, og det vil antagelig også være mindre interessant.
- Det er også flere områder hvor FDS mangler aktuell funksjonalitet, antagelig på grunn av at disse tingene simpelthen er mindre interessante for det applikasjonsområdet som FDS hittil har blitt brukt til.
- På andre området har FDS mer utviklet fritekstsøkfunksjonalitet. Mange av disse områdene, slik som kategorisering og rangering, sier standarden lite om. Funksjonalitet for å vekte nøkkelord ulikt, søke over flere felter med ulik vekt, samt statisk styre rangeringen er eksempler på dette. Det siste tilfelle vil typisk være opp til implementasjonene, mens de to foregående ville det kreves støtte for i spørrespråket .

Tabell 5-1 gir en tabularisk oversikt over de viktigste funksjonelle områdene slik som jeg definerte dem i 4.6, og forskjellene mellom de systemene jeg har beskrevet i dette kapittelet.

FDS mangler altså både funksjonalitet og et presist spørrespråk for å kunne implementere SQL/MM, men et subsett, og kanskje det viktigste subsettet, vil kunne være mulig å få til. Den viktigste IR-funksjonaliteten til en relasjonell database vil antagelig være det samme som for en søkemotor, nemlig søk på nøkkelord, boolske operatorer og rangering av de dokumentene som ble funnet relevante.

	FDS	SQL/MM	Oracle Text
Tekstanalyse og lingvistikk, se 4.6.1			
Stemming/lemmatisering	Ja	Ja	Ja
Stoppord	Vekting[1]	Ja	Ja
Språkdeteksjon	Ja	-[2]	Ja
Ordrelasjoner og kryssreferanser, se 4.6.2			
Synonymer	Ja[3]	Ja	Ja
BT, NT.....	Nei	Ja	Ja
Fuzzy søk	Nei	Ja	Ja
Soundex søk	Ja	Ja	Ja
About søk	Ja	Ja	Ja
Tekstformatering og oppbygning, se 4.6.3			
Fraser	Ja	Ja	Ja
Nærhet-/kontekstsøk	Nei[4]	Ja	Ja
Mønstersøk	Ja[5]	Ja	Ja
Analyse av spørring, se 4.6.4			
Stavkontroll	Ja	-[2]	Nei
Egennavngjenkjenning	Ja	-[2]	Nei
Anti frasing	Ja	-[2]	Nei
Segmentering	Ja	-[2]	Nei
Kategorisering, se 4.6.5			
Overvåket	Ja	-[2]	Ja
Uovervåket	Ja	-[2]	Nei
Søk i kategorier	Ja	-[2]	Nei
Find similar	Ja	-[2]	Nei
Gruppering av like treff	Ja	-[2]	Ja
Drill-Down	Ja	-[2]	Nei
Relevans og rangering, se 4.6.6			
Relative term frequency	-[6]	-[2]	-[6]
Inverse document frequency (IDF)	Ja	-[2]	Ja
Ordnærhet	Ja	-[2]	Nei
PageRank eller lignende	Ja	-[2]	Nei
Vektet søk	Ja	-[2]	Nei
Rangering, sammendrag, se 4.6.6 og 4.6.7			
Rangering	Ja	Ja	Ja
Statisk sammendrag	Ja	Nei	Nei
Dynamisk sammendrag	Ja	Nei	Ja

Tabell 5-1 Sammenligningstabell over IR funksjonalitet i FDS, SQL/MM og Oracle DB

[1] Stoppord nedvektes, men kan brukes i søk.
[2] Ikke beskrevet/angitt. Kan være avhengig av implementasjon.
[3] Kan ikke angis per spørring
[4] Nærhet fører til høyere rangering
[5] Kun bruk av *
[6] Unødvendig ved bruk av IDF

Kapittel 6. Implementasjon

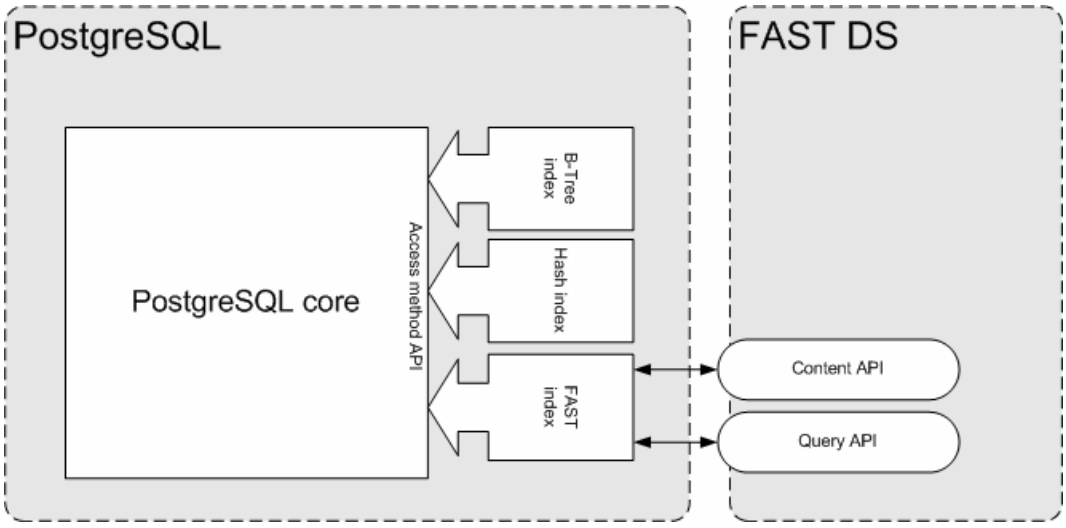
6.1. Målet med implementasjonen

Målet med implementasjonen var å integrere Fast's søkemotor i PostgreSQL som en indeksaksessmetode på lik linje med andre indeksaksessmetoder i databasen. Vi ønsket i første omgang kun å bruke eksisterende SQL-kommandoer og syntaks i PostgreSQL, slik som:

```
1) CREATE TABLE sample (id int4, data text);
2) CREATE INDEX sample_index ON sample USING fast (data);
3) INSERT INTO sample VALUES (1, 'sample text');
4) SELECT FROM sample WHERE data CONTAINS 'sample';
```

Her vil kommandoen i første linje lage en tabell med et vilkårlig stort tekstfelt egnet for fritekst. I neste kommando lager vi en indeks over denne kolonnen ved bruk av vår egendefinerte aksessmetode *fast*, til erstatning for B-tre som blir valgt hvis man ikke definerer noe annet. I steg 3 setter vi inn et tuppel med fritekst. Den siste *SELECT*-setningen skal returnere de tupler som inneholder søkeordet, i dette tilfelle tuppelet over. Her bør man tillate et vilkårlig boolsk uttrykk på samme måte som man er vant med ved bruk av en søkemotor på Internett. *CONTAINS* er tenkt som en egendefinert operator, som betegner at vi ønsker å ha ut de dokumentene som inneholder ordene i søkeuttrykket. Databasen må selv komme til konklusjonen at den må bruke den definerte fast-indeksen for å finne svaret, hvis ikke vil den ikke klare å svare på fritekstsøk-spørring.

Figur 6-1 viser en oversikt over de viktigste komponentene i de systemene som skal knyttes sammen. Indeksaksessmetoden vil ikke lagre noen informasjon innenfor databasen slik som de andre indeksaksessmetodene gjør, men isteden kommunisere med FDS og dets grensesnitt.



Figur 6-1: Oversikt over implementasjonen

6.2. PostgreSQL

Jeg vil i de neste to avsnittene gå nærmere inn på to egenskaper ved PostgreSQL som gjør det mulig å utvide databasen med en ny aksessmetode.

6.2.1. Systemkataloger

PostgreSQL lagrer metadata og systeminformasjon i *systemkataloger*. Dette er informasjon om databasen og dens tilstand, blant annet oversikt over alle relasjoner, tabeller, indekser, attributter, operatorer, funksjoner, statistikk for beregning av kjøreplaner, hvilke språk og indeksmetoder som er tilgjengelige og en hel del mer. Disse systemkatalogene er i virkeligheten helt vanlige tabeller, prefikset med pg_, og kan manipuleres slik som alle andre tabeller. Det er nettopp dette som gjør at PostgreSQL er lett å utvide og manipulere, fordi mange utvidelser og endringer kan gjøres uten endringer i databasens kildekode, kun ved endringer i systemkatalogene. Dette fører til at utvidelse gjøres under kjøring, uten å måtte rekompilere systemet eller å ha tilgang til kildekoden. De viktigste systemkatalogene for vår implementasjon var følgende:

pg_proc

> Inneholder en oversikt over alle funksjoner som er tilgjengelige i PostgreSQL. Denne kan manipuleres indirekte ved hjelp av SQL-kommandoene CREATE/DROP FUNCTION.

pg_am

> Alle indeksaksessmetoder og deres egenskaper.

pg_operator

> Denne katalogen inneholder en oversikt over alle operatorene i systemet og hvordan de er implementert. En operator har et operatortegn, slik som f.eks. =, en eller to datatyper som argumenter ettersom det er en unær eller binær operator, og en funksjon som utfører operatoren. Tabellen behøver normalt ikke modifiseres direkte, men gjennom SQL-kommandoene CREATE/DROP TYPE.

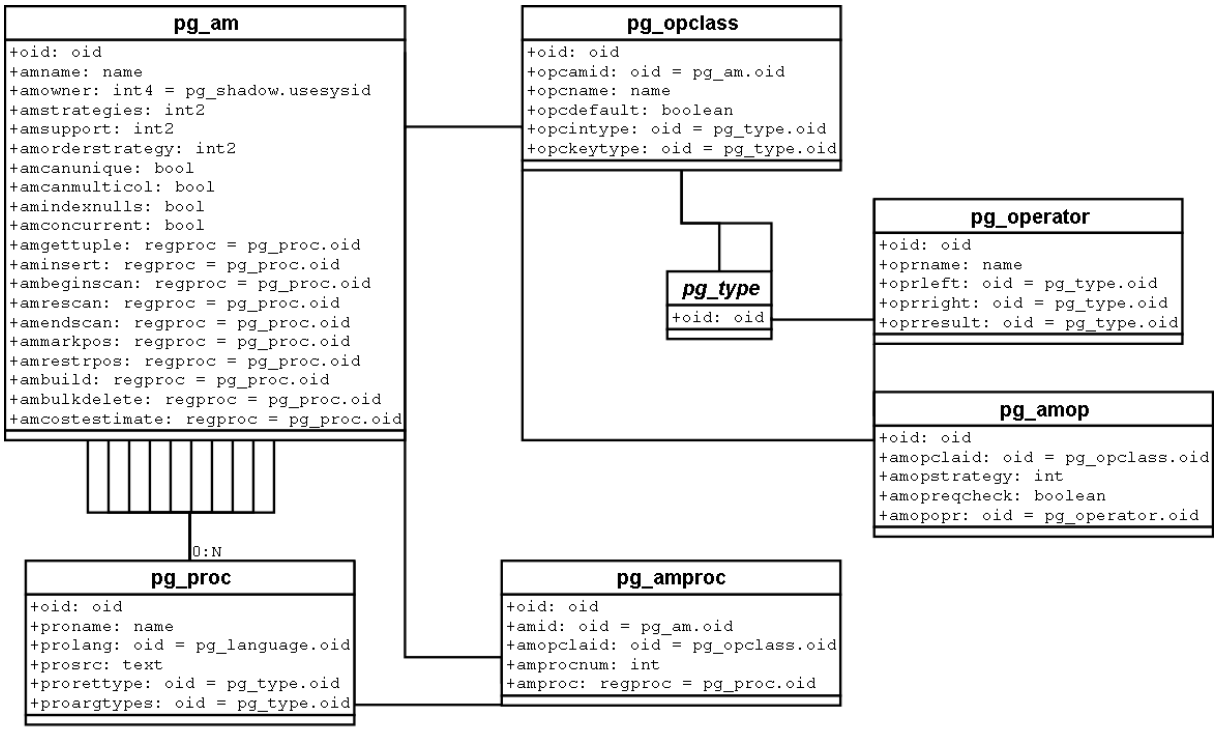
pg_amop

> Bindingen mellom en operator og en aksessmetode.

pg_opclass

> Denne katalogen definerer indeksaksess-operatorklasser. Sammen med pg_amop utgjør denne katalogen en operatorklasse som er bindeleddet mellom datatyper, aksessmetoder og operatorer. For at systemet skal kunne vurdere bruk av indekser i forbindelse med spørringer må aksessmetoder som støttes ved bruk av en bestemt operator og datatype være kjent. Hvis det f.eks. gjøres en spørring av typen
>
> ```
> SELECT data FROM table WHERE id = 1
> ```
>
> må likhetsoperatoren for to integer-kolonner knyttes sammen med f.eks. B-tre-aksessmetode for at denne spørringen skal kunne optimaliseres av en eventuell B-tre-indeks på kolonnen *id* i tabellen *table.*

Katalogen behøver normalt sett ikke modifiseres direkte, men gjennom kommandoen CREATE/DROP OPERATOR CLASS. Operatorer knyttes til ved hjelp av katalogen pg_amop.



Figur 6-2: De viktigste systemkataloger i PostgreSQL med hensyn på vår implementasjon.

6.2.2. Dynamisk lasting av kode

I PostgreSQL kan man dynamisk laste inn kompilert C-kode, eller kode som er kompatibel med C slik som C++, mens databasen kjører. På denne måten kan man også gjøre utvidelser som trenger ny kode uten å måtte rekompilere eller endre i koden til selve databasen. Vi benyttet oss av dette når vi skulle legge til de grensesnittsfunksjonene som var nødvendige for å lage en ny indeksaksessmetode.

6.3. Grensesnitt i PostgreSQL for en ny aksessmetode

Indeksaksessmetodene i PostgreSQL er definert i systemkatalogen *pg_am*. For å legge til en ny indeksaksessmetode, må vi legge til et tuppel i denne tabellen.

Et tuppel i *pg_am* ser i hovedsak slik ut:

amnames

Navnet på indeksaksessmetoden.

amstrategies

Dette er et tall på hvor mange operatorstrategier denne aksessmetoden støtter, og det vil si hvor mange operatorer denne aksessmetoden kan brukes med. Et b-tre støtter fem (>, <, >=, <=, =), mens en hash støtter kun en (=). Vi vil, i hvertfall til å begynne med, kun støtte en strategi, nemlig *CONTAINS*.

amsupport

Dette atributtet angir antall støttefunksjoner denne aksessmetoden må ha. Støttefunksjoner til en aksessmetode er funksjoner som er implementert forskjellig ettersom hvilken datatype aksessmetoden skal indeksere. Hvis man legger til en ny datatype og vil ha mulighet til å indeksere denne typen med for eksempel et b-tre, må man implementere en funksjon som definerer ordningsrelasjonen mellom to attributter. Et generelt B-tre har ingen mulighet til å sammenligne brukerdefinerte datatyper, til dette trengs det kjennskap til typen. På samme måte krever også hash en støttefunksjon, nemlig en funksjon som genererer hash-verdien for en nøkkel av en gitt type. Foreløpig tar vi kun sikte på å indeksere typen text, og vi trenger derfor ingen støttefunksjoner. Dette er imidlertid aktuelt ved en utvidelse.

amorderstrategy

Dette er et tall som angir hvilken operator som skal brukes til å definere ordningen i indeksen. En B-tre indeks er ordnet etter > eller <, mens vår indeks ikke vil ha noen slik ordning.

amcanunique

Dette er en boolsk verdi som angir om aksessmetoden støtter unike indekser. En unik indeks tillater kun ett tuppel med samme verdi i indeksen. Det er ikke naturlig å gjøre en slik begrensning på fritekstfelt, så en slik egenskap synes unaturlig for oss å støtte. Vi er dessuten avhengig av støtte for dette i den eksterne indeksen, noe det ikke er i FDS.

amcanmulticol

Denne boolske verdien angir om aksessmetoden kan indeksere flere kolonner. En flerkolonnes indeks er primært mulig for to formål: Det ene er definering av primær- eller unike nøkler som består av flere kolonner/attributter. Det vil ikke være naturlig å definere noen slike nøkler på fritekstkolonner av de grunner som er beskrevet over. Det andre formålet er ved søk som inkluderer flere kolonner. Dette kan være aktuelt i en fritekstsøk-sammenheng, men antagelig er dette problemet bedre løst ved hjelp av separate indekser på alle kolonnene. Måten vi oppretter indeksen på i FDS gjør at vi ikke vil få noen bedre ytelse ved å lage en flerkolonnes indeks. Vi vil simpelthen la søkemotoren gjøre søk over flere indekser, ikke bare begrenset til en.

Det er imidlertid grunner for at man bør støtte flerkolonnesindeks allikevel. For det første bør man støtte den rent syntaksmessige oppretting av en flerkolonnes indeks, istedenfor å måtte opprette en og en indeks. I tillegg vil spørreplanleggeren i PostgreSQL utføre søket forskjellig ettersom den har tilgjengelig en flerkolonnes indeks eller ikke. I tilfellet en flerkolonnes indeks antar databasen at den vil kunne få svaret direkte av indeksen ved å søke én gang. Hvis det kun eksisterer indekser på enkeltkolonnene, vil den gjøre et søk mot hver av dem. Spørsmålet her blir altså

om man vil at den eksterne indeksen skal beregne snittet/unionen (AND/OR) av delresultatene eller ikke. La oss f.eks. si at man har indeksert to kolonner og gjør en spørring som innebærer en begrensning på begge kolonnene. Fra kolonne A har vi 1000 tupler som oppfyller betingelsen og fra kolonne B har vi 2000. Videre er det kun 20 dokumenter som tilfredsstiller begge kriteriene. Hvis vi kan implementere flerkolonnes indekser, vil vi kunne la den eksterne indeksen søke på begge kriteriene på en gang, og dermed returnere kun de 20 som oppfyller begge kravene. Hvis ikke, vil vi gjøre to indekssøk som returnerer henholdsvis 1000 og 2000 tupler, og så la databasen utføre beregningen av snittet/unionen på disse to resultatsettene. Databasen vil nok i utgangspunktet kunne gjøre dette vel så raskt som den eksterne databasen, dette er en typisk databaseoperasjon som det ligger mange års forskning bak å optimalisere. Den eksterne indeksen kan derimot spre denne operasjonen over flere noder, noe som vil gjøre at dette går raskere hvis resultatsettet er stort, og vi slipper også å overføre så mye av dataene tilbake til databasen. Det vil altså være å foretrekke å gjøre dette i den eksterne indeksen.

Vi valgte i første omgang å implementere kun enkolonnes indekser. En nærmere diskusjon av dette og illustrasjon av poenget i forrige avsnitt for enkolonners indeks følger mot slutten av avsnitt 6.5.4. Flerkolonners indekser vil jeg ikke diskutere noe nærmere i denne oppgaven, men det vil imidlertid være interessant for en utvidelse, samt teste ytelsesforskjellen mellom de to overnevnte løsningene.

amindexnulls

Denne boolske verdien angir om man kan indeksere NULL-verdier. Det vil ikke være interessant for oss å indeksere dette siden vi kun ser på enkolonnes indekser. I en mulisegmentsindeks vil det derimot være interessant.

amconcurrent

Dette attributtet angir om aksessmetoden støtter samtidige oppdateringer. FDS støtter selvfølgelig dette, og siden dokumentprosessering tar lang tid, er det viktig å tillate samtidige oppdateringer.

Videre følger ti attributter som alle er pekere til pg_proc-tabellen, det vil si at det er pekere til dynamisk lastede funksjoner. Dette utgjør grensesnittet til funksjoner som implementerer en indeksaksessmetode. For å lage en ny indeksaksessmetode må disse implementeres og kalles av systemet når indeksaksessmetoden brukes.

ambuild

Denne funksjonen kalles når en ny indeks skal opprettes, det vil si når brukeren gjør CREATE INDEX. Funksjonen skal klargjøre og opprette de nødvendige datastrukturer for å kunne lagre en indeksrelasjon, samt indeksere eventuelle eksisterende tupler.

aminsert

Denne kalles når et tuppel skal settes inn i indeksen, det vil si under en INSERT-kommando til en tabell på det attributtet som er indeksert. Den kalles også når man gjør en UPDATE-kommando med den nye verdien til attributtet. Den gamle verdien merkes da som slettet. Funksjonen skal sette et tuppel inn i indeksrelasjonen.

ambulkdelete

Denne kalles når et sett av tupler som er market som slettet skal fjernes fra indeksrelasjonen. Denne funksjonen kalles under en VACUUM-operasjon i databasen, som har til hensikt å frigjøre plass. Tuplene som slettes, må først ha blitt merket ved hjelp av for eksempel en UPDATE- eller DELETE-kommando.

ambeginscan

Denne kalles når systemet skal starte et søk i en indeks, og skal sette opp de nødvendige datastrukturer for å gjøre et søk i indeksen. Dette skjer når brukeren gjør SELECT fra tabellen, og det involverer en WHERE-del på det indekserte attributtet.

amrescan

Denne starter et søk i en indeks. Den blir kalt rett etter *ambeginscan* og skal begynne selve søket i indeksrelasjonen på nytt.

amgettuple

Denne funksjonen skal returnere et tuppel fra et søk, og kalles flere ganger etter *amrescan* inntil det ikke er flere tupler å returnere fra søket.

amendscan

Denne avslutter et søk og rydder opp i de allokerte datatrukturene. Denne kalles etter siste kall til *amgettuple*.

amcostestimate

Denne funksjonen skal estimere kostnaden ved bruk av indeksaksessmetoden på en bestemt indeksrelasjon. Det er avgjørende for databasens effektivitet at den velger riktig aksessmetode for å evaluere spørringer. Kostnaden beregnes hovedsakelig på bakgrunn av hvor mange diskaksesser og I/O-operasjoner som må gjøres.

De to siste funksjonene i grensesnittet, *ammarkpos* og *amrestpos* skal lagre og hente opp posisjonen for et søk i indeksen. Disse funksjonene er imidlertid uinteressante fordi de kun brukes hvis indeksen benyttes til å utføre en mergejoin. Generelt er det ikke hensiktsmessig å bruke fulltekstdata til å joine tabeller med hverandre, og vi kan derfor anta at dette ikke er et interessant bruksområde for indeksen vår[1]. Disse metodene er derfor bare implementert slik at de kaster en feilmelding.

For at spørsmålsoptimereren skal vurdere bruk av vår aksessmetode under en spørring, må vi knytte aksessmetoden til en operator slik som forklart under pg_opclass. For å kunne bruke syntaksen WHERE text CONTAINS 'searchexpression' må vi først legge til operatoren CONTAINS. Deretter må vi lage en ny standard-operatorklasse for denne datatypen og aksessmetoden, altså for text/fastindex. Dette må gjøres ved å sette inn et tuppel i pg_opclass. Deretter knytter vi vår nye operator CONTAINS til denne operatorklassen ved å sette inn et tuppell i pg_amop.

[1] Grunnen til at vi kan anta at dette er uinteressant er i hovedsak på grunn av hensikten med å utføre en join-operasjon. Dette er som regel for å sette sammen normaliserte tabeller. Det vil være uhensiktmessig å velge f.eks. et fritekstatributt som fremmednøkkel, til dette brukes som oftest korte strenger eller tall.

6.4. Grensesnitt mot FDS

FDS organiserer indeksen i *collections*, som er logiske samlinger av indekserte dokumenter. Vi hadde valget mellom enten å lage en ny collection for hver ny databaseindeks eller å putte alle indeksene i en collection. Vi valgte det siste, slik at man lettere kan søke over flere indekser, og fordi grensesnittet til FDS ikke tilbyr noen god måte å manipulere collections på.

6.4.1. Indeksprofil

For å tilpasse FDS-indeksen til den informasjonen vi ville lagre og ta ut ved søk, definerte vi en indeksprofil. En indeksprofil er en konfigurasjonsfil som forteller hvordan det fysiske utseende på indeksen skal være, og FDS bruker denne filen til å generere andre og mer kompliserte konfigurasjonsfiler. Indeksprofilen inneholder informasjon om hvilke felter indeksen skal lagre, hvilke som skal indekseres og dermed kunne søkes på, og hvilke felter som skal være med i resultatet. Videre kan man blant annet gjøre valg om feltene skal lemmatiseres og sorterer.

Indeksprofilene definerer først en liste med felter som skal lagres i indeksen, og så eventuelle sammensatte felt. Vi hadde behov for følgende felter:

data

> Dette er teksten som er indeksert og som det skal søkes i. Vi spesifiserer at feltet skal lemmatiseres og at det kan sorteres hvis ønskelig. Dynamisk resultat gir oss mulighet til å hente ut et sammendrag av dokumentet, til forskjell fra et statisk resultat som ville gitt oss hele innholdet. Dette er imidlertid lagret i databasen, så vi trenger ikke å hente ut dette fra indeksen.

block/offset

> Dette er en peker til hvor data ligger på disk i databasen. PostgreSQL trenger denne informasjonen for å finne tilbake til tuppelet. Disse feltene er heltall, og siden det ikke er noen hensikt i å søke i dem, lar vi dem være uindeksert.

indexOid, database, host.

> Denne informasjonen identifiserer til sammen én indeks i én database på én spesifikk maskin. På denne måten kan indekser fra flere forskjellige databaser og maskiner indekseres på et sted. Ved søk innenfor en indeks må disse verdiene oppgis, og vi har derfor angitt at feltene må indekseres.

rowOid

> Dette er en tuppelidentifikator som kan identifisere et tuppell innenfor en database. Vi bruker dette feltet for å finne frem til det indekserte tuppelet i databasen i contains_table-funksjonen (6.6). Dette er et heltall og trenger ikke indekseres, fordi det ikke vil være nødvendig å søke i det.

Det sammensatte feltet, *content,* består kun av et felt, nemlig data. Det er dette feltet vi søker i. Hvis vi hadde hatt flere felter vi ville søke over samtidig, kunne vi satt opp flere feltreferanser her.

Alle dokumenter i en collection må ha en unik identifikator. Denne er bygget opp av følgende felter i indeksen:

```
Document-id: host/database/indexoid/block/offset
```

Vår indeksprofil ble seende slik ut:

```
<?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="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>
```

6.4.2. Implementasjonen mot FDS

FDS har to grensesnitt for å aksesserere indeksen:

- Innholdsgrensesnitt
- Søkegrensesnitt

Innholdsgrensesnittet gir mulighet for å endre innholdet i søkemotoren, det vil si å legge til, endre og slette dokumenter. Spørregrensesnittet gir tilgang til å hente ut informasjon fra søkemotoren. Alle grensesnittene er tilgjenglig for Java, C++ eller gjennom COM. Siden funksjonene som skal lastes inn i PostgreSQL må være skrevet i C, eller et språk som er kompatibelt med C, lagde vi et sett kommunikasjonsfunksjoner i C++. Disse linket vi sammen med grensesnittsfunksjonene i en delt objektfil som vi lastet inn i PostgreSQL. Deretter kalte vi C++ funksjonene fra C, men holdt den FDS-spesifikke implementasjonen i C++, slik at denne lett kunne byttes ut med en annen type ekstern indeks og slik at skillet skulle bli klarere.

Kommunikasjonsgrensenittet ble da som følger:

insert_document

Setter inn et dokument i den eksterne indeksen. Vi vil her måtte ta imot dokumentet, det vil si teksten som skal indekseres, i tillegg til dokumentidentifikatoren og de andre feltene som skal settes inn. Funksjonen bygger opp et dokument av denne informasjonen og sender det videre til FDS.

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

    icontent_manager_ptr cm(
```

```
     get_content_manager(HOST, PORT, COLLECTION));
    content_factory_ptr cfact(create_content_factory());
     .
     .
    document_element_ptr data_elem(
     new string_element("data", d));
    doc->add_element(data_elem);
    document_element_ptr db_elem(
     new string_element("database", db));
    doc->add_element(db_elem);
    document_element_ptr db_host_elem(
     new string_element("host", dbhost));
    doc->add_element(db_host_elem);
     .
     .
    cm->add_content(doc.get());
```

do_scan

Denne funksjonen skal gjøre et søk i den eksterne indeksen og returnere resultatet. Resultatet bygges opp som en lenket liste slik av vi kan returnere det til C-koden og behandle den videre der. Søket skjer innenfor en oppgitt indeks, database og maskin.

```
extern "C" FastSearchResult* _do_fastscan(
     char* searchKey, int indexOid, char * database, char *
dataHost)
{
..
    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;
}
```

do_delete

Denne funksjonen skal slette et dokument fra den eksterne indeksen. Den må motta og bygge opp en dokumentidentifikator og slette det angitte dokumentet:

```
extern "C" char * _do_fastdelete(
     int block, int offset, int indexOid,
     char * database, char * dataHost) {
  string docId = getDocumentId(
     block, offset, indexOid, database, dataHost);
    icontent_manager_ptr cm(get_content_manager(
     HOST, PORT, COLLECTION));
    content_factory_ptr cfact(create_content_factory());
    icontent_id_ptr id = cfact->create_id(docId);
```

```
        cm->remove_content(id);
    }
```

6.5. Implementering av aksessmetoden

PostgreSQL består i skrivende stund av ca. 500.000 linjer C-kode. Grensesnittet til indeksaksessmetoder er ikke dokumentert, og vi tok derfor i stor grad utgangspunkt i de aksessmetodene som finnes i PostgreSQL fra før. De aksessmetodene som ligger i databasen, er R-tre for indeksering av multidimensjonale data, B-tre, Hash og Gist, som er et generelt søketre.

Det er imidlertid flere områder der en ekstern indeksaksessmetode skiller seg fra de som allerede finnes i PostgreSQL. Tildels er dette forenklende, men det innfører også en hel rekke nye problemer.

Den mest opplagte forskjellen er at de eksisterende aksessmetodene lagrer selve indeksen fysisk innenfor databasen. En ekstern indeks lagrer omtrent ingen informasjon her, men har hele indeksen lagret utenfor. Forenklingen går stort sett på at vi ikke skriver til disk i kontekst av databasen. Dette gjør at vi slipper å tenke på potensielle samtidighetsproblemer, vi trenger ikke å låse buffere vi skal skrive til og vi trenger ikke å kjenne til hvordan PostgreSQL's buffersystem fungerer.

Problemene med nettopp dette er at PostgreSQL antar at alle indeksaksessmetodene lagrer indeksen i databasesystemet. Grensesnittet for å utvide databasen med en ny aksessmetode tar rett og slett ikke høyde for at indeksen lagres eksternt. Dette fører til at grensesnittet blir enkelt, fordi PostgreSQL gjør mye av jobben når det gjelder lagring og sletting av diskområdene som indeksen ligger på. Men dette fører imidlertid til mangler i grensesnittet for vår del, og det er spesielt et problem når vi skal rydde opp etter oss. Sletter vi en tabell, fjerner buffersystemet automatisk de sidene på disk den vet at indeksen bruker, som for vår del er omtrent ingen. Grensesnittet signaliseres ikke om dette, og vi har dermed i utgangspunktet ingen mulighet for å rydde opp i den eksterne indeksen. For å få til dette må vi antagelig gjøre om på grensesnittet.

6.5.1. Modifisering av systemkataloger

For at PostgreSQL skulle kunne bruke vår aksessmetode, måtte vi som nevnt modifisere flere forskjellige systemkataloger. Noen kunne modifiseres ved hjelp av spesielle SQL-kommandoer, mens andre måtte modifiseres direkte.

Først lastet vi inn grensesnittsfunksjonene for den nye aksessmetoden slik:

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

Dette registrerer funksjonen *fastbuild* i systemkatalogen pg_proc, og på samme måte gjorde vi det med de 9 andre funksjonene.

Deretter la vi inn en ny indeksaksessmetode ved å legge til et 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'),
     …
     …
);
```

Dette tuppelet beskriver alle egenskapene til aksessmetoden og inneholder peker til alle grensesnittsfunksjonene. Man kan nå lage en ny indeks basert på den nye indeksaksesmetoden ved hjelp av kommandoen *CREATE INDEX*.

For at man skal kunne bruke aksessmetoden i forbindelse med en spørring må vi endre ytterligere noen systemkataloger. For det første trenger vi *CONTAINS*-operatoren. På grunn av begrensninger i PostgreSQL kan ikke en operator bestå av vanlige bokstaver. Vi valgte derfor å la tegnene @@ betegne *CONTAINS*. Den definerte operatoren ble slik:

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

Her definerer vi at operatoren @@ både har venstre og høyre argument av typen *text*. Dette gjør den til en binær operator. Videre er operatoren implementert med funksjonen *contains_operator* som er en funksjon lastet inn i PostgreSQL på tilsvarende måte som grensesnittsfunksjonene til aksessmetoden. I vårt tilfelle er denne implementasjonen tom, den kaster kun en feilmelding, fordi vi aldri har til hensikt å evaluere denne operatoren på noen andre måter enn ved hjelp av aksessmetoden. Hvis vi skulle ha implementert denne operatoren, måtte det ha vært ved hjelp av den eksterne indeksen.

For at PostgreSQL skal vurdere bruk av vår indeks for å aksessere attributter av typen *text* når operatoren @@ brukes, må vi definere en operatorklasse, og binde den nye operatoren til denne. Operatorklassen definerer et sett med operatorer som kan evalueres med en aksessmetode for en spesiell datatype. En ny operatorklasse defineres for hver type aksessmetode den skal brukes sammen med, det vil si en binding mellom en aksessmetode og en datatype. Denne definerte vi slik:

```
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);
```

Så knyttet vi operatoren til klassen gjennom å legge til et nytt tuppell i systemkatalogen pg_amop:

```
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'
 ), --fast operator class
 1, --Strategy no 1
 false, --Recheck the hit? No we 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')
 ) - The @@ (CONTAINS) operator for the text type
);
```

Deretter var alt klart for å gjøre *SELECT* og hente resultater ut fra den eksterne indeksen.

Implementasjonen av grensesnittsfunksjonene kan sees på i 4 deler. De funksjonene som blir kalt når vi lager indeksen, når vi setter tupler inn i indeksen, når vi spør fra indeksen, og når vi sletter fra den. I litt mer detalj gjorde vi følgende under hver av disse delene. Figur 6-7 gir en oversikt over gangen i funksjonskallene ved hjelp av et sekvensdiagram.

6.5.2. Bygging av indeks

PostgreSQL kaller grensesnittsfunksjonen *ambuild* når den skal lage en ny indeks, altså når brukeren gjør CREATE INDEX/TABLE. Databasen har allerede laget indeksrelasjonen, og det som gjenstår, er å passe på at alle tuplene som allerede er i tabellen blir indeksert. Dette gjøres ved hjelp av en grensesnittsfunksjon som kaller tilbake til vår *aminsert* funksjon for alle dokumenter som ikke er indeksert. Innsettingen er forklart nærmere under.

6.5.3. Innsetting

Når ett tuppel settes inn i en indeksert tabell, kalles funksjonen *aminsert* for den korrekte aksessmetoden. Her henter vi opp verdien som skal indekseres, og sender den videre til kommunikasjonsfunksjonen. For å få aksessmetoden til å fungere med flere datatyper burde vi her brukt støttefunksjoner for å hente opp den korrekte verdien, slik som nevnt i 6.3. Nå antar vi at den er av typen *text*.

I kommunikasjonsfunksjonen lager vi et dokument som kan settes inn i FDS. Dette dokumentet skal ha en unik ID, slik at vi unikt identifiserer et dokument. I dokumentet setter vi inn teksten som skal indekseres og en ItemPointer, som er en peker til hvor på disk dette tuppelet ligger. Denne er bygd opp av et blokknummer og et offset. PostgreSQL vet selv hvilken fil dette blokk/offsetnummeret relaterer seg til innenfor en indeks. På denne måten kan vi finne tilbake til tuppelet når vi henter ut dokumentet fra indeksen.

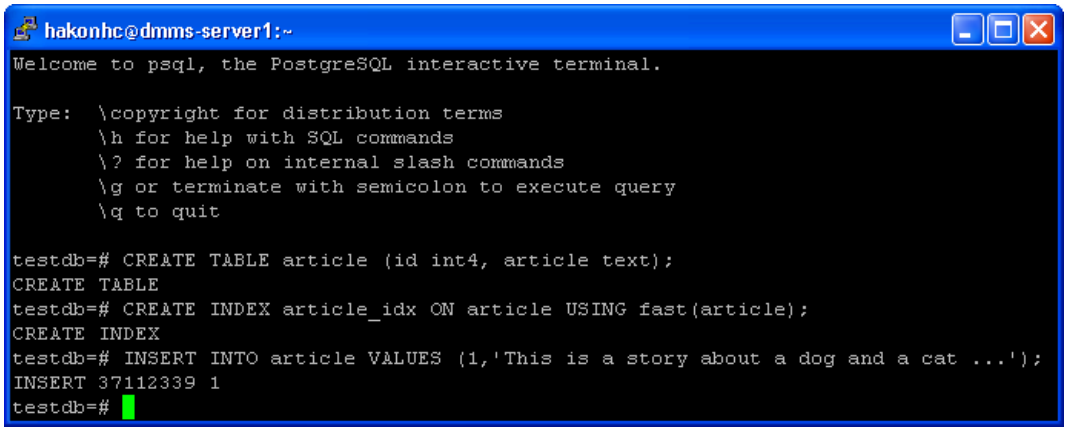
Ved en UPDATE-spørring merkes først det originale tuppelet som ugyldig, og deretter settes det nye tuppelet inn på samme måte som over. Det gamle tuppelet som er merket for sletting, vil først bli fjernet under en VACUUM-operasjon, se avsnitt 6.5.5.

Erfaringer fra implementasjon av innsetting

Flere ting kan sies om prosessen med å sette inn dokumenter. I tillegg til problematikken som allerede er nevnt med å indeksere flere datatyper, bør antagelig innsettingen foregå asynkront. Innsetting i den eksterne indeksen kan ta tid, spesielt i de tilfeller der indeksen

og databasen ikke ligger på samme maskin. Uansett foregår innsettingen via nettverkslagene, og her er det stort overhead i forhold til vanlige PostgreSQL-indekser som lagrer data direkte innenfor databasen. Responstiden på å sette inn et dokument, spesielt når man skal indeksere store mengder data, vil bli uakseptabelt lang hvis det ikke foregår asynkront. På en annen side vil det bli vanskeligere å håndtere feil og å avbryte innsettingen i tabellen hvis noe har gått galt under prosessen med å sette inn dokumentet i den eksterne indeksen. Vi vil også kunne få et problem hvis vi skal lage en ny prosess for hvert dokument som skal settes inn, fordi dette kan være et ganske stort antall hvis man indekserer en tabell med mye data. Alternativt kan vi ha en egen prosess som kommuniserer med den eksterne indeksen og legge dokumenter som skal settes inn i en kø, for så å sette inn én av gangen.

Uansett om dokumentene settes inn asynkront eller ikke, vil den eksterne indeksen bruke tid på å indeksere dokumentet etter at det er satt inn. Dette betyr at dokumenter som blir satt inn, ikke umiddelbart blir gjort tilgjengelig for søk, og derfor kan data vi har i databasen i slike tilfeller unnlate å komme med i resultatet fra et søk. Systemet blir derfor ikke strengt korrekt med hensyn til oppdateringer, men i fritekstsøk-sammenheng vil dette ofte være mindre viktig i forhold til responstid. Fritekstsøkteknologi i dag baserer seg ofte på langt mindre krav i forhold til oppdatering enn dette, ved for eksempel å bruke crawlere og pushing av data ved regelmessige tidsintervaller.



Figur 6-3: Dette bildet viser psql, et klientprogram mot PostgreSQL for å kjøre SQL kommandoer. Her lages en tabell, en FDS indeks på en av tabellens atributter, og dernest settes et tuppel inn i tabellen som da indekseres med den nye indeksen.

6.5.4. Søking

Når PostgreSQL mottar en spørring av denne typen:

```
SELECT * FROM tabell
WHERE attribute CONTAINS 'search exporession'
```

vil den forsøke å optimalisere WHERE-delen ved bruk av en indeksaksessmetode i stedet for den vanlige heap-aksessmetoden (sekvensielt søk). Det som er spesielt for vår aksessmetode, er at operatoren CONTAINS (@@) ikke kan evalueres annet enn via den eksterne indeksen. Dette gjør at vi må sikre oss at spørsmål av denne typen aldri blir forsøkt evaluert ved hjelp av et sekvensielt søk. I tilfelle med spørringen over, vil databasen se at operatoren CONTAINS er knyttet til operatorklassen text/fastindex, slik at aksessmetoden fastindex vil være en mulig aksessmetode for denne spørringen.

Databasen vil deretter vurdere flere forskjellige mulige planer for eksekvering av spørringen, der en av dem vil være aksessmetoden vår og en vil være et sekvensielt søk.

For å kunne velge, baserer spørsmålsoptimereren seg på en del statistikk, slik som hvor mange tuppler/blokker det er i tabellen, hvor mange som potensielt kommer til å bli plukket ut av operatoren og så videre. Kostnadene ligger stor sett i hvor mye disk I/O som hver plan medfører. For å finne ut av hvor mye det koster å kjøre vår indeksaksessmetode, vil databasen kalle grensesnittsfunksjonen *amcostestimate.* Denne funksjonen mottar en liste med WHERE-klausuler som er evaluerbare med denne aksessmetoden og må returnere et estimat på hvor mye det koster å aksessere indeksen og hvor stor selektiviteten til WHERE-klausulen er. Selektiviteten vil si et estimat på hvor mange tupler som kommer til å bli returnert i forhold til det totale antall tupler. For å sikre at vår aksessmetode blir brukt, returnerer vi bare null i kost, slik at den alltid vil finne ut at indeksen er raskest å bruke.

```
Datum fastcostestimate(PG_FUNCTION_ARGS)
{

  Cost *indexStartupCost = (Cost *) PG_GETARG_POINTER(4);
  Cost *indexTotalCost = (Cost *) PG_GETARG_POINTER(5);
  Selectivity *indexSelectivity =
     (Selectivity *) PG_GETARG_POINTER(6);
  Double *indexCorrelation = (double *) PG_GETARG_POINTER(7);

  *indexStartupCost = 0;
  *indexTotalCost = 0;
  *indexSelectivity = 0;
  *indexCorrelation = 1.0;

  PG_RETURN_VOID();
}
```

Når så indeksaksessmetoden har blitt valgt, kalles funksjonen *ambeginscan* for den korrekte aksessmetoden. Denne setter opp en del variabler for søket, og kaller deretter *amrescan* som gjør selve søket. Her allokerer vi en datastruktur for å legge resultatet fra søket inn, finner frem søkenøkkelen og sender dette videre til kommunikasjonsfunksjonen, som gjør selvet søket opp mot FDS. Dette sender vi sammen med parametere om hvilken tjener, database og indeks vi skal gjøre søket innenfor. Tilbake fra kommunikasjonsfunksjonen får vi en resultatstruktur som består av ItemPointere, altså pekere til disk for de aktuelle tuplene vi fant. Denne returneres til PostgreSQL.

Deretter kaller databasen *amgettuple* helt til denne funksjonen returnerer usann. For hver gang returnerer vi neste ItemPointer i resultatstrukturen vår. Når det ikke er flere tupler å hente kaller systemet *amendscan* der vi rydder opp og frigjør resultatstrukturen. Figur 6-7 viser gangen i kallene.

Erfaringer fra implementasjonen av søk

For det første bør vi, slik som under innsetting, ta høyde for at det kan komme andre datatyper enn text som søkeargument. Vi kan også her bruke støttefunksjonene for å gjøre dette generisk.

I vår implementasjon kan man oppgi et boolsk uttrykk som en streng. Dette kan støttes på flere måter og her kommer vi inn på en mer omfattende diskusjon om hvordan et

fritekstsøkespråk bør være. Vi oppdaget imidlertid en del ting om hvordan PostgreSQL bygger opp og eksekverer WHERE-klausulen som vil være viktig uansett hvilken måte man velger å gjøre det på senere.

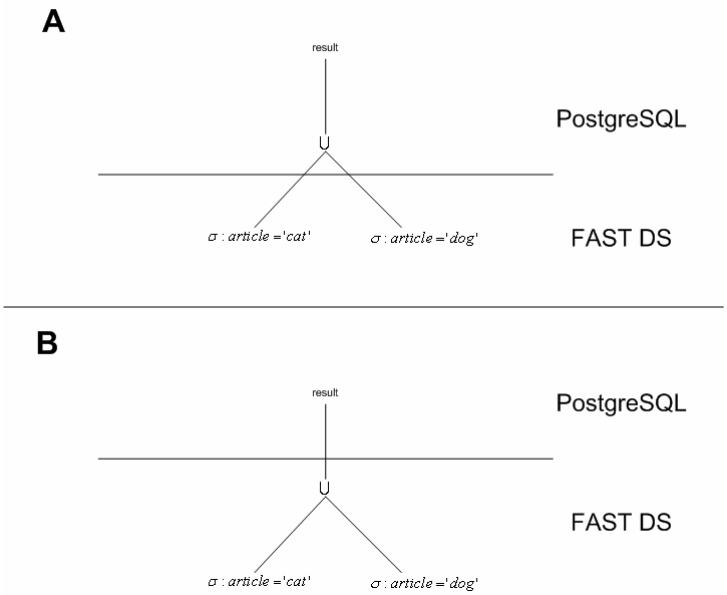
Søkeuttrykket sendes som et boolsk uttrykk i WHERE-delen i spørringen slik:

```
SELECT * FROM articles WHERE article CONTAINS 'cat AND dog';
```

Søkeuttrykket kan FDS bruke uten modifikasjon ved at vi benytter deres eget spørrespråk i søkeuttrykket. Denne løsningen ligger nærmest SQL/MM og andre databaser. En annen mulighet er å kunne spesifisere søket i mer naturlig SQL92/99, av denne typen:

```
SELECT * FROM articles
WHERE article CONTAINS 'cat' AND article CONTAINS  'dog';
```

Hvis vi skal gjøre det på denne måten, kommer vi inn på en del problematikk i forbindelse med hvordan spørsmålsbyggeren og parseren til PostgreSQL jobber. I noen tilfeller (det gjelder også i tilfellet over) vil den sende med to søkenøkler til *amrescan* slik at vi kan bygge opp søket med begge to. Hvis vi derimot hadde brukt OR og ikke AND, ville PostgreSQL valgt å gjøre to kall til *amrescan* med de to nøklene hver for seg, for så å gjøre en union av resultatene etterpå. Spørsmålet er om det er mer effektivt å sende flere søk og la databasen sette sammen resultatet etterpå ved hjelp av mengde funksjoner (snitt, union, differanse), eller om FDS gjør denne jobben like raskt. Figur 6-4 illustrerer denne forskjellen. Databaser er optimalisert for nettopp denne typen operasjoner, men det er flere grunner til at denne operasjonen helst bør gjøres i indeksen. For det første er indeksen skalerbar og kan benytte seg av flere noder til å utføre disse operasjonene. For det andre kan mellomresultatet fra de forskjellige operasjonene bli store (for eksempel ved en AND), mens sluttresultatet er lite. Det lønner seg derfor å gjøre mengdeoperasjonen før man sender over resultatet for å spare på kostbar kommunikasjon mellom de to systemene. Skal vi gjøre noe effektivt her, må vi sannsynligvis endre på mer grunnleggende ting i PostgreSQL, og det har vi ikke gjort i første omgang.



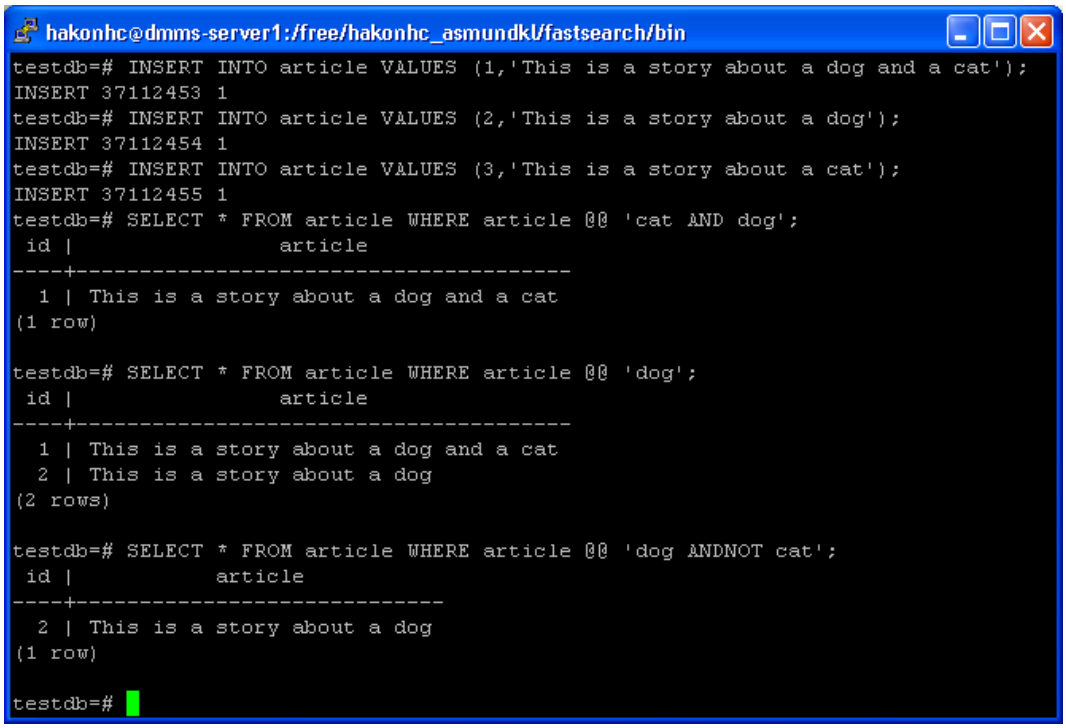
Figur 6-4: Figuren illustrerer hvor mengdeoperasjonene gjøres. I A gjøres unionen mellom to projeksjoner (relasjonsalgebra) i databasen, mens i B gjøres dette i FAST DS.

Ved å bruke en søkefunksjon eller en operator samt spesifisere hele søkeuttrykket i en streng slik som foreslått over, vil vi kunne gjøre alle operasjonene i den eksterne indeksen, uten at vi må gjøre endringene i grensesnittet til PostgreSQL. Siden dette ligger nærmest SQL/MM og andre databaser, synes det som dette er den foretrukne løsningen. Noen av de samme problemstillingene får man ved bruk av flerkolonnes indekser, det vil jeg imidlertid ikke diskutere videre.

Et alternativ til å bruke en CONTAINS-operator er å definere en CONTAINS-funksjon som kan brukes i WHERE-delen av spørringen. Denne funksjonen kan da returnere sann eller usann ettersom en oppgitt spørring er inneholdt i en fritekst eller ikke. På denne måten brukes ikke aksessmetodegrensesnittet. En slik CONTAINS-funksjon vil kunne brukes på følgende måte:

```
SELECT * FROM article WHERE Contains(articles, 'dog AND cat');
```

Funksjonen returnere her sann eller usann ettersom et tuppel fra attributtet articles innholder 'dog AND cat' eller ikke. Ulempen med denne måten å gjøre det på er vanskeligheten med å gjøre kall til den eksterne indeksen kun en gang, siden funksjonen vil bli kalt for hvert tuppel som skal evalueres. Denne måten blir derfor enten ineffektiv eller vanskelig å implementere. I PostgreSQL er det derfor bedre å bruke en operator.



Figur 6-5: Dette kjøreeksempelet viser noen innsetting og søkeoperasjoner der CONTAINS operatoren (@@) og noen enkle boolske søkeuttrykk brukes.

Et alternativ til å bruke en CONTAINS-funksjon i WHERE-delen er å bruke den i FROM-delen av spørringen. På denne måten blir resultatet fra funksjon en tabell som returnerer flere tupler:

```
SELECT * FROM CONTAINS_TABLE(article, articles, 'dog AND cat');
```

Det vil si på formen:

```
SELECT * FROM CONTAINS_TABLE(table, attribute, 'search
expression');
```

Her vil tabellfunksjonen bli kalt bare én gang, slik at vi har mulighet til å gjøre spørringen mot den eksterne indeksen en gang og samtidig returnere flere svar, på samme måten som vi kunne med grensesnittsfunksjonene til en aksessmetode. Fordelen med dette er imidlertid at vi i mye større grad har kontroll over hva som blir kalt og hva vi skal returnere.

Denne måten å spørre på gir oss dessuten muligheten til å kunne returnere mer enn selve tuppelet. I fritekstsøksammenhenger er det ofte mer interessant med et generert sammendrag av den teksten vi søker etter enn hele teksten, slik vi ser det når vi søker i en søkemotor på Internett. En annen ting som er vanlig, er å rangere dokumentene i forhold til det oppgitte søkeuttrykket. Begge disse tingene støttes av den eksterne indeksen, men kan ikke presenteres når vi gjør søk via aksessmetoden. Med en tabellfunksjon kan vi derimot gjøre noe slikt:

```
SELECT rank, summary
FROM CONTAINS_TABLE(article, aricles, 'dog') ORDER BY rank;
```

Vi kan også returnere en primærnøkkel til selve dokumentet slik at vi kan bruke dette i forbindelse med en join, slik som dette:

```
SELECT r.rank, r.summary, a.author
FROM CONTAINS_TABLE(article, articles, 'dog'
     as t(rank, summary, documentOid) r, article a
WHERE a.oid = r.documentOid ORDER BY rank;
```

Alternativt kan vi hente opp hele tuppelet og modifisere det ved å legge til de atributtene vi ønsker.

Når det gjelder ytelse er det særlig *en* forskjell mellom søkemotoren og databasen som har betydning. En søkemotor er svært optimalisert for å hente ut de høyest rangerte resulatene først, det vil si spørringer som sorteres på rangering, altså ORDER BY rank i SQL. Søkemotorer er også optimalisert for ”lat” evaluering av søket. Dette vil si at den faktisk ikke utfører hele søket slik som den gjør når vi skal ha ut hele resulatet, hvis vi kun vil ha f.eks. de 10 beste. I en database derimot er alle radene i resulatet like vikitge, tradisjonelt sett. Vår implementasjon drar derfor ikke nytte av denne type optimalisering. Vi kunne derimot tenkt oss at spørringer på denne formen kunne vært optimalisert ved hjelp av ”lat” evaluering:

```
SELECT rank, summary
FROM CONTAINS_TABLE('articles', 'article', 'dog AND cat')
ORDER BY RANK LIMIT 10;
```

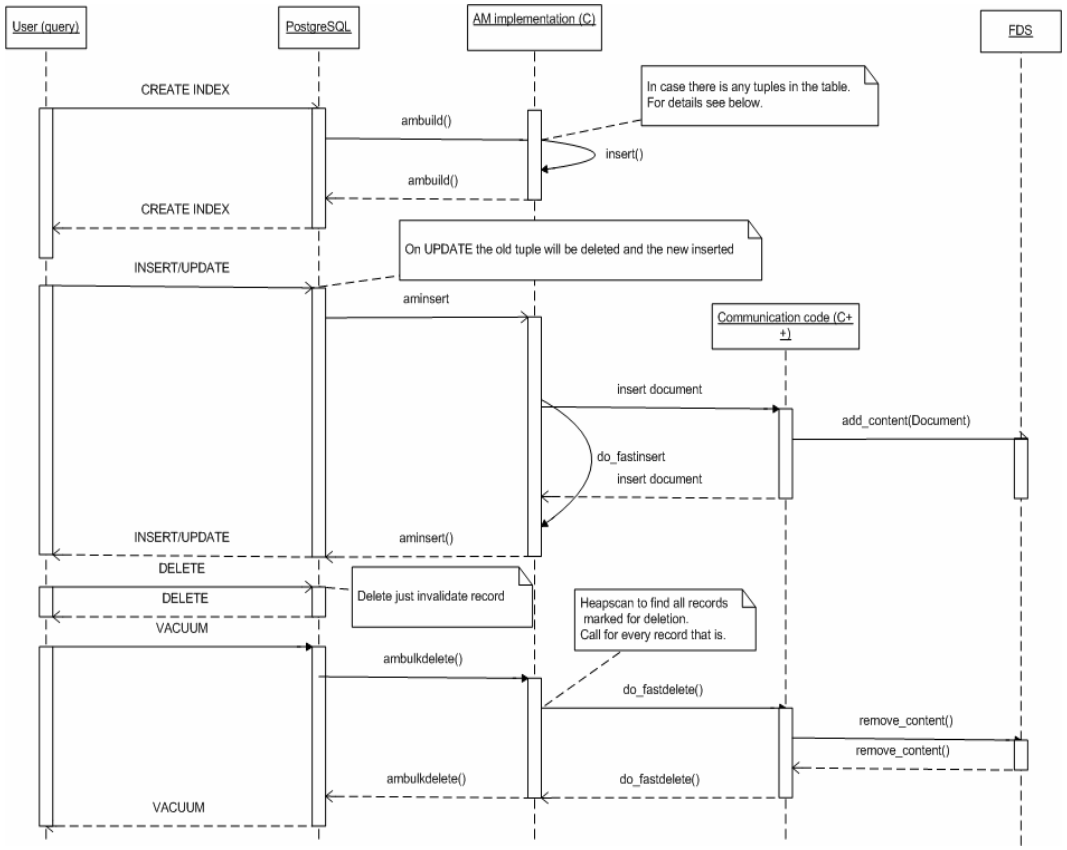
Denne spørringen er SQL måten å hente ut de 10 beste resultatene på. PostgreSQL gjør derimot, naturlig nok, ingen forsøk på å gjøre en ”lat” spørring, men begrenser resulatet etter at søket er utført. Hvis vi skal kunne optimalisere dette må vi igjen inn spørreoptimereren til PostgreSQL for at aksessmetodegrensesnittet kan bli informert om at vi ønsker å gjøre et begrenset søk. En annen grunn til å gjøre slike ”begrensede” spørringer er i tilfeller der resulatet er stort. Ved å minimere kommunikasjonen mellom den eksterne indeksen og databasen gjennom å begrense søket, vil søket bli ytteligere effektivisert. Hvis man uansett ikke er interessert i alle resultatene, er dette å foretrekke.

6.5.5. Sletting

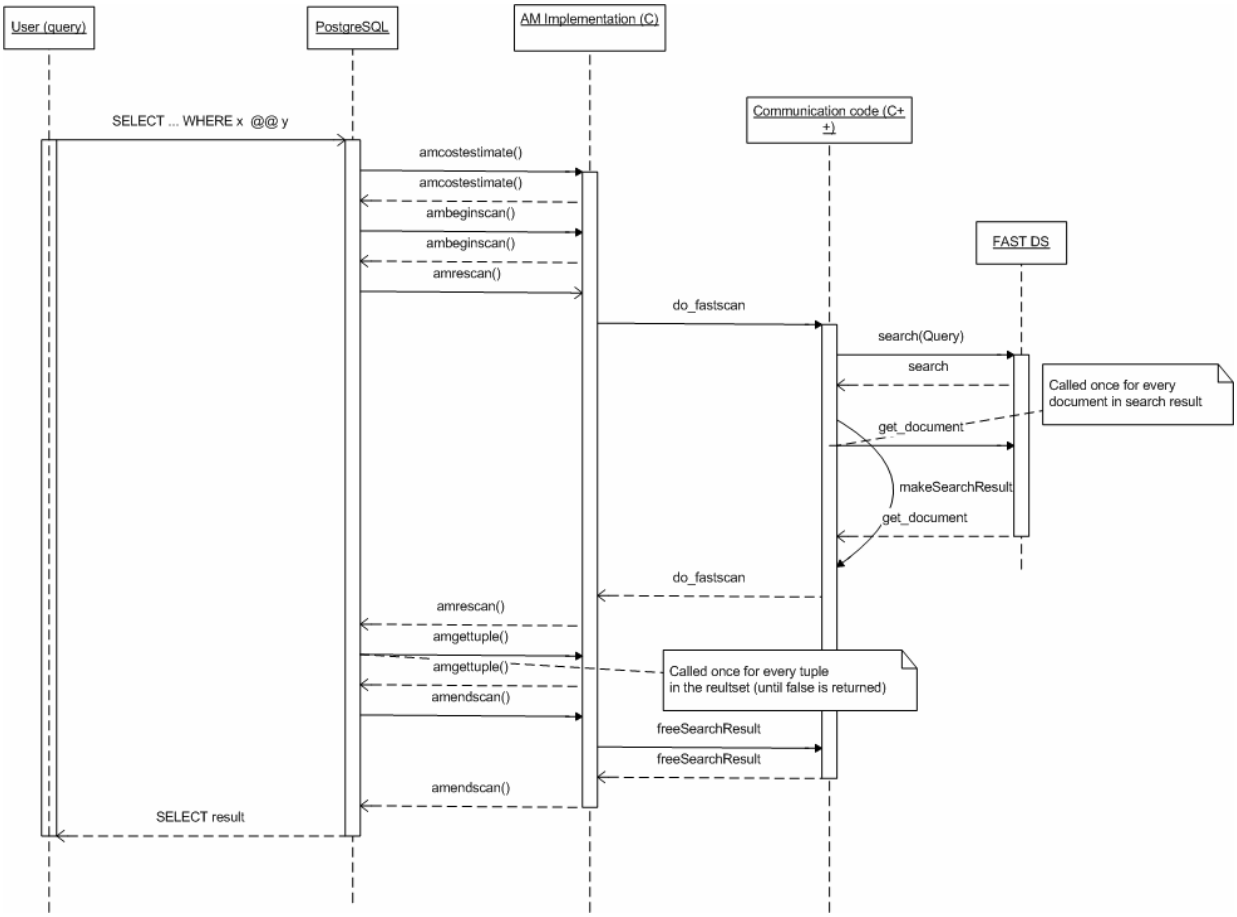
I PostgreSQL vil normalt ikke tupler bli slettet fysisk når man sletter dem med en DELETE-kommando. De blir markert som slettet, men blir ikke fjernet før VACUUM blir gjort. Dette gjelder vanlige tabeller, så vel som indeksrelasjoner. VACUUM er en prosess som frigjør ubrukt diskplass, samt samler statistikk om tabeller og indekser. Dette medfører at ingen grensesnittfunksjoner blir kalt når man utfører en DELETE, men siden databasen selv vet hvilke tupler som er slettet, vil søk som resulterer i at indeksen returnerer pekere til tupler som er slettet, ikke bli en del av resultatet som presenteres til brukeren. Først når man kjører kommandoen VACUUM, kalles grensesnittsfunksjonen *ambulkdelete* hvis det er indekserte tupler som har blitt slettet. I vår implementasjon av denne funksjonen går vi gjennom relasjonen vi indekserer med en 'heap scan', det vil si et sekvensielt søk, og plukker ut de tuplene som er merket for sletting, og som derfor skal slettes fra den eksterne indeksen. Deretter kaller vi vår kommunikasjonsfunksjon som bygger opp den unike identifikasjonen for dette tuppelet, og sletter det aktuelle dokumentet i den eksterne indeksen.

```
/* 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;

       /* delete the data from the page */
        _do_fastdelete(ItemPointerGetBlockNumber(&heaptup),
    ItemPointerGetOffsetNumber(&heaptup), RelationGetRelid(indexRel),
            DatabaseName, host);
      }
 }
```

Det finnes imidlertid andre tilfeller hvor man ønsker å slette fra den eksterne indeksen, men grensesnittet mot aksessmetodene ikke blir kalt. Dette skjer blant annet når man sletter indeksen, tabellen, eller selve databasen. I alle tilfeller antar PostgreSQL at indeksen ligger fysisk på disk innenfor databasen. Alle disse tilfellene skal medføre at indeksen blir slettet i sin helhet, og databasen sletter derfor de sidene den vet at indeksen ligger på. Selv om dette ikke medfører feil i forhold til søk, betyr dette et stort problem i forhold til oppryddingen og derfor størrelsen på den eksterne indeksen. Det kan også bety at søk vil gå tregere, siden det ligger mye unødvendig data i indeksen. For å løse dette må vi antagelig utvide grensesnittet mot aksessmetodene. Alternativt må man sørge for å slette alle tuplene og gjøre VACUUM før man sletter en hel indeks.



Figur 6-6: Dette UML sekvensdiagrammet [34] viser en oversikt over hvilke funksjonskall som skjer når brukeren gjør CREATE INDEX, INSERT og DELETE/VACUUM. Til venstre vises brukeren som gir kommandoer til PostgreSQL. Denne gjør så kall til vår aksessmetodeimplementasjon. Her kaller vi så vår kommunikasjonskode som igjen gjør kall mot FDS.



Figur 6-7: Dette UML sekvensdiagrammet viser det samme som forrige figur, men for kommandoen SELECT.

6.6. Implementasjon av tabellfunksjoner

Etter å ha implementert aksessmetoden, gikk vi videre med implementasjonen ut i fra en del av de erfaringene og behovene vi hadde sett under den første implementeringsfasen. I første omgang gikk vi inn for å implementere funksjonalitet som kunne utvides innenfor de rammene som PostgreSQL er designet for. PostgreSQL lar en skrive egne funksjoner, eller såkalte 'stored procedures' (SQL/92), som kan implementeres i C og lastes inn på samme måte som vi gjorde i den forrige delen ved å sette inn et tuppel i systemkatalogen pg_proc. Funksjonene kan returnere en atomisk datatype, og kan da brukes i SELECT eller WHERE, samt andre deler av spørringen der det forventes en atomisk verdi. Alternativt kan en funksjon returnere tupler og brukes som en "tabell" i FROM-delen av spørringen. Som tidligere nevnt vil dette oppfylle flere av de kravene vi har satt opp:

- At funksjonen blir kalt én gang, slik at vi kan gjøre en spørring mot den eksterne indeksen kun én gang.
- At andre eller flere atributter enn selve tuppelet kan returneres. Vi ønsker å returnere felter som den eksterne indeksen gir oss, og som det er naturlig å bruke i en fritekstsøksammenheng, nemlig rangering og sammendrag av dokumentet.
- At spørringene syntaksmessig er mest mulig lik gjeldende standarder.

Vi implementerte en contains_table-funksjon som returnerer rangering og sammendrag for de dokumentene spørringen treffer. For å integrere den i PostgreSQL gjorde vi følgende:

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

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

Man kan deretter gjøre spørringer på denne formen:

```
SELECT *
FROM CONTAINS_TABLE(table, attribute, 'search expression');
```

Denne spørringen finner frem til riktig indeks på grunnlag av *table* og *attribute*-parameterene og kaller vår kommunikasjonsfunksjon mot den eksterne indeksen for å gjøre søket. Deretter bygger den opp tupler definert av en kompleks type contains_table_result, som den returnerer. Den gir også rangering og sammendrag, og i tillegg en identifikator til det opprinnelige tuppelet, slik at tabellen lett kan joines inn igjen som vist under avsnitt 6.5.4.

6.7. Testing av implementasjonen

For å avdekke svakheter i implementasjonen, samt undersøke dens effektivitet gjorde vi noen tester opp mot eksisterende løsninger for å gjøre fulltekstsøk i PostgreSQL.

6.7.1. Testoppsett

Testene ble utført på en maskin med følgende spesifikasjon.

Tabell 6-1: Maskinvare

CPU:	2 x Pentium Xeon 2.2Ghz.
Minne:	5.6 GB
Disk:	UltraWide2 SCSI, RAID
OS:	RedHat Linux 8.0, Linux 2.4.18-14bigmem
Søkemotor:	Fast DataSearch 3.2.2
Database:	PostgreSQL 7.3.4

En multiprosessor-maskin ble valgt fordi dette ville gi FDS mulighet til å benytte flere prosessorer. Testene, som kun kjører inne i PostgreSQL, har derimot ikke mulighet til å utnytte flere prosessorer fordi en databasetilkobling blir tildelt en prosess, og PostgreSQL bruker ikke tråder for å tillate en prosess å bruke flere prosessorer. Dette kan synes litt urettferdig for de testene som kun kjører i PostgreSQL, men litt av hensikten med å bruke en ekstern indeks er nettopp muligheten til å skalere; ikke bare til flere prosessorer, men også til flere noder. Uten dette vil man nok kunne anta at FDS ikke vil gjøre det noe særlig bedre enn en optimalisert aksessmetode i databasen. Databasens manglende evne til å skalere over flere noder blir på en måte simulert her uten å sette opp FDS i en multinode-konfigurasjon, nettopp fordi PostgreSQL ikke sprer en tilkobling over flere prosesser. Det er imidlertid flere fordeler ved å spre indeksen over flere noder utover det

å utnytte flere prosessorer. Som nevnt tidligere, er disk-I/O ofte flaskehalsen i et databasesystem, og i en multinode-konfigurasjon vil FDS derfor ha enda bedre skaleringsmuligheter.

Krav til datasett som testen ble utført på, er i hovedsak at det er menneskegenerert[35]. I tillegg er det viktig at datasettet er tilstrekkelig stort.

Tabell 6-2: Datasett

Total størrelse	29 MB
Antall dokumenter	7170
Antall ord	4083837
Antall unike ord	251233

Datasettet ble hentet av FDS ved å crawle nettstedet http://odin.dep.no[1]. Vi implementerte en variant av funksjonen *contains_table* for å kunne hente ut innholdet i en FDS-collection og sette det inn i en tabell i databasen:

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

Egenskapene i datasettet er oppgitt i Tabell 6-2. Vi brukte det samme datasettet flere ganger for å oppnå den ønskede datamengden i testene. Testene ble utført med 1-30 datasett. Dette gav dermed på det meste 870 MB med rå tekst. Ideelt sett burde originaldatasettet vært større, slik at vi ikke måtte bruke det samme om igjen flere ganger for å oppnå den ønskede størrelsen. Tilgangen på et datasett som er forholdsvis homogent, det vil si tekst av noenlunde samme type og lengde, er imidlertid dårlig. Tidsbegrensningen for denne oppgaven gjorde at vi valgte å ikke bruke tid på å heve kvaliteten på disse testdataene.

6.7.2. Databaseskalerbarhet

Testen fokuserer på å avdekke skaleringsegenskaper ved transaksjonene som gjøres mot databasen. For databasesystemer er en ”batch”-skalering definert som det å eksekvere samme spørring mot en database som vokser med en bestemt faktor. Denne er lineær dersom størrelsen på databasen og tiden det tar å eksekvere en spørring, vokser med den samme faktoren mens de maskinelle ressursene holdes konstant[36]. Hvis man dobler størrelsen på databasen, vil en spørring skalere lineært kun dersom prosesseringstiden øker med en faktor på to. Dette betyr at man teoretisk, med en dobbelt så kraftig maskin, kan løse et dobbelt så stort problem på like lang tid. Denne egenskapen er avgjørende for om problemet er skalerbart. Ved bruk av aksessmetoder ønsker vi å se en mye bedre skalering enn en lineær. Ved bruk av et B-tre, vil man forvente en økning på *log(n)* IO-operasjoner når indeksen vokser med *n* ord.

Siden vi i dette tilfellet laster samme datasettet flere ganger, vil datastrukturer, slik som inverterte filer, ikke øke. Dette er fordi kun inverteringslisten vil øke (se Figur 4-4) når man legger til det samme datasettet. Både tsearch og FDS bygger selvfølgelig på slike indekser, så for disse vil vi forvente en sublineær økning av transaksjonstiden.

[1] Informasjonstjeneste fra Regjeringen og departementene

6.7.3. Metoder som ble testet

Vi satte opp tre forskjellige tester, altså tre forskjellige måter å gjøre fulltekstsøk i PostgreSQL på:

1. I den første testen brukte vi ordinær SQL LIKE. Søk med denne operatoren vil føre til sekvensielle søk gjennom alle data, og det er derfor forventet at tiden dette tar, vil øke lineært med mengden data i tabellen. Det vil være greit å sammenligne med denne siden det ikke vil være godt nok hvis en annen aksessmetode bruker lengre tid enn et sekvensielt søk. For at testene skal bli så like som mulig, formulerte vi spørringer som i utgangspunktet bør gi likt resultat. SQL LIKE er imidlertid ikke spesielt utrykkskraftig så det er ikke å forvente at svaret vil bli akkurat det samme som ved de andre metodene. Vi brukte følgende LIKE uttrykk i testene som ga 29 treff i ett datasett:

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

2. Den andre metoden vi testet var tsearch2 som er beskrevet i 2.2.3. Tsearch er en indeksbasert teknikk som blir eksekvert i kontekst av PostgreSQL. Spørringen for søk i denne databasen ble slik, og det ga 34 treff i datasettet:

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

3. Den siste metoden er vår nye indeksaksessmetode basert på FDS. Spørringen ble slik og ga 32 treff:

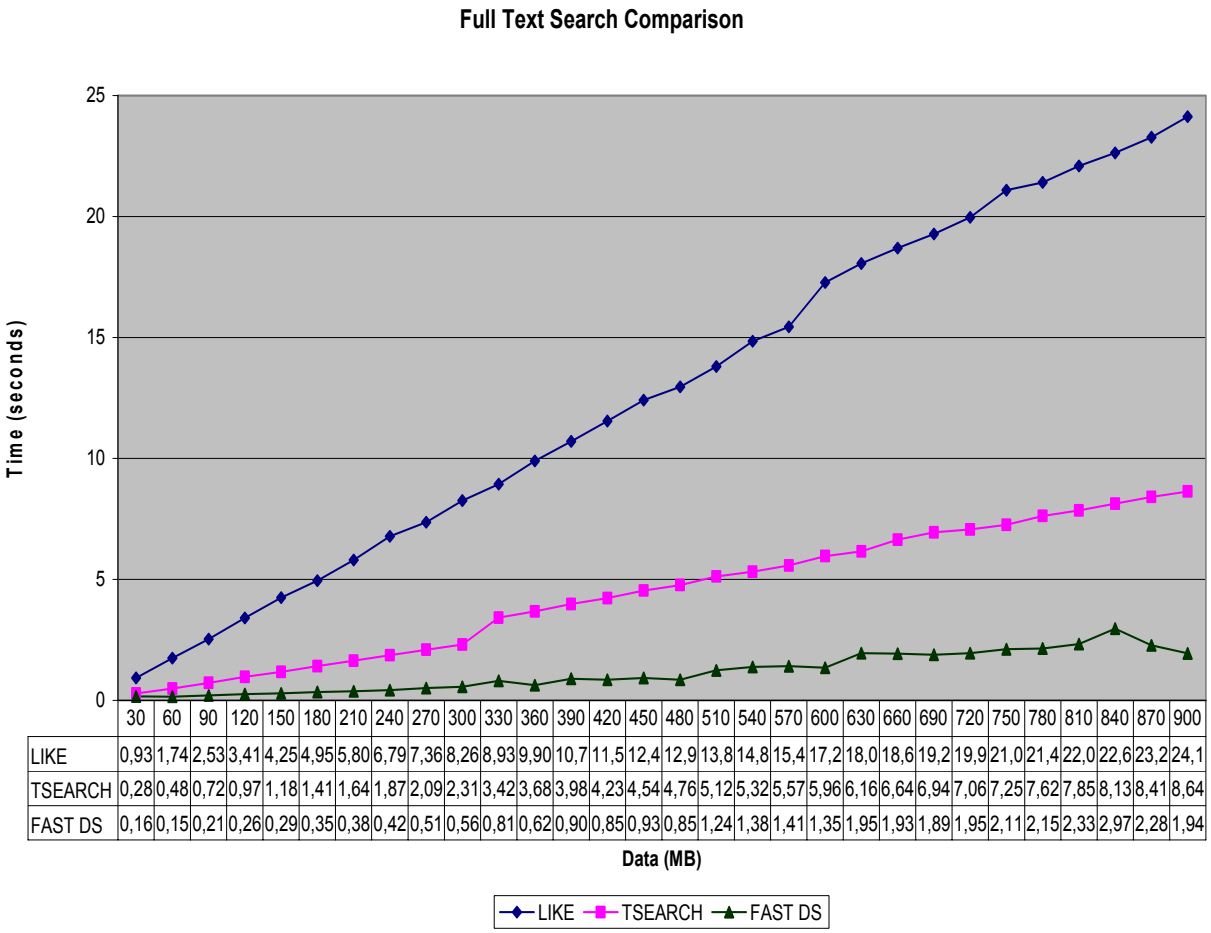
   ```
   SELECT count(*) FROM data_table WHERE data @@ 'hans and grete';
   ```

Vi brukte Unix-verktøyet *time* for å ta tiden på de forskjellige spørringene. Vi tok ikke tiden på innlasting av data eller bygging av indeks. Hovedgrunnen til dette er at FDS bygger opp indeksene mye saktere enn det databasen gjør, og dette gjøres asynkront etter at dokumentene har blitt satt inn, noe som gjør det vanskelig å ta tiden på dette. FDS er heller ikke optimalisert for å gjøre dette raskt på en enkeltnode. Dette førte til at etter innlasting av data i tabellen måtte vi vente noen minutter før vi eksekverte spørringen slik at alle de innsatte dokumentene var indeksert. Dette gjaldt også ved sletting mellom testene. Vi utførte i alt 10 tester med ett til 30 datasett hver gang for alle de tre metodene. Hver av de 10 testene ble kjørt på følgende måte:

```
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
```

6.7.4. Resultat

Vi regnet ut gjennomsnittsdata basert på de 10 kjøringene, og resultatet ble som vist på Figur 6-8.



	30	60	90	120	150	180	210	240	270	300	330	360	390	420	450	480	510	540	570	600	630	660	690	720	750	780	810	840	870	900
LIKE	0,93	1,74	2,53	3,41	4,25	4,95	5,80	6,79	7,36	8,26	8,93	9,90	10,7	11,5	12,4	12,9	13,8	14,8	15,4	17,2	18,0	18,6	19,2	19,9	21,0	21,4	22,0	22,6	23,2	24,1
TSEARCH	0,28	0,48	0,72	0,97	1,18	1,41	1,64	1,87	2,09	2,31	3,42	3,68	3,98	4,23	4,54	4,76	5,12	5,32	5,57	5,96	6,16	6,64	6,94	7,06	7,25	7,62	7,85	8,13	8,41	8,64
FAST DS	0,16	0,15	0,21	0,26	0,29	0,35	0,38	0,42	0,51	0,56	0,81	0,62	0,90	0,85	0,93	0,85	1,24	1,38	1,41	1,35	1,95	1,93	1,89	1,95	2,11	2,15	2,33	2,97	2,28	1,94

Figur 6-8: Testresultat

6.7.5. Diskusjon

Som ventet økte tiden for LIKE-spørringen lineært med hvor mye data som var i databasen. Dette er altså en typisk batch-skalerbar transaksjon som kunne vært skalert hvis man f.eks. spredte data ut over to disker.

Tsearch har også en lineær økning, selv om denne metoden er mye raskere enn et sekvensielt søk. Videre ser vi tydelig hva vi hadde forventet av FDS. Den er ikke jevnt økende slik som de to andre, men den øker. Enkeltresultatene viser også store interne variasjoner. Det kan være flere grunner til dette:

- FDS har mulighet til å benytte flere prosessorer, og et jevnt resultat er derfor ikke å forvente.

- Siden vi laster det samme datasettet flere ganger, øker antall treff linært. Et søk kan sees på som to separate operasjoner: selve søket og opphentingen av resultatet. I tsearch vil et indeksoppslag være nok til å avgjøre hvor mange rader det er i et resultat, mens i FDS genereres det både rangering og dokumentsammendrag for alle dokumentene i resultatsettet. Tiden det tar og generere dette vil derfor øke med antall dokumenter i resultatsettet.
- Kommunikasjonen mellom FDS og databasen vil også øke på grunn av et større resultatsett. Vi kan regne med at tiden som går med til kommunikasjon mellom PostgreSQL og FDS utgjør en stor andel av den totale tiden søket tar, og dette vil øke ettersom resultatet blir større. De andre aksessmetodene har ikke et like stort ”overhead” i kommunikasjonen siden de ligger innenfor databasen.

På grunn av disse forholdene kan vi derfor forvente nettopp det testen viser, at den totale søketiden til FDS, når vi bruker aksessmetoden, vil øke ettersom resultatsettet øker. I forhold til spørringen som gjøres, hadde vi ikke trengt å overføre mer en tallet på antall dokumenter i resultatsettet. På denne måten ville kommunikasjonen blitt nær sagt null. Som tidligere nevnt er søkemotoren også optimliaser for late søk som kun henter de beste resulatene. Hvis implementasjonen vår hadde omfattet slike optimaliseringer ville vi antagelig sett at forskjellen i søketid i FDS var tilnærmet null ettersom mengden data øker – på samme måte som om vi hadde søkt rett i FDS.

Denne testen er ikke noen god ”benchmark” eller bevis for den eksterne indeksens hastighet, til det er det flere forhold som burde vært annerledes: Datasettet burde, som tidligere nevnt, ha vært større og ikke blitt lastet om igjen flere ganger. Spørringene bør dessuten være tilfeldig generert[35]. I forhold til FDS har jeg allerede nevnt en del svakheter i vår implementasjon som gjør at resulatet kunne ha blitt bedre. I tillegg ville det vært mer realitisk for FDS med større last og spørrevolum, gjerne med flere spørringer i sekundet samtidig. En annen ting som hadde vært interessant å se på er maskinvareutnyttelsen. Teoretisk sett burde FDS utnytte dette bedre. Testen hjalp oss imidlertid med å avdekke flaskehalser i vår implementasjon av aksessmetoden, særlig i kommunikasjonen mellom FDS og PostgreSQL, og er ellers en pekepinn på at det etter all sannsynlighet vil skalere godt i forhold til aksessmetoder i databasen.

I dette kapittelet har jeg presentert en implementasjon av en integrasjon mellom PostgreSQL og FDS. I neste kapittel vil jeg diskutere hvor fullstendig denne integrasjonen er og burde være.

Kapittel 7. Integrasjon

I foregående kapittel har jeg presentert begynnelse på en integrasjon mellom en søkemotor og en relasjonsdatabase for å tilføre databasen IR-egenskaper. Denne er naturlig nok noe begrenset på grunn av tiden det tar å utvikle en slik prototyp i forhold til denne oppgavens omfang. Det er derfor interessant å diskutere hvor fullstendig en slik integrasjon kan bli. En full integrasjon bør inneholde[2]:

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 samtidige oppdatering og søk.
5. Skalerbar dokumentindeksering og søk.

7.1. Lagring, indeksering, oppdatering og søk

Dette første punktet omhandler grunnleggende funksjonalitet for enhver type data i en database. Ved å benytte en søkemotor som ekstern indeks og en relasjonsdatabase, har vi ivaretatt alle disse funksjonene. Under vil jeg si kort hvorfor og hvordan dette er ivaretatt:

Lagring

> Dokumentet legges inn i systemet fra databasen. Alle dokumentene lagres her, og man kan derfor se på databasen som ”document-repository”-delen av et IR-system. I databasen vil dokumentene lagres sammen med andre ustrukturerte og strukturerte data. Dette er en av hovedfordelene ved å bruke en relasjonsdatabase, siden ustrukturerte data som oftest er tilknyttet strukturerte data. I tillegg til å lagres i databasen, lagres også dokumentene i den eksterne indeksen.
>
> Ved bruk av en relasjonsdatabase som ”dokument-repository”, oppnår man også datauavhengighet, som er en stor fordel fordi fysisk og logisk datalagring er separert. I tillegg kan man lese, hente, manipulere og slette dokumenter med et standardisert grensesnitt, nemlig SQL.

Indeksering

> Indeksering skjer ved hjelp av databasens aksessmetodegrensesnitt mot en ekstern indeks, en søkemotor. Søkemotoren er basert på inverterte filer som er den vanligste måten å søke i tekstdokumenter på, og tilbyr indeksering, dokumentprosessering og søking med høy ytelse og skalerbarhet. Dessuten gir den eksterne indeksen oss IR-funksjonalitet som vanligvis ikke er å finne i en database, uten at vi behøver implementere dette om igjen fra grunnen av i databasen.

Oppdatering

> Oppdateringer skjer gjennom databasen ved hjelp av vanlig SQL-kommandoer. Indeksen blir så oppdatert ved å slette og/eller sette inn det nye dokumentet i FDS.

Søk

Søk gjøres også gjennom SQL-grensesnittet ved hjelp av en kombinasjon mellom SQL og spørrespråket til FDS. Dette er diskutert nærmere under 7.4.

7.2. Transaksjoner

Transaksjonsstøtte er en vesentlig del av et databasesystem, og en mangel i de fleste IR-systemer. En av hovedhensiktene med en integrasjon er nettopp å tilføre støtte for dette. Vi må derfor sikre at transaksjoner som innebærer bruk at den eksterne indeksen, er omfattet av transaksjonsstøtten i databasesystemet.

Som nevnt i avsnitt 3.2, er en transaksjon ofte forbundet med fire sentrale egenskaper; de såkalte ACID-kravene:

1. Atomitet – alt eller ingenting.
2. Konsistens – I dette tilfellet spesielt mellom den eksterne indeksen og databasen.
3. Isolasjon – mellom transaksjoner.
4. Varighet – endringer i databasen er varige.

I likhet med nær sagt alle andre relasjonsdatabaser på markedet, oppfyller PostgreSQL disse kravene. Det er imidlertid en rekke ting som gjør at det ikke er enkelt å inkludere IR-operasjoner i denne klassiske transaksjonsstøtten. I relasjonsdatabaser er transaksjoner som oftest meget korte og transaksjonssystemet bygger på nettopp denne antagelsen. Det er imidlertid flere applikasjonsområder der det naturlig finnes lange transaksjoner. I IR-systemer er dette tilfellet, da indeksering og oppdatering kan ta lang tid. Dette vil føre til liten samtidighet i klassiske transaksjonssystemer. Et system som skal støtte IR-operasjoner, må derfor bygge på andre prinsipper enn de som klassiske transaksjonssystemer bruker. Forskjellene mellom et tradisjonelt system og et IR-system må analyseres, og kravene til et IR-transaksjonssystem må legges på grunnlag av dette. I 7.2.2 skisserer jeg noen av disse egenskapene.

I et integrert system er det derfor nærliggende å forvente to typer transaksjoner, en for strukturerte data, altså det vi tradisjonelt lagrer i et databasesystem, samt en annen transaksjonsmodell for ustrukturerte data, det vil si tekst. Følgende bør derfor gjelde:

- Transaksjoner som omfatter strukturerte data, bør ha de egenskapene som vanlige transaksjoner har, det vil si full ACID-støtte.
- Operasjoner med IR-data bør kunne utføres i en transaksjon med andre tilpassede egenskaper til denne type data.
- Transaksjoner som involverer både strukturerte og ustrukturerte data, må nødvendigvis oppfylle de strengeste kravene fra begge, noe som klart vil gå på bekostning av ytelsen.

Fokus på effektiv transaksjonsstøtte i IR-systemer har vært liten[9], og enda mindre i et integrert system.

7.2.1. Evaluering av implementasjonen

Vår implementasjon bruker kun transaksjonsstøtten i de systemene vi integrerer, og har ingen egne teknikker for å sikre IR-transaksjoner. FDS har ingen transaksjonsstøtte, mens

PostgreSQL oppfyller som sagt alle ACID-kravene. Det er spesielt to teknikker i PostgreSQL som gjør at ACID blir ivaretatt [37, 38]:

Write Ahead Log (WAL)

Dette er en ”redo”-logg-teknikk som gjør at en ferdig transaksjon alltid vil være bevart fordi en logg over transaksjonen blir skrevet til disk før transaksjonen blir erklært ferdig. Hvis systemet siden skulle gå ned, vil alle ferdige transaksjoner som enda ikke er blitt skrevet til disk, bli skrevet, og alle uferdige transaksjoner som enda ikke har blitt skrevet til disk, simpelthen bli ignorert. WAL gir databasen varighet (4), siden alle fullførte transaksjoner vil være varige, og atomitet (1), siden uferdige transaksjoner vil bli ignorert.

Multi Version Concurrency Control (MVCC)

Dette er en samtidighetskontrollteknikk som gjør at databasen vil ivareta eldre versjoner av et element, i tillegg til den gjeldende versjonen som er lagret i databasen. Formålet med denne teknikken er både å ivareta isolasjon, samt å oppnå høy grad av samtidighet gjennom å minimere antall transaksjoner som må avbrytes.

En transaksjon T1 som ønsker å lese en verdi som en annen transaksjon T2 har skrevet til, må vanligvis avbryte hvis T2 starte senere og ikke har fullført. Denne verdien skal nemlig ikke være synlig for T1, og hvis T2 avbryter, er verdien ugyldig. Dette er gjerne omtalt som en ”skitten les”. I tradisjonelle transaksjonssystemer må da T1 avbryte. Alternativt kan den vente og se om T2 fullfører, men den har allikevel brutt isolasjonsprinsippet, fordi den bruker en verdi som egentlig ikke var tilgjengelig for den når transaksjonen begynte. Verdien som er gyldig for T1, er den verdien elementet hadde før T2 skrev til den. Hvis T1 kan lese denne verdien, behøver den ikke avbryte eller vente. Det er nettopp dette MVCC gjør, ved at eldre verdier av elementene tidsstemples og tas vare på så lenge det finnes pågående transaksjoner som er eldre, og som dermed kan være interessert i denne verdien. Dette sikrer isolasjon (3).

Når det gjelder det siste kravet i ACID, konsistens (4), så betyr det at alle databasens regler, fremmednøkler og andre konsistenskrav skal være oppfylt når en transaksjon er fullført. Dette sikres ved at transaksjoner som ikke oppfyller kravene, avbrytes av databasen.

Transaksjoner i PostgreSQL, inkludert de som omfatter bruk av vår indeks, omfattes altså av disse teknikkene. Ettersom vi ikke lagrer indeksen i databasen, men bruker et eksternt system til dette, vil vi måtte ta spesielle forholdregler for at det skal fungere. Vår integrasjon er derfor *ikke* transaksjonssikker, men så lenge man bruker aksessmetoden til å oppdatere og å søke i den eksterne indeksen, vil den tilby en del av de egenskapene en transaksjon skal ha. For å begrunne dette må vi se på hvert av ACID-kravene:

Atomitet

Så lenge den eksterne indeksen ikke feiler, vil en transaksjon som også involverer denne være atomisk. Dette fordi databasen vil ignorere alle utførte operasjoner på indeksen hvis den blir avbrutt.

La oss ta som eksempel tilfellet der en transaksjon blir avbrutt av bruker eller applikasjon. Innsetting i indeksen vil skje selv om transaksjonen ikke er ferdig

enda, på denne måten vil man kunne søke frem det riktige resultatet også innenfor en transaksjon. Hvis transaksjonen avbrytes, vil tuppelet merkes ugyldig og slettes fra indeksen ved neste VACUUM ved hjelp av *ambulkdelete.* Det ugyldige tuppelet vil filtreres bort av databasen hvis det forekommer i et søkeresultat. Følgende historier for to samtidige transaksjoner T1 og T2 illustrerer dette:

```
T1:                T2:           RESULT:       INDEX:

BEGIN
                   BEGIN
INSERT a                                       INSERT 1 row
                   SELECT a      0 rows
SELECT a                         1 row
ABORT
SELECT a                         0 rows
VACUUM;                                        DELETE 1 row
```

Hvis det oppstår feil under oversendelsen til den eksterne indeksen, blir transaksjonen avbrutt på riktig måte. Problemet er at man ikke vet hvorvidt dokumentet kommer til å bli indeksert eller ikke hvis oversendelsen lykkes. Kallet mot den eksterne indeksen for å sette inn et dokument returnerer nemlig før dokumentet er ferdig indeksert. Det kan oppstå feil under dokumentprosesseringen som skulle tilsi at transaksjonen skulle vært avbrutt, men så lenge man ikke har tilgjengelig et grensesnitt som returnerer etter at dokumentet er ferdig indeksert, vil man ikke kunne garantere atomitet.

Konsistens

Konsistenskravet til den eksterne indeksen vil være at etter en oppdateringstransaksjon må alle endringer i databasen også være reflektert i indeksen. Så lenge vi ikke kan garantere at et dokument blir indeksert når vi setter det inn i den eksterne indeksen, kan det oppstå inkonsistens. Siden innsettingen returnerer før dokumentet er ferdig indeksert og tilgjengelig, er det også inkonsistens mellom indeksen og databasen i tidsrommet fra innsettingen til det er tilgjengelig for søk, og da antar vi at indekseringen ikke vil feile. Dette vil muligens være tolererbart ut i fra et IR-synspunkt, men bryter klart med det tradisjonelle konsistenskravet.

Selv med antagelsen om at en innsetting ikke feiler, er det vanskelig å trekke noen sikre konklusjoner om den eksterne indeksen, siden den åpenbart ikke er utviklet med tanke på transaksjonssikkerhet. Hvis det skulle oppstå en feil slik at den eksterne indeksen krasjer etter indeksering, har vi ingen garanti for at alt er skrevet til disk, og inkonsistens kan oppstå. Konsistenskravet ser derfor ut til å være vanskelig å oppnå uten mer støtte for transaksjoner i den eksterne indeksen. Dette henger sammen med FDS's mulighet for gjenoppretting og garanti for varighet og atomitet.

Isolasjon

I databasen blir dette som nevnt ivaretatt av MVCC, med hensyn til hva slags verdier som er synlige for samtidige transaksjoner. Dette gjelder også ved bruk av den eksterne indeksen. Databasen vil selv avgjøre om et tuppel er gyldig for en gitt transaksjon eller ikke, og så lenge den eksterne indeksen returnerer alle aktuelle tupler, vil dette fungere. Det er også slik at så lenge det eksisterer transaksjoner som kan være interessert i en eldre versjon av et element, vil denne ikke slettes fra

indeksen før disse transaksjonene har fullført. La oss se på følgende historier for to samtidige transaksjoner, T1 og T2:

```
T1:                T2:                 INDEX:

BEGIN
                   BEGIN
UPDATE a                               INSERT 1 row
                   SELECT a
COMMIT
VACUUM                                 DELETE 0 rows
                   COMMIT
VACUUM                                 DELETE 1 row
```

Som vi ser vil T1 oppdatere a, dette vil føre til at en ny versjon av a vil bli satt inn i indeksen og den gamle vil bli merket ugyldig for T1. For T2 vil denne forsatt være gyldig, og den gamle a-verdien vil først bli slettet i indeksen etter at T2 har fullført. I mellomtiden vil den nye a-verdien være ugyldig for T2.

Når det gjelder operasjoner som kan komme i konflikt med hverandre, bruker PostgreSQL MVCC for å garantere serialiserbarhet[23]. Som over vil dette fungere også ved bruk av den eksterne indeksen. På grunn av egenskaper ved MVCC er det kun skriveoperasjoner som er i konflikt med hverandre, siden det ikke er mulig å gjøre en ”skitten les”. I slike tilfeller vil den siste transaksjonen vente til den foregående har fullført for å få lov til å skrive til samme element.

Varighet

Varigheten til data i databasen for avsluttede transaksjoner ivaretas av databasen. Den eksterne indeksen har derimot ingen gjenopprettingsfunksjonalitet, og databasen har ingen direkte kontroll med den eksterne indeksen, slik at databasen og indeksen kan være inkonsistente og indeksen må derfor bygges på nytt. Det vil imidlertid aldri føre til datatap, så dette bryter egentlig ikke med varighetskravet, men med konsistenskravet.

Et generelt problem er altså at vi kan havne i situasjoner der indeksen ikke er i samsvar med databasen. På grunn av manglende måter å detektere og reparere slike feil på, må en manuelt reindeksere dersom slike feil skulle oppstå. Dette er imidlertid helt uakseptabelt, både fordi et indeksoppslag vil returnere for få tupler, fordi man vil kunne få feil svar der en ikke er klar over at indeksen er inkonsistent, og fordi en fullstendig reindeksering av en fulltekst indeks ikke er akseptabel. Dette er en operasjon som kan ta dagesvis for en datamengde som er tilstrekkelig stor.

På flere områder ser det imidlertid ut til at den eksterne indeksen fungerer i en transaksjon. Det er i hovedsak to grunner til dette:

For det første lagrer vi i den eksterne indeksen informasjon om hvor data er plassert på disk i form at et blokk- og offsetnummer. Det er på dette nivået transaksjonshåndteringen til PostgreSQL skjer, og den vet selv hvilke blokker som er gyldige tupler. Det vil derimot ikke fungere å bruke den eksterne indeksen direkte til å søke i, det vil si å søke direkte i FDS ved hjelp av for eksempel webgrensesnittet. Årsaken til det er at det ikke foregår noen kommunikasjon mellom FDS og PostgreSQL om hvilke tupler som er gyldige og ugyldige. For å få dette til må integrasjonen mellom de to systemene bli mye tettere enn det vi oppnår ved å integrere den som en aksessmetode. Vi kan for eksempel

gjøre dette ved at PostgreSQL buffer manager kommuniserer direkte med tilsvarende modul i FDS.

Vår implementasjon innholder også en tabellfunksjon, *contains_table,* som heller ikke vil fungere fordi den ikke går gjennom aksessmetodegrensesnittet. Denne kan imidlertid utvides til å sjekke om tuplene den returnerer er gyldige, men inntil da vil den ikke en gang fjerne slettede tupler før etter VACUUM.

For det andre overlater ikke PostgreSQL transaksjonshåndteringen til aksessmetoden, men gjør selv alt nødvendig arbeid. I dette ligger det selvfølgelig en del antagelser, blant annet om at indeksen fysisk ligger innenfor databasen, noe den ikke gjør i vårt tilfelle. Dette er grunnen til at det kan oppstå inkonsistens. Kommandoer som *TRUNCATE, DROP INDEX, DROP TABLE* og så videre, sletter simpelthen buffer der den tror indeksen ligger, men signaliserer ikke grensesnittet. Som jeg påpekte i implementasjonen, vil dette kreve utvidelse av det interne grensesnittet til PostgreSQL. På grunn av MVCC fører ikke disse begrensningene til feil svar eller en inkonsistent indeks, men vi har et problem når det gjelder oppryddingen av gamle tupler som skulle vært fjernet ved en VACUUM-operasjon.

Når det gjelder transaksjonsmodellen som denne implementasjonen bruker, er det klart at dette absolutt ikke er optimalt for denne typen data. Dokumentprosessering og indeksering er langvarig, og det å låse transaksjonen og databaseelementer mens dette pågår vil være svært uhensiktsmessig. Innsetting av dokumenter er en typisk ”batch”-jobb, og den eksterne indeksen er optimalisert for å prosessere flere dokumenter i en ”pipeline”. Innsetting av ett og ett dokument, samt å vente til det er helt ferdig indeksert er unødvendig tidkrevende.

Vi kan oppsummere transaksjonsstøtten i implementasjonen slik:

- **Atomitet** kan ikke garanteres fordi FDS ikke kan garantere at et dokument blir indeksert når vi har lagt det til i den eksterne indeksen. Hvis vi hadde hatt tilgjengelig et API-kall som ikke returnerte før dokumentet var ferdig indeksert, kunne vi muligens garantert dette hvis FDS kunne garantere at det var skrevet til disk[1].
- **Konsistens** mellom den eksterne indeksen og databasen kan ikke garanteres fordi PostgreSQL antar at indeksen er fysisk plassert i databasen. For å oppnå konsistens må operasjoner på de buffere som indeksen egentlig skulle ligget på, føre til operasjoner på den eksterne indeksen. Konsistens kan heller ikke garanteres av samme grunn som under atomitet.
- **Isolasjon** er oppfylt hvis vi følger antagelsen om at den er atomisk. MVCC i databasen vil også omfatte data i den eksterne indeksen.

[1] Neste versjon av FDS, versjon 4.0, tilbyr et API som gjør det mulig å få et tilbakekall når et ”batch” er ferdig indeksert og garantert skrevet til disk.

- **Varighet** er sikret fordi dokumentene lagres i databasen. Vi kan derimot ikke garantere at indeksen er varig, og dokumentene må derfor reindekseres hvis det skulle oppstå feil i den eksterne indeksen, for eksempel at den krasjer. Et annet alternativ er at FDS tilbyr dette[1].

7.2.2. Karakteristikk av IR-transaksjoner

Siden transaksjonshåndteringen til vanlige RDBMS-systemer ikke er optimalt for prosessering av fulltekstdokumenter, må vi først sette opp en del karakteristika for en transaksjon i et IR-system, slik at vi kan utvikle transaksjonshåndteringen til å bli mer effektiv.

1. De fleste transaksjoner i et IR-system vil være søk, altså lesetransaksjoner. Det er derimot ikke sikkert at det vil være denne typen transaksjoner som vil oppta mest systemtid, snarere tvert i mot. Et IR-system er i stor grad optimalisert for søk og ikke for oppdateringer. Søk må imidlertid prioriteres høyere enn oppdateringer.

2. I et IR-system kan man anta at det ikke eksisterer avhengighet mellom dokumenter[9], til forskjell fra data i en tradisjonell database. Hvis man er villig til å gjøre en slik avgrensning, kan det ha store innvirkninger på transaksjonssystemets effektivitet, siden ingen transaksjoner behøver å avbrytes. Dette er fordi det ikke vil kunne oppstå logiske feil.

3. Oppdateringer skjer som oftest ved at vi legger data til indeksen, på grunn av egenskapene ved inverterte indekser. Vi kan anta at dette er normen, og at fjerning av data fra de inverterte filene tilhører sjeldenhetene.

4. De siste dokumentene som ble indeksert, er ofte de som er mest relevante. Derfor er det viktig å ta hensyn til de dokumentene som nylig er mottatt av systemet når man prosesserer en spørring.

5. Oppdateringstransaksjonene kan være langvarige – dokumentprosessering og indeksering tar tid. Oppdateringer skjer gjerne på en større mengde dokumenter av gangen.

6. Full reindeksering vil ikke være akseptabelt. Heller ikke vil det være akseptabelt å starte en stor ”batch”-transaksjon helt fra starten av, selv om det skulle oppstå feil under indeksering av et dokument.

7.2.3. Hva vi kunne tenke oss i et integrert system

La oss så skissere hva slags egenskaper ved transaksjoner vi kunne tenke oss i et kombinert IR og RDBMS-system.

I vår implementasjon ligger det en stor svakhet i oppdateringstransaksjonene fordi dokumentprosessering er tidkrevende. For å oppnå god ytelse er vi avhengig av å kunne

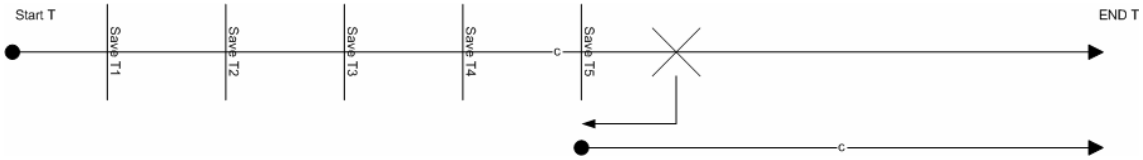
[1] Internt i FDS vil dette være problematisk, blant annet fordi den opererer med flere versjoner av indeksen spredt utover flere noder. Når versjonene byttes, det vil si at ny blir erstattet med gammel, vil den ikke være synkronisert over alle nodene med en gang. Rent intuitivt kan en tro at en synkronisering av dette vil føre til en alt for stor nedgradering av ytelsen i forhold til søk. Det er imidlertid sikkert mulig å løse dette på en effektiv måte.

sende flere dokumenter inn i dokumentprosesseringen samtidig, slik at de kan behandles i en pipeline. Vi kan ikke vente med å sette inn det neste til det foregående er ferdig indeksert, og vi er derfor avhengig av å kunne behandle oppdateringen i "batcher". Vi vet at slike "batcher" er langvarige, siden oppdateringer gjerne skjer på flere dokumenter og indekseringen tar tid. Fra litteraturen har vi flere transaksjonsmodeller som er utviklet med tanke på at transaksjonene er langvarige. Mini-batch eller Saga[21] synes som gode valg[9]. Det er flere grunner til dette:

- Disse transaksjonsmodellene deler opp en lang transaksjon i flere mindre. La oss si at en subtransaksjon består av ett dokument - hvis det skulle oppstå en feil under prosesseringen, vil vi kunne gjenoppta transaksjonen fra forrige fullførte subtransaksjon eller eventuelt velge å avbryte, men likevel la de fullførte subbtransaksjonene være. Ved hjelp av en slik transaksjonsmodell slipper vi derfor å starte en lengre oppdateringstransaksjon på nytt.
- Selv om vi deler opp transaksjonen i flere subbtransaksjoner, kan vi bevare atomitetsprinsippet for hele hovedtransaksjonen. Når vi fullfører hovedtransaksjonen, må vi vite at alle dokumentene i transaksjonen er ferdig indeksert. La oss si vi starter en transaksjon og setter inn noen hundre dokumenter. Disse vil da kunne bli sendt inn i pipelinen, og når vi gjør commit, vil ikke denne returnere før alle dokumentene er ferdig indeksert.
- Ved å bruke en slik transaksjonsmodell kan vi også velge å variere hvordan isolasjonen mellom transaksjoner skal være. Slik som beskrevet i forrige avsnitt, vil ofte de siste dokumentene som har blitt prosessert, være relevante for pågående søk. For å oppnå en mer oppdatert database kan vi la resultatet av subtransaksjonene være synlig utenfor hovedtransaksjonen selv om denne ikke har fullført, noe som bryter med det tradisjonelle isolasjonskravet. Dette vil for eksempel være ønskelig hvis vi skal foreta en større oppdatering av innhold på websider som ikke har noen logiske avhengigheter mellom seg. I tilfelle transaksjonen inneholder avhengige dokumenter eller modifiserer eller leser relasjonelle data i databasen, kan vi hindre dette ved å isolere alle endringer i subtransaksjonene til hovedtransaksjonen er fullført.

Et annet viktig element ved bruk av slike langvarige transaksjoner er å sikre høy grad av samtidighet. De lange oppdateringstransaksjonene trenger ikke holde på låser slik at lesetransaksjoner må vente. I vår implementasjon vil ikke dette være noe problem, siden databasens MVCC sikrer at eldre versjoner av tuppelet er tilgjengelig slik at vi slipper å vente på oppdateringstransaksjoner.

Vi trenger altså et transaksjonsbegrep i den eksterne indeksen for å støtte en type oppdateringstransaksjon som har de egenskaper som er nevnt over. Den eksterne indeksen må i tillegg kunne gjøre gjenoppretting og garantere atomiske operasjoner.



Figur 7-1: Figuren viser en Saga-transaksjon. Transaksjon T krasjer og rulles tilbake til siste lagringspunkt (T5) og gjenopptas derfra.

7.2.4. En transaksjonsmodell for FDS

Vi skal se på to nivåer av transaksjonsstøtte i den eksterne indeksen. For det første det vi trenger for å støtte transaksjonsegenskapene vi har nevnt i forrige avsnitt, og så hvordan vi kan tilby brukeren å spørre direkte mot indeksen.

Først antar vi at man gjør alle oppdateringer og spørringer gjennom databasen. Vi må tilføre den eksterne indeksen selve begrepet om en transaksjon, og vi trenger en transaksjonshåndteringsmodul i FDS for å støtte dette. Grunnen til at vi må innføre denne i FDS og ikke kan overlate det til transaksjonshåndtereren i PostgreSQL, er fordi vi ønsker denne ”batch”-oppdateringstransaksjonen, og det medfører at FDS *må* vite hvilke dokumenter som hører til hvilken transaksjon for å kunne garantere transaksjonens atomitet, samt gjenoppretting.

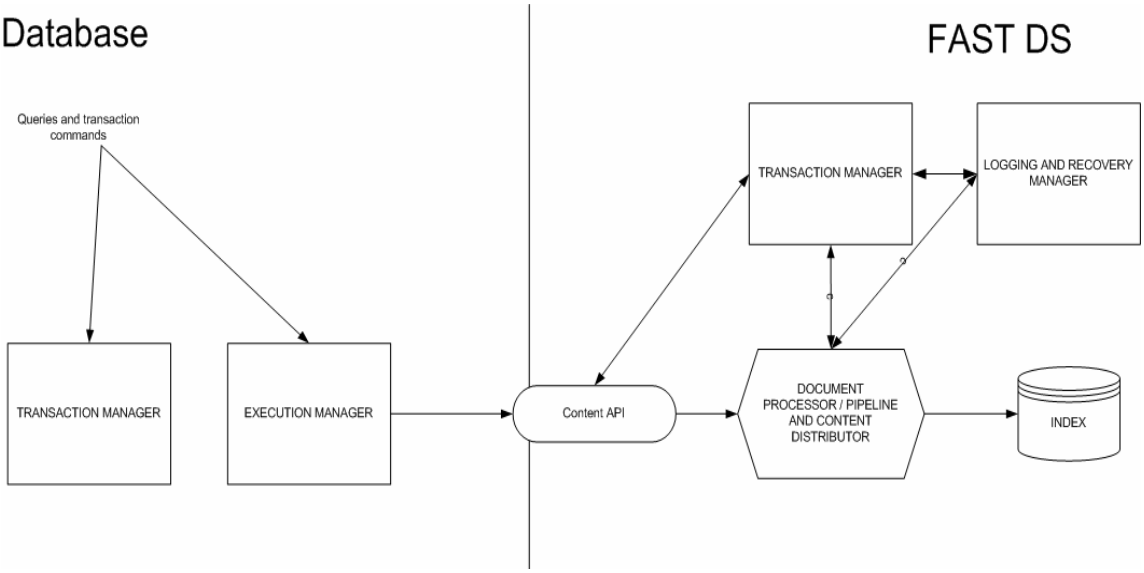
FDS trenger også en egen gjenopprettingshåndterer. Begrunnelsen for dette er for det første at indeksen ikke fysisk er håndtert av databasen, og gjenoppretting kan da heller ikke gjøres av tilsvarende modul i PostgreSQL. I tillegg ønsker vi å kunne gjenoppta en avbrutt transaksjon fra sist fullførte subtransaksjon, noe som er ulikt måten databasen gjør det på. Databasen trenger ikke vite om dette, noe som gjør at vi kan unngå å endre databasens gjenoppretting. Dermed må FDS gjøre dette selv.

Transaksjonshåndtereren (TM[1]) i FDS må derfor ha ansvar for følgende:

- Når en transaksjon startes, må den tildeles et unikt transaksjonsnummer. Dokumenter som sendes til dokumentprosessering og indeksering, må assosieres med dette transaksjonsnummeret. Ved jevne mellomrom må TM logge statusen for transaksjonen til disk, det vil si at den må logge hvilke dokumenter som er prosessert og hvilke som gjenstår. Hvis systemet krasjer, kan transaksjonen gjenopptas fra sist lagrede ”savepoint”. Når det gjelder indekseringsoperasjoner, må et slikt ”savepoint” inneholde informasjon om hvilke inverterte filer som er oppdatert og hvilke som ikke er det. I tilfeller der vi ikke har logiske avhengigheter mellom dokumenter, trenger man ikke gjøre noe form for ”undo”-operasjon, og vi trenger derfor ikke logge førverdier [9]. Hvis transaksjonen allikevel skal avbryte, vil dokumenter som ble satt inn under transaksjonen, uansett bli merket som ugyldige av MVCC i databasen.
- Når alle dokumentene er ferdig indeksert, kan TM gi beskjed om at transaksjonen er fullført. TM må da kunne garantere at alle endringene er skrevet til disk. Det kan være en utfordring, spesielt fordi FDS er distribuert, og de inverterte filene potensielt ligger på flere forskjellige noder, både vertikalt og horisontalt fragmentert.

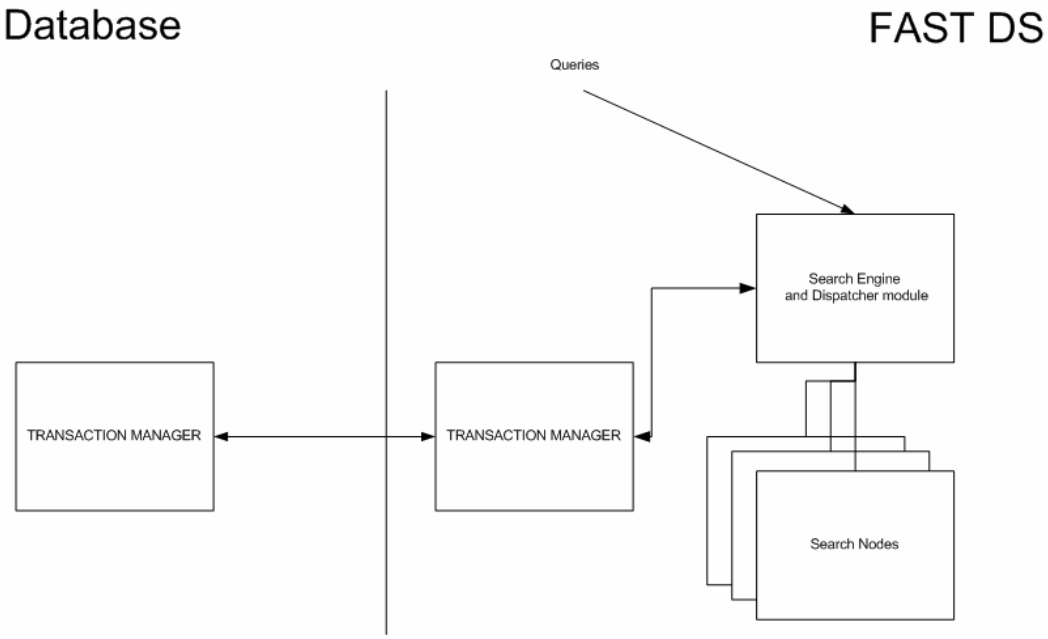
Atomisitet er sikret fordi FDS ikke vil returnere fra innsettingsoperasjonen før alle dokumentene i batchen er indeksert. Indeksen vil da også være konsistent med databasen. For at den skal være konsistent videre, må FDS signaliseres når PostgreSQL gjør operasjoner på selve indeksen. Dette gjelder f.eks. ved DROP TABLE. Isolasjon er sikret gjennom databasens MVCC. Varighet må sikres ved at FDS selv utfører gjenoppretting ved hjelp av sin egen gjenopprettingsmodul.

[1] Transaction Manager



Figur 7-2: Figuren viser noen av de involverte komponentene i et transaksjonssystem mellom en database og FDS. TM i FDS overvåker innsetting av dokumenter i en batch og skriver savepoint til en logg. FDS har også en gjenopprettingsmodul for å sikre varighet.

Så til hva som må til for eventuelt å gjøre det mulig å søke direkte i FDS uten å måtte gå gjennom databasens aksessmetodegrensesnitt. Det enkleste er å bygge på databasens kjennskap til om et element er gyldig eller ikke. Det betyr også at det er databasen som kontrollerer samtidighet og isolasjon, noe som vil være uhensiktmessig å implementere en gang til, samt vanskelig å holde konsistent med hverandre. Dette kan løses på følgende måte: Dispatcher-modulen (DM) i FDS, som setter sammen resultatet fra en spørring, kan på lik linje med PostgreSQL, filtrere bort elementer som ikke er gyldige for den aktuelle transaksjonen. DM starter så en transaksjon i PostgreSQL og spør om et eller flere gitte elementer er gyldige eller ikke innenfor denne transaksjonen, før den presenterer resultatet. Dette er illustrert ved Figur 7-3. Tatt i betraktning et stort antall spørringer vil, imidlertid databasen bli en flaskehals i dette systemet, siden vi kun kan ha en database å kontrollere transaksjonen mot mens FDS vil kunne skaleres til et ønsket antall noder.



Figur 7-3: Når søkemotoren mottar en spørring startes en transaksjon også i databasen, og databasen avgjør om de enkelte tupler er gyldig innenfor transaksjonen eller ikke.

Hvis derimot FDS selv skal kunne avgjøre om et gitt element er gyldig eller ikke, må vi kunne tolerere at vi ikke nødvendigvis vil kunne få det korrekte svaret. Oppdateringstransaksjoner vil FDS selv avgjøre om er gyldige eller ikke, fordi TM til FDS vet hvilke dokumenter som er i en pågående transaksjon, og de vil kunne fjernes. Dokumenter som er slettet, men som allikevel er i indeksen fordi det ikke er gjort VACUUM eller det finnes pågående transaksjoner som vil kunne ha interesse av disse dokumentene, er det derimot verre med. Her må eventuelt DM vedlikeholde en liste med ugyldige dokumenter, en slags ”blacklist”, og bli signalisert hver gang et dokument blir slettet fra databasen. DM må selv avgjøre på hvilket tidspunkt dokumentet ble merket ugyldig, og hvorvidt et dokument er gyldig for et gitt søk. Søketransaksjonene er imidlertid så korte at dette ikke vil by på store problemer. Og igjen, hvis vi antar at sletting og oppdatering av dokumenter er langt sjeldnere enn søk, vil kommunikasjonen bli mer effektiv.

7.3. Samtidighet

Både FDS og PostgreSQL tilbyr stor grad av samtidighet. Siden oppdateringstransaksjoner mot den eksterne indeksen er tidkrevende slik det er implementert nå, er det viktig at ikke andre transaksjoner må vente på eventuelle låser som tas av oppdateringstransaksjonen. PostgreSQL’s MVCC gjør at dette ikke er noe problem. Det integrerte systemet gir derfor høy grad av samtidighet.

7.4. Spørrespråk

Spørrespråk er en svært viktig del av et integrert IR- og RDBMS-system, og en god integrasjon i SQL er helt nødvendig for at et slikt system skal lykkes. Dette vil føre til den nødvendige aksepten, samt forenkle en utvidelse av eksisterende systemer og applikasjoner til å omfatte fulltekstsøk. Videre er det viktig at disse utvidelsene, slik som

resten av SQL, følger en standard. Dette har både industri og standardiseringsorganer tatt inn over seg, og mye tyder på at SQL/MM[3] vil bli implementert og brukt.

Vi kan se på integrasjon av et fulltekstspørrespråk i SQL i to deler: et spørrespråk som lar brukerne definere et søkemønster, og så hvordan dette søkespråket skal integreres i SQL, for eksempel i spørringens WHERE-del.

7.4.1. Integrasjon mot SQL

Spørrespråket i vår implementasjon integrerer SQL og spørrespråket til den eksterne indeksen ved hjelp av en CONTAINS-operator og en CONTAINS_TABLE-funksjon. Disse kan brukes på følgende måte:

```
SELECT * FROM article WHERE text CONTAINS 'fast search
expression';

SELECT id, rank, teaser
FROM contains_table('article','text','fast search expression');
```

Grunnen til dette er først og fremst oppbygningen av PostgreSQL som gjør det mulig å evaluere operatorer ved hjelp av indeksaksessmetoder. I SQL/MM gjøres dette ved hjelp av en metode til den brukerdefinerte datatypen *FullText*:

```
SELECT * FROM article WHERE text.CONTAINS(FT_Pattern);
```

Her er *FT_Pattern* et SQL/MM-søkeuttrykk. Alternativt kan det implementeres som en funksjon:

```
SELECT * FROM article
WHERE CONTAINS(text, FT_Pattern);
```

ADT er en utvidelse av SQL, som gjør at typer kan ha egne metoder. PostgreSQL støtter ikke slike brukerdefinerte typer, men både funksjoner og operatorer kan "overloades", det vil si at de kan implementeres forskjellig etter hva slags datatype de blir kalt med. Grensesnittet til PostgreSQL mot indeksaksessmetoder gjør det også enklest å benytte seg av en operator, og vi har derfor implementert det slik. Bruk av en funksjon er beskrevet i SQL/MM som en alternativ måte å være konform med standarden på. Siden dette ville ført til endringer i hvordan PostgreSQL evaluerer utrykk, har vi heller ikke valgt denne måten å gjøre det på. Disse tre forskjellige måtene å binde søkeuttrykket sammen med resten av en SQL-spørring på er imidlertid svært like. Oracle DB og Microsoft SQL Server bruker, som beskrevet tidligere, en funksjon.

I tilegg til tuppelet som oppfyller søkekriteriet, er det i fritekstsøk ofte interessant med to attributter til, en rangering og et dokumentsammendrag. I likhet med MS SQL Server har vi implementert det ved å lage en tabellfunksjon som returnerer disse, samt et dokument ID. I SQL/MM returneres tekstens rangering i forhold til spørringen ved hjelp av funksjonen *score:*

```
SELECT docId, text.score(FT_Pattern)
FROM article WHERE text.CONTAINS(FT_Pattern);
```

I forhold til SQL/MM virker en tabellfunksjon som en like god løsning fordi rangeringen kommer som et attributt fra resultatsettet, og man behøver da ikke skrive inn søkeuttrykket to ganger. På den andre siden kan det ha en hensikt nettopp å gjøre dette,

fordi man da har muligheten til å rangere på noe annet enn det man søker på. I vår implementasjon er dette også mulig ved å angi rangeringsutrykket i selve søkeuttrykket ved hjelp av nøkkelordet *RANK,* etterfulgt av nøkkeord man ønsker å rangere etter.

Når det gjelder dokumentsammendrag er dette en mangel i SQL/MM. Ofte er dette en naturlig del av et IR-system, og blant annet Oracle tilbyr dette i sin fulltekstintegrasjon. En naturlig utvidelse av SQL/MM ville være å legge til en ny metode i typen *FullText*:

```
SELECT text.teaser() FROM …
```

for et statisk dokumentsammendrag eller:

```
SELECT text.teaser(FT_PATTERN) FROM …
```

for et dynamisk dokumentsammendrag basert på spørringen.

7.4.2. IR-spørrespråk

Selve søkeuttrykket oppgis i vår implementasjon som et FDS søkeuttrykk, på samme måte som man i SQL/MM oppgir et *FT_Pattern* uttrykk.

Dette søkeuttrykket følger ikke direkte av noen standard, slik som et *FT_Pattern* er, men må følge den eksterne indeksens spørrespråk. Det er flere ulemper med dette, for eksempel hvis man vil bytte ut den eksterne indeksen med en annen. Da blir man nødt til å skrive om alle spørringene for å tilpasse seg et annet spørrespråk. Nå er imidlertid spørrespråket til FDS ganske enkelt, kun boolske uttrykk sammensatt med AND, OR, ANDNOT og gruppert med parenteser. Dette vil nok de fleste søkemotorer og IR-systemer støtte. At spørrespråket er såpass enkelt, er selvfølgelig også en stor svakhet. Slik vi har beskrevet i 5.4, mangler FDS en del IR-funksjonalitet. En del av den funksjonaliteten er ikke tilgjengelig fra spørrespråket og noe er det ikke er mulig å endre per spørring. Grunnen til at spørrespråket til FDS er såpass enkelt, er applikasjonsområdet det opprinnelig er tilpasset til, nemlig websøk. For at FDS skal egne seg for en integrering, bør spørrespråket bli mer avansert.

SQL/MM er en grundig standard og har et spørrespråk som vil dekke de fleste behov for IR-funksjonalitet i databaser. I forhold til denne blir mulighetene i FDS små. Det er imidlertid ikke, slik jeg kan se det, noe som gjør at ikke spørrespråket til FDS kan utvides til å omfatte en del mer av SQL/MM. Funksjonalitet som allerede er inkludert i FDS, men som ikke er omfattet av spørrespråket, eller som det i hvert fall ligger gode muligheter i FDS for å støtte på grunn av måten dette er bygd opp på, bør kunne gjøres tilgjengelig i spørrespråket. I standardens[18] del 4 beskrives de forskjellige konseptene om IR-funksjonaliteten som brukes i denne standarden. Flere av disse områdene støttes ikke av spørrespråket til FDS, men det er likevel sammenfallende funksjonalitet. Dette kan vi se av Tabell 5-1. Områder som FDS kan støtte er:

- Kontekstsøk/ordnærhet, punkt 4.2.5 i standarden.
- Boolske operatorer (&,|), punkt 4.2.6
- Ord og fraser, punkt 4.2.1 og .4.2.2 er allerede konformt.
- Bruk og jokertegn i fraser og ord, punkt 4.2.3.1 og 4.2.4
- Basisform og bøyninger av ord, punkt 4.2.3.4

- Synonymutvidelse, deler av punkt 4.2.4.2

For å utvide vår implementasjon til bruk av SQL/MM-søkeuttrykk har man to valg: Man kan enten implementere SQL/MM spørrespråk i FDS som et alternativ til FAST's eget spørrespråk, eller man kan foreta en oversettelse mellom de to språkene i databasen. Det første er antagelig det beste alternativet, fordi FDS allerede vil inneholde mye av det som skal til for gjøre en slik implementasjon, samtidig som FDS har behov for et mer avansert spørrespråk også utenfor databaser. Så vidt meg bekjent finnes det ikke noen standard for spørrespråk i IR-systemer eller søkemotorer. SQL/MM synes å dekke de fleste behov for et mer avansert spørrespråk. Det å kunne tilby et slik alternativ i tillegg til det mer ad-hoc-pregede språket som allerede finnes, kan være svært nyttig. Selvfølgelig er det spesielt nyttig i forhold til relasjonsdatabaser, at søkeuttrykket følger en SQL-standard, men i andre sammenhenger er det gunstig å kunne tilby et standardisert spørrespråk.

Som nevnt i kapittel 5 er det også områder der FDS har funksjonalitet som SQL/MM mangler støtte for. I forrige avsnitt nevnte jeg dokumentsammendrag, og ulik vekting av nøkkelord er f.eks. et annet område. Dette kunne f.eks. vært gjort slik:

```
text.CONTAINS('dog RANK 3 & cat');
```

Dette kan imidlertid simuleres i SQL, uten direkte støtte for det i SQL/MM, slik:

```
SELECT docID, text.score('cat')+text.score('dog')*3
FROM article WHERE text.CONTAINS('cat & dog');
```

Områder som kategorisering, drill-down søk, og find-similar-søk er ikke mulig å simulere med SQL/MM og det er dermed opp til implementasjoner og finne løsninger på dette. I vår implementasjon kunne vi løst dette ved å lage egne utvidelser, f.eks. nye funksjoner som brukte grensesnittet til FDS.

7.5. Skalerbarhet

Fulltekstsøk er helt avhengig av en skalerbar arkitektur og mulighet for parallellisering. Den eksterne indeksen i vårt system er laget nettopp for å tilby et skalerbart søkesystem. De forskjellige oppgavene til FDS: indeksering, dokumentprosessering og søking kan spres til et ønsket antall forskjellige noder. Den inverterte indeksen kan både fragmenteres og dupliseres over flere noder, slik at flere noder kan arbeide med samme søk parallelt. Den eksterne indeksen i vår implementasjon er altså i aller høyeste grad skalerbar og oppfyller derfor krav man måtte ha til et skalerbart fulltekstsøk.

Databasen, som fungerer som systemets "dokument repository", er, slik som relasjonelle databaser tradisjonelt er, svært monolittisk og vanskelig å distribuere over flere noder. Fordelen i forhold til å integrere fulltekstfunksjonalitet inn i en database, er at databasen neppe blir noen flaskehals, i hvert fall ikke i forhold til IR-funksjonalitet. De fleste databaser, inkludert PostgreSQL, har mulighet for replikeringsløsninger hvis man ønsker flere noder å kunne spørre fra. Kommunikasjon med den eksterne indeksen bør gå gjennom databasen, for blant annet å oppnå transaksjonsstøtte, men dette kan da lett bli en flaskehals. I et IR-system er størsteparten av transaksjonene lesetransaksjoner, og det er også viktig at systemet optimaliseres for dette.

Kapittel 8. Konklusjon og videre arbeid

I denne oppgaven har jeg jobbet med en idé om å integrere en søkemotor og en relasjonsdatabase for å kunne gjøre effektive fritekstsøk i databasen. Ut i fra tidligere implementasjoner, forskning og standarder har jeg kartlagt hvilken funksjonalitet en søkemotor bør ha for at en integrasjon skal være hensiktsmessig. Gjennom dette arbeidet har jeg sett at søkemotorer er langt fremme i utviklingen på flere viktige områder innenfor fritekstsøk; spesielt når det gjelder effektivitet, skalerbarhet og teknikker for å bedre presisjon og gjenfinning ut fra enkle spørringer. På visse områder, slik som utrykkskraftige søkespråk og transaksjoner, later det til at søkemotorene har kommet for kort i forhold til en integrasjon mot databaser. Jeg har imidlertid ikke sett på andre søkemotorer enn FDS og et eksempel på dette er noe mangelfullt. Likevel vil jeg konkludere med at en integrasjon, ut fra et teoretisk perspektiv, er interessant og høyst relevant.

Videre har vi implementert en prototyp, og arbeidet med denne har bestått i å sette sammen to meget store og kompliserte systemer på en måte som ikke har vært forsøkt før. Med denne prototypimplementasjonen har vi vist at en slik integrasjon er mulig, samt avdekket en rekke problemområder som det må jobbes videre med. Disse områdene vil jeg komme tilbake til i neste avsnitt. Vi har også vist, gjennom en enkel test, at integrasjonen har en bedre ytelse enn løsninger basert kun på databasen. Fra arbeidet med implementasjonen har jeg fått innsikt i, samt mulighet for å teste ut problemområder som spørrespråk og transaksjonsstøtte. Ut fra dette har jeg diskutert mulige løsninger og mulige retninger for videre arbeid med dette.

Dermed skulle problemstillingene være besvart med at:

1. Søkemotorer kan integreres med en relasjonsdatabase, og dette vil være hensiktmessig fordi søkemotorer på flere sentrale områder har kommet lengre enn databaser.

2. Søkemotorer har i stor grad den IR-funksjonaliteten som trengs for at en integrasjon skal være vellykket, men mangler et uttrykkskraftig spørrespråk og transaksjonsstøtte.

3. Spørrespråket SQL og fritekstspørrespråk lar seg godt integrere.

8.1. Videre arbeid

Arbeidet som er presentert i denne oppgaven er en begynnelse og idé til en mulig måte å forbedre IR-funksjonalitet i databaser på. Flere utvidelser av implementasjonen og testing er nødvendig for at dette skal kunne være anvendbart. Jeg vil nevne noen at disse områdene først. Videre er det flere spennende muligheter som også bør undersøkes nøyere.

For det første må implementasjonen av aksessmetoden utvides til å omfatte operasjoner som vil kunne føre til inkonsistens mellom indeksen og databasen. Dette er operasjoner som nå ikke går om aksessmetodegrensesnittet, slik som DROP TABLE, DROP INDEX.

For å klare dette vil man måtte gjøre endringer i måten PostgreSQL bruker grensesnittet på. Videre må en COMMIT-operasjon etter innsetting av et nytt dokument i den eksterne indeksen vente til alle dokumenter er ferdig indeksert før den returnerer. Dette vil kreve endringer i grensenittet mot FDS ved at den venter med å returnere til indekseringen av et dokument er ferdig. En bedre løsning er imidlertid den jeg foreslår i 7.2.4, slik at flere dokumenter kan settes inn i parallell. Dette krever at FDS implementerer transaksjonshåndtering.

Transaksjonssikre-operasjoner mot den eksterne indeksen bør være tilgjengelig. Transaksjonsmodellene jeg har foreslått i 7.2 og problemene rundt disse må testes, verifiseres og bedres. De foreslåtte metodene er en begynnelse på dette, men det vil kreve mer arbeid for å påvise at det fungerer i praksis.

En utvidelse av grensesnittet er også nødvendig hvis man effektivt skal tillate søk i indekser med flere kolonner, eller utnytte informasjon i flere indekser for å besvare en spørring i den eksterne indeksen. Et klart anvendelsesområde for et integrert system er å gi eksisterende databaser, som ikke klarer å skalere godt nok, såkalte ”dust ducks”, nytt liv ved å indeksere teksten med en søkemotor. For at dette skal kunne gjøres best og enklest mulig, bør man kunne indeksere hele tabeller, eller hele databasen, i *en* indeks.

Effektiviteten til den eksterne indeksen kan utnyttes ytteligere hvis man optimaliserer søk som henter ut de N beste resulatetene, fordi FDS er optimalisert for nettopp dette. Slik det er nå, utnyttes ikke dette.

For å teste effektiviteten og skalerbarheten til den eksterne indeksen, bør det gjennomføres tester i et flernodeoppsett. Testene bør også gjennomføres på et ”real life”-datasett for virkelig å kartlegge nytten av integrasjonen, helst på en eksisterende databaseapplikasjon som har behov for å bedre fulltekstsøkoperasjoner.

Hvis en integrasjon med FDS som ekstern indeks skal ha kommersiell interesse bør den også integreres i andre databaser, slik som Oracle, DB2 og MS SQL Server. Måten integrasjonen har vært gjort på i denne oppgaven, bygger på hvordan PostgreSQL er bygd opp, og det er ikke sikkert at denne typen integrasjon vil være mulig i andre databaser.

Det er flere områder jeg har vært innom i oppgaven som det ville vært interessant å jobbe videre med ut i fra muligheter som ligger i et integrert system. Et av disse er SQL/MM, der det hittil ikke finnes implementasjoner av standarden. På grunn av egenskaper i FDS vil det som nevnt i 7.4.2 være mulig å implementere et subsett av funksjonaliteten i standarden. Det opplagt beste, hvis man ønsker å kunne bruke FDS mot andre databaser enn PostgreSQL, er å implementere dette som et alternativt spørrespråk i FDS.

Referanser

[1] Carl Shapiro and Hal R.Varian, *The Information Economy*. In *Information Rules*. pp. 1-18, Harvard Business School Press, 1999.

[2] Samuel DeFazio, Amjad Daoud, Lisa Ann Smith and Joe Hellerstein. Integrating IR and RDBMS Using Cooperative Indexing. 84-92. 1995. ACM SIGIR, ACM Press.

[3] Jim Melton and Andrew Eisenberg. SQL Multimedia and Application Packages (SQL/MM). 30[4], 97-102. 2001. ACM SIGMOD, ACM Press.

[4] Gerald J.Kowalski and Mark T.Maybury, Information Storage and Retivial Systems, Kluwer Academic Publisher, 2000.

[5] Robert G.Crawford, *The Relational Model in Information Retriveal*. Journal of the American Society for Information Science *32,* 51-64 (1981).

[6] Norbert Fuhr and Thomas Rõlleke, *A Probabilistic Relational Algebra for the Integration of Information Retrieval and Database Systems*. ACM Transactions on Information Systems *15,* 32-66 (1997).

[7] P. Aoki. Implementation of Extended Indexes in POSTGRES. 25[1], 2-9. 1991. ACM SIGIR Forum, ACM Press.

[8] Ian A.Macleod, *Text Retrieval and the Relational Model*. Journal of the American Society for Information Science *1991,* 155-165 (1991).

[9] Mohan Kamath and Krithi Ramamriham. Efficient Transaction Support for Dynamic Information Retrivel Systems. 19, 147-155. 1996. ACM SIGIR, ACM Press.

[10] M. Stonebraker, *Document Processing in a Relational Database System*. ACM Transations on Office Information Systems *1,* 143-158 (1983).

[11] David A.Grossman, David O.Holmes and Ophir Frieder. A Parallel DBMS Approach to IR in TREC-3. 1994.

[12] Oracle. Oracle Text, White paper. 2002.

[13] Microsoft. Textual Search on Database Data using Microsoft SQL-Server 7. 1999.

[14] IBM. Text Information Extender Administration and User's Guide. 2001.

[15] Oleg Bartunov and Teodor Sigaev. Tsearch2. 2003.

[16] Omar Alonso. Oracle Text Technical White Paper. 2002. Oracle.

[17] Microsoft Corporation. Textual Search on Database Data using Microsoft SQL-Server 7.0. 1999. MSDN.

[18] International Organization for Standardization. ISO/IEC 13249-2 Information technology - Database languages - SQL Multimedia and Application Packages - Part 2: Full-Text. 2000.

[19] Joe Hellerstein. The GiST Indexing Project. 1999.

[20] O. Bartunov. OpenFTS Primer. 2002.

[21] J.Gray and A.Rauter, Transaction Processing: Concepts and Techniques, Morgan Kaufmann, 1993.

[22] Tedd Codd, *A relational modell for large shared data banks*. Communications of the ACM *13,* (1970).

[23] Hector Garcia-Molina, Jeffery Ullman and Jennifer Widom, Database Systems - The complete book, Prentice Hall, 2002.

[24] PostgreSQL. 2003.

[25] M. Stonebraker, Lawrence A Rowe and Michael Hirohama, *The implementaion of POSTGRES*. IEEE Transactions on Knowledge and Data Engineering *2,* 125-142 (1990).

[26] M. Stonebraker. Inclusion of new types in releational database systems. 1986. IEEE.

[27] M. Stonebraker and Lawrence A Rowe, *The Design of Postgres*. ACM SIGMOD Record *15,* 340-355 (1986).

[28] M. Stonebraker. Inclusion of new types in relational database systems. 262-269. 1986. International Conferencs on Data Engineering, IEEE.

[29] Robert R.Korfhage, Information Storage and Retrivial, Wily Computer Publishing, 1997.

[30] Rochard K.Belew, Finding Out About, Cambrige University Press, 2000.

[31] Arvind Arasu, Andread Paepcke, Hector Garcia-Molina, Junghoo Cho and Sriram Raghavan, *Searching the web*. ACM Transactions on Internet Technology (TOIT) *1,* 2-43 (2000).

[32] Fast Search & Transfer ASA. Under the hood of FAST Enterprise Search Solution. DS001. 2002.

[33] Knut Magne Risvik, Peter Boros and Tomasz Mikolajewski. Query Segmentation for Web Search. 20-5-2003. Poster on ACM WWW Conference 2003.

[34] Martin Fowler, UML Distilled, Addison Wesley, 2000.

[35] Jim Gray, The Benchmark Handbook For Database and Transaction Processing Systems, Morgan Kaufman, 1993.

[36] David DeWitt and Jim Gray, *Prarallel Database Systems: The future of High Performance Database Systems*. Communications of the ACM *35,* 85-98 (1992).

[37] Tom Lane and Bruce Momjian. Transaction Processing in PostgreSQL. 30-10-2000. 30-10-2000.

[38] PostgreSQL Global Development Group. PostgreSQL 7.3 Administrator's Guide. 2003.