NTNU

Det skapende universitet

Apache Derby som MMDB

Knut Magne Solem

Master i datateknikk

Oppgaven levert: Juni 2007

Hovedveileder: Svein Erik Bratsberg, IDI

Norges teknisk-naturvitenskapelige universitet

Institutt for datateknikk og informasjonsvitenskap

Oppgavetekst

Implementer en minnebasert lagermodul ved å erstatte Storage-laget i Derby. Derby sin modulære struktur skal gjøre dette overkommelig. Sammenlign ytelsen på denne løsningen med standard Derby. Eksperimenter med å legge til persistering, krasjrecovery, sjekkpunkting og transaksjonslogging til disk samtidig som databasen bare gjør endringer i minnet.

Oppgaven gitt: 20. januar 2007
Hovedveileder: Svein Erik Bratsberg, IDI

Sammendrag

Apache Derby er en Open Source-database utviklet i Java. Den er designet som en tradisjonell diskbasert database og er optimalisert for diskaksess. Målet med denne oppgaven er å finne måter å øke ytelsen på databaser der vi antar at hele databasen får plass i minnet. Vi ønsker å optimalisere for aksessering av data i minnet istedenfor på disken og på den måten gjøre databasen mer lik MMDB[1]-databaser. Dette har vi gjort ved å identifisere og erstatte flere disk-spesifikke moduler i Derby med nye moduler optimalisert for minneaksessering. Samtidig bruker vi eksisterende moduler for å oppnå persistens av dataene. Resultatene viser at ytelsen på leseaksesser i siste stabile versjon av Derby (10.2.2.0) kan økes med 70-200% og skriveaksesser med 20-100% avhengig av hvor mange klienter som benytter databasen samtidig.

[1] Main Memory database - En database der data ligger primært i minnet

Forord

Denne oppgaven er resultatet av faget "TDT4900 - Datateknikk og informasjonsvitenskap, masteroppgave" ved institutt for datateknikk og infomasjonsvitenskap ved Norges teknisk-naturvitenskapelige universitet. Prosjektet er utført som en følge av en oppgave gitt av professor Svein Erik Bratsberg og er en videreføring av prosjektarbeidet vi gjorde høsten 2006 i faget "TDT4740 Databaseteknikk og distribuerte systemer, fordypningsemne".

Den originale oppgaveteksen er:

> Build a working in-memory DBMS, by replacing the Derby storage layer with an in-memory byte array. Derby's modular structure should make this easy to do. Compare the performance of this solution to standard Derby. Experiment with adding persistence and crash recovery, retaining checkpoints and transactional logging to disk, while doing database page updates only in memory.

Oppgaven har gått ut på å forbedre ytelsen til Apache Derby når en database i sin helhet får plass og ligger i minnet. Apache Derby er designet som en tradisjonell DRDB[2]. Utfordringene har derfor bestått i å endre nåværende design til et som har flere likhetstrekk med en MMDB[3] uten å endre for mye av koden.

Jeg ønsker å takke min veileder Svein Erik Bratsberg ved NTNU i tillegg til Svein-Olaf Hvasshovd ved NTNU og Øystein Grøvlen ved Sun Microsystems for god veiledning og hjelp under implementering og design. I tillegg til konstruktive innspill med henblikk på struktur, omfang og innhold i arbeidet med rapporten.

Knut Magne Solem

[2]Disk resident database - En database der data ligger primært på disk
[3]Main Memory database - En database der data ligger primært i minnet

Innhold

Sammendrag i

Forord iii

Innholdsfortegnelse iii

Figurer x

Tabeller xi

Lister xiii

1 Introduksjon 1
1.1 Oppgaven 1
1.2 Prosjektmål 2
1.3 Bakgrunn 2

2 State of the art 5
2.1 Aksessmetoder 5
2.2 Skiplist 6
2.2.1 Balansert 7
2.2.2 Søking 8
2.2.3 Innsetting og sletting 9
2.2.4 Samtidighet 9
2.3 Recovery i DRDB 12
2.3.1 Logging 13
2.3.2 Sjekkpunkting 14
2.4 Recovery i main-memory databaser 16
2.4.1 Logging 16
2.4.2 Sjekkpunkting 17
2.4.3 Gjenoppretting 19
2.5 Tidligere arbeid 20

2.5.1 Starburst . . . 20
2.5.2 TimesTen . . . 20
2.5.3 MySQL Cluster . . . 21
2.5.4 Dali/Datablitz . . . 21
2.5.5 Monet . . . 22
2.5.6 Stephen Fitch sin MemoryStorageFactory . . . 23
2.5.7 Clustra . . . 24
2.5.8 C-store . . . 25

3 Derby **27**
3.1 Introduksjon til derby . . . 27
3.2 Moduler og monitor . . . 28
3.3 Låsing og samtidighet . . . 29
3.4 Caching . . . 30
3.5 Conglomerate . . . 31
3.6 Persistens . . . 32
3.7 MemStore . . . 32
3.7.1 MemHeap . . . 33
3.7.2 MemHash . . . 34

4 Design og implementasjon **35**
4.1 Ny datastruktur i MemStore . . . 35
4.2 Endring av eksisterende aksessmetode . . . 37
4.3 Ny aksessmetode . . . 38
4.3.1 MemSkiplist . . . 38
4.3.2 Implementering i Derby . . . 40
4.4 Logging . . . 41
4.5 Recovery i MemStore . . . 44
4.5.1 Transaksjonsrecovery i MemStore . . . 45
4.5.2 Sjekkpunkt . . . 45
4.5.3 Gjenoppretting ved krasj . . . 46
4.5.4 Implementering i Derby . . . 47

5 Tester og målinger **51**
5.1 Oppsett . . . 51
5.2 Testdata . . . 52
5.3 Fordeling av CPU-tid . . . 53
5.4 Aksessmetoder . . . 54
5.4.1 Skiplist . . . 54
5.4.2 MemSkiplist og MemHeap . . . 55
5.5 Sjekkpunkt-algoritmen . . . 61
5.6 Responstid . . . 63
5.7 Minneforbruk . . . 64

6 Konklusjon **69**

7 Videre arbeid **71**

Bibliografi **73**

Figurer

1.1 Utvikling av minneprisene, hentet fra `www.jcmit.com/mem2006.htm` . . . 3
1.2 Viser throughput for en TPC-B-basert belastning. Databasen er på 10MB mens bufferet er på 50MB. 3
1.3 Viser throughput for en TPC-B-basert belastning. Databasen er på 50MB mens bufferet er på 10MB. 4

2.1 Rebalansering av AVL-tre 7
2.2 En typisk skiplist-struktur 8
2.3 Søking i Skiplist med indekser på to nivåer 9
2.4 Innsetting av noder i Skiplist 10
2.5 Sletting av noder i Skiplist 11
2.6 Sletting av noder i Skiplist med markeringsnoder 11
2.7 Vertikal fragmentering av data som brukt i Monet 22

3.1 Arkitekturen til Derby 27
3.2 Ressursbruk i Derby [AD06] 30
3.3 Klassediagram av cache-tjenesten i Derby [Cer06] 31
3.4 Gamle MemStore sin lagringsstruktur 34

4.1 Ny lagringsstruktur i MemStore 36
4.2 Loggformat i Derby 42
4.3 Gjennoppretting av MemStore etter systemkrasj 47
4.4 Gjennoppretting av MemStore etter systemkrasj 48

5.1 Tabell i Wisconsin benchmark 52
5.2 Graf som viser hvordan CPU-tiden er fordelt i de ulike lagene i Derby ved innsetting 53
5.3 Graf som viser hvordan CPU-tiden er fordelt i de ulike lagene i Derby ved oppdateringer 54
5.4 Graf som viser hvordan CPU-tiden er fordelt i de ulike lagene i Derby ved søk på primærnøkkel 55

5.5 Ytelsestester for de ulike implementaringene av Skiplist kjørt i ulike versjoner av Java Virtual Machine 56
5.6 Søking på primærnøkkel i Wisconsin-testdata mot Derby i embedded-modus. 57
5.7 Søking på ikke-indeksert nøkkel i Wisconsin-testdata mot Derby i embedded-modus. 58
5.8 Områdesøk på ikke-indeksert nøkkel i Wisconsin-testdata mot Derby i embedded-modus. 58
5.9 Wisconsin sin join-operasjon utført ved hashjoin i MemStore og RawStore i embedded-modus. 59
5.10 Update med skrive-cache deaktivert. 60
5.11 Update-operasjon mot Wisconsin-testdata i MemStore og RawStore i embedded-modus. Skrive-cache er aktivert. 60
5.12 *Insert* med skrive-cache deaktivert. 61
5.13 Innsetting av Wisconsin-testdata mot Derby i embedded-modus. med skrive-cache er aktivert. 62
5.14 Grafen viser hvordan ytelsen til databasen påvirkes av et sjekkpunkt. Starter sjekkpunktet etter ca. 70 sekunder, og det varer i ca 20 sekunder. 62
5.15 Grafen viser tiden det tar å utføre et sjekkpunkt og størrelsen det tar avhengig av antall rader. 63
5.16 Responstid for *select*, viser fordeling av responstidene ved hjelp av kvartiler. 64
5.17 Histogram over responstid for en klient som kjører kontinuerlige *select*-setninger mot primærnøkkel. 65
5.18 Histogram over responstid for 10 samtidige klienter som kjører kontinuerlige *select*-setninger mot primærnøkkel. 65
5.19 Minneforbruk i Derby ved innsetting av Wisconsin-data i RawStore 66
5.20 Minneforbruk i Derby ved innsetting av Wisconsin-data i MemStore . . . 66
5.21 Minneforbruk og bruk av *garbage collection* ved søking i RawStore 67
5.22 Minneforbruk og bruk av *garbage collection* ved søking i MemStore 67

Tabeller

5.1 Lagoppdelingen i Derby [AD06] 53

Listings

4.1 MemHeap.fetch(...), her blir data hentet ut fra et RowLocation-objekt som er hentet fra den sekundære indeksen MemSkiplist. 37
4.2 Bruk av CAS-instruksjon i ConcurrentSkiplistMap 40
4.3 Vår CAS-metode i MemSkiplist 40
4.4 Innsetting av rad i MemHeap, der posisjon returneres for oppdatering i indeks. 43
4.5 Skriving av MemInsertOperation i loggen 44

Kapittel 1
Introduksjon

Dette kapittelet skal gi en introduksjon til hva denne oppgava handler om. Vi vil her beskrive bakgrunnen og målene for oppgaven, og hvilke problemer vi må løse for å oppnå disse.

1.1 Oppgaven

Gjennom prosjektet i faget *TDT4740 Databaseteknikk og distribuerte systemer* utarbeidet vi en prototype av en modul som tillot primærlagring av databaser i minnet kalt MemStore [Sol06]. Denne gav lovende resultater, men vi påpekte også flere mangler ved denne. Vi ønsker i dette prosjektet å utvide og gi ny funksjonalitet til MemStore.

Manglene er listet opp her:

Persistens - Implementasjonen skrev bare en *dummy*-logg for å kunne sammenligne ytelsen med eksisterende Derby-implementasjon. Dette betyr at verken checkpointing, recovery eller commit/rollback fungerer.

Indeksmetode - MemStore har bare en en hash-basert indeks som bare støtter direkte oppslag på primærnøkkel. Det er ønskelig å ha en indeksmetode som støtter sekvensielle søk.

Sletting - MemStore har ikke støtte for sletting av rader.

DDL - Den eneste DDL-spørringa som støttes er CREATE TABLE.

Låsing - Det blir brukt rad-basert låsing ved INSERT men ikke ved SELECT. Låseeskalering fungerer heller ikke.

Join - Nåværende indeksmetode støtter ikke *join* av flere tabeller.

Både MemStore og RawStore - Det er bare mulig å bruke enten MemStore eller RawStore i nåværende implementasjon, det bør være mulig å bruke tabeller basert på både MemStore og RawStore i samme database.

Den opprinnelige oppgaveteksten var:

> Build a working in-memory DBMS, by replacing the Derby storage layer with an in-memory byte array. Derby's modular structure should make this easy to do. Compare the performance of this solution to standard Derby. Experiment with adding persistence and crash recovery, retaining checkpoints and transactional logging to disk, while doing database page updates only in memory.

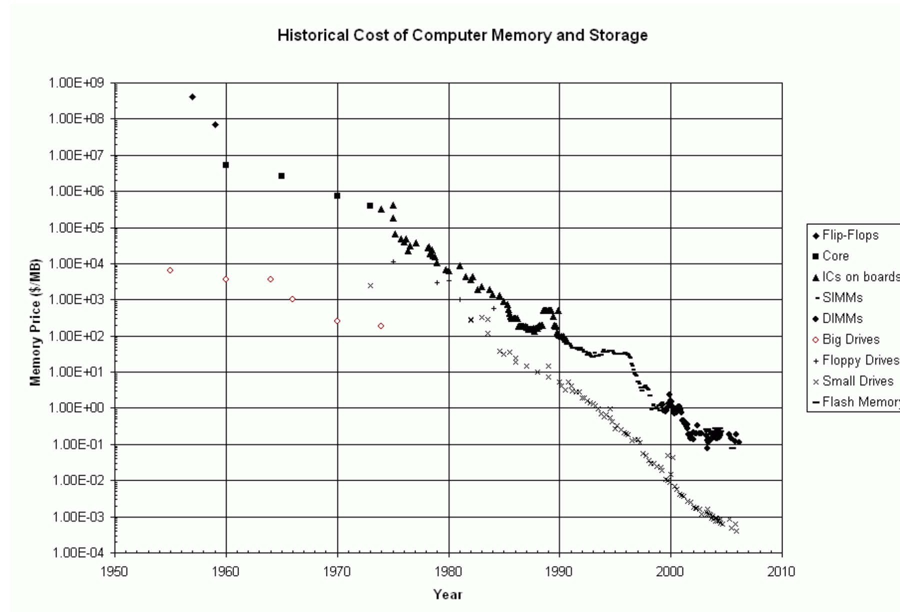
1.2 Prosjektmål

Vi vil med prosjektet fortsette utviklingen av MemStore for Derby. Hovedfokus vil være å eksperimentere med persistens og utvikling av en ny indeksmetode. Målet er at vi skal kunne gi MemStore grunnleggende databasefunksjonalitet og samtidig gi en høyere ytelse enn RawStore.

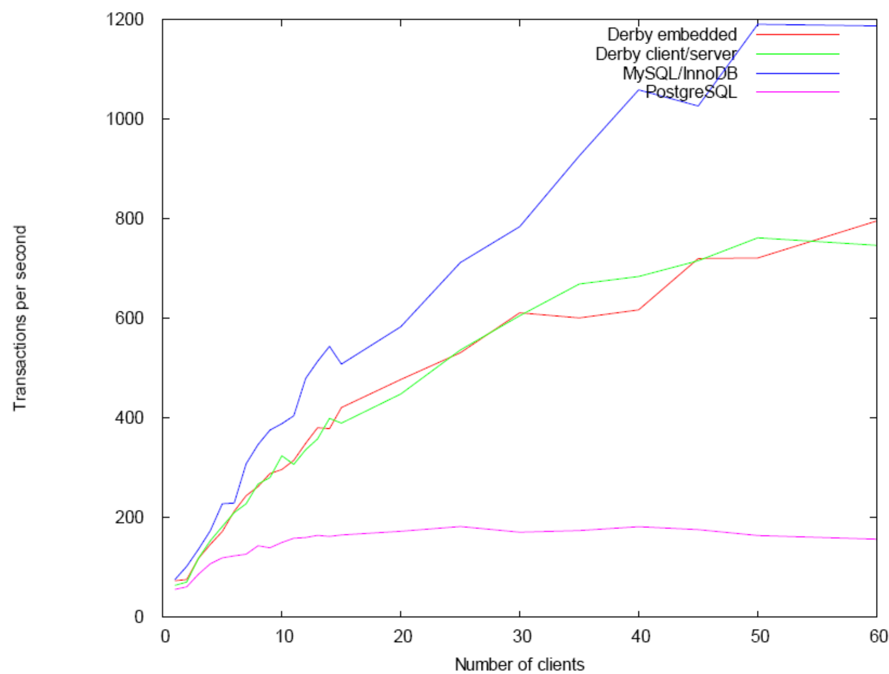
1.3 Bakgrunn

Det er flere grunner for ønsket om å utvide Derby med en egen minnebasert lagringsmodul. De siste tiårene har minneprisen som vi ser i figur 1.1 bare gått nedover og er ikke lengre en like kostbar ressurs. Den tradisjonelle måte å utnytte dette minnet i databasesystemer har vært å øke bufferstørrelsen [GMS92]. Dette er veldig enkelt og gjøre, og vil øke ytelsen en hel del. Likevel så blir ikke minnet brukt så effektivt som det kunne vært brukt. Ved å designe et databasesystem med minneresidente data og optimalisert for minneaksesser vil man bruke minnet mer effektivt noe som vil øke ytelsen enda mer.

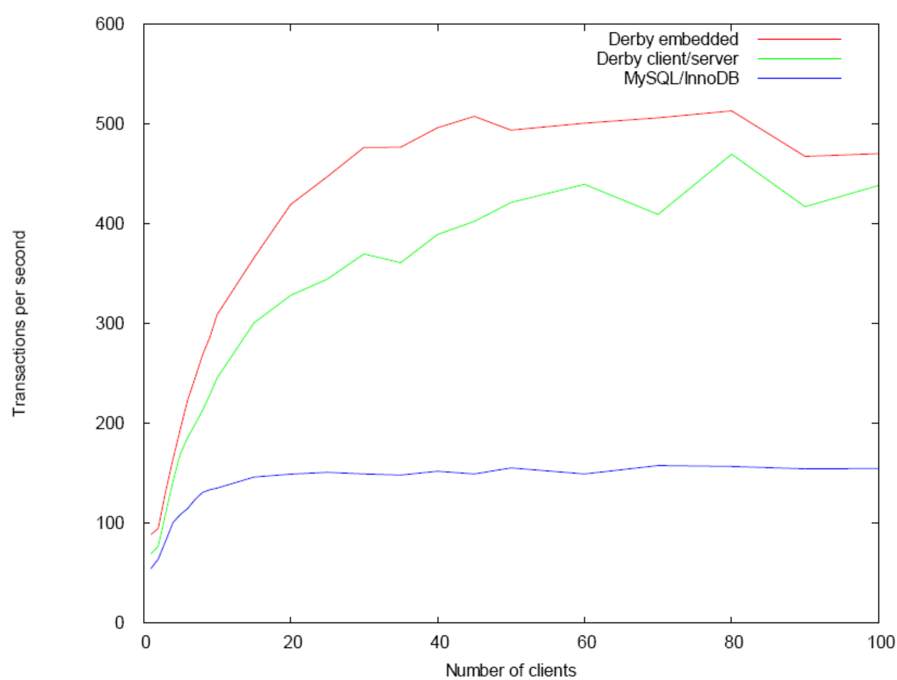
Derby har tidligere vist seg å ha en dårligere ytelse på små databaser som i sin helhet passer i minnet i forhold til andre databasesystemer. Bernt Mariues Johansen holdt et foredrag på JavaZone i 2005 [Joh05] der han vise grafene i figur 1.2 og 1.3. De viser at Derby har en bra ytelse for store databaser, men en dårligere ytelse sammenlignet med MySQL og PostgreSQL når databasen er liten og får plass i minnet. Siden Derby er en typisk embedded-database som ofte inneholder begrensede mengder data viser dette at det gjenstår endel arbeid for å optimalisere Derby for dette.



Figur 1.1: Utvikling av minneprisene, hentet fra `www.jcmit.com/mem2006.htm`



Figur 1.2: Viser throughput for en TPC-B-basert belastning. Databasen er på 10MB mens bufferet er på 50MB.



Figur 1.3: Viser throughput for en TPC-B-basert belastning. Databasen er på 50MB mens bufferet er på 10MB.

Kapittel 2
State of the art

Vi vil her gi en kort introduksjon til fundamentale teknikker og problemer relatert til design og implementering av minnebaserte databaser. Vi tar for oss aksessmetoder i databaser, metoder for gjennoppretting av databaser og ser til slutt på på tidligere arbeid.

2.1 Aksessmetoder

En database som inneholder store mengder data uten å effektivt kunne hente de frem igjen har begrenset verdi. Det brukes derfor aksessmetoder som sørger for "snarveier" til ønskede data for å slippe å søke sekvensielt i dataene. Disse hjelpestrukturene kan man grovt dele inn i to kategorier: indekssekvensielle og ikke-indekssekvensielle aksessmetoder.

Indekssekvensielle aksessmetoder er en datastruktur som er sortert, og den kan derfor effektivt støtte områdesøk. Eksempler på slike aksessmetoder er B-trær, AVL-trær og T-trær. Aksessmetoder som ikke er indekssekvensielle gir kun direkte oppslag til dataelementer i databasen, dette er typisk for hash-baserte aksessmetoder [LC86]. Vi vil her gå igjennom de mest kjente aksessmetodene som vi tok for oss i [Sol06].

AVL-Trær ble opprinnelig utviklet med tanke på bruk i main-memory-systemer. Veldig rask men utnytter lagringsplassen dårlig fordi den bare lagrer et dataelement per node.

B-trær Det finnes ulike typer B-trær, og det konkluderes med at B+-trær er best egnet for disk, mens et vanlig B-tre er å foretrekke for et MMDB-system [LC87]. Grunnen til dette er at et B+-tre lagrer all data i bunnen av treet, noe som er fordelaktig for disk siden tilfeldig lesing er veldig kostbart. For minnebruk er ikke dette like fordelaktig fordi tilfeldig lesing ikke er vesentlig dyrere enn sekvensiell lesing.

Chained Bucket Hashing er en statisk indeksstruktur brukt i både minne og disk. Den er meget rask siden den er en statisk struktur, og trenger derfor ikke å reorganisere dataene. Ulempen er også at den er så statisk, man må vite eller i det minste anslå hvor mange hash-bøtter man trenger, hvis anslaget er for lite blir ytelsen dårlig og om det er for stort blir sløsing med plass et problem. Siden dette er en hash-basert indeksmetode støtter den ikke effektivt sekvensiell lesing av data i sortert rekkefølge.

Extendible Hashing er en dynamisk hashtabell som øker størrelsen etterhvert som data legges til. Den dobler antall noder når en node blir full, og er derfor avhengig av å ha en hashfunksjon som er velbalansert. Støtter ikke sekvensiell søking fordi den baserer seg på hashtabeller.

Linear Hashing er også en dynamisk hashtabell, men denne øker bare nodeantallet lineært ved nodesplitting. Denne støtter ikke sekvensiell søking.

Modified Linear Hashing er en modifisert versjon av *Linear Hashing* som er mer orientert mot MMDB-systemer. Denne støtter heller ikke sekvensiell søking siden det er en hashbasert datastruktur.

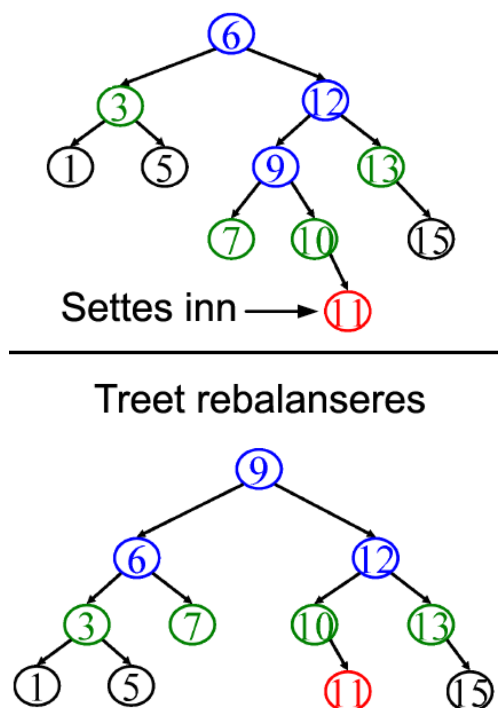
T-trær er en ny type binærtre som er utviklet fra AVL- og B-trær og støtter derfor sekvensiell søking. Den tar konsekvensen av den dårlige lagerutnyttelsen som AVL-trær tilbyr ved å lagre flere dataelementer i hver node.

Vi foreslo også å se nærmere på en datastruktur kalt Skiplist. Den ble utviklet så sent som i 1990 av William Pugh [Pug90b], og videreutviklet i Håkan Sundell sin PhD [Sun04] der han kom frem til en blokkeringsfri Skiplist implementasjon. Den nye algoritmen til Sundell er utviklet med fokus på samtidighet og bruk av vanlig hardware. Den kan derfor egne seg bra i databasesystemer med flere CPU'er. Antagelsen til blokkeringsfrie algoritmer er at kollisjoner skjer sjeldent og bør behandles som et unntak. En kollisjon skjer når en tråd prøver å skrive til en post som allerede er under skriving. Det må derfor være mulig å oppdage når dette skjer. Om en kollisjon oppdages, kjører man et eller flere av stegene i operasjonen på nytt og i håp om at denne ikke kolliderer igjen. Denne algoritmen bruker en "CompareAndSwap"-operasjon for å sjekke om kollisjoner har oppstått, dette er en veldig rask atomisk operasjon som mange prosessorer tilbyr.

Vi kan ikke se at noen har implementert skiplist som indeks i databasesystemer. Skiplist er per definisjon en *dictionary*, det vil si at den knytter dataverdier opp mot tilhørende nøkkelverdier [MS01]. Den kan derfor uten modifikasjoner brukes som en indeks i databasesystemer.

2.2 Skiplist

Skiplist [Pug90b] er en probabilistisk listebasert datastruktur som er beregnet for bruk i minne. Den er både enkelt oppbygd, effektiv og kan erstatte tradisjonelle balanserte tre-strukturer. Målet er å ha en mest mulig balansert struktur til enhver tid, uavhengig av hvilke typer data som settes inn. Vi har valgt å bruke denne datastrukturen som basis for en ny aksessmetode til MemStore.



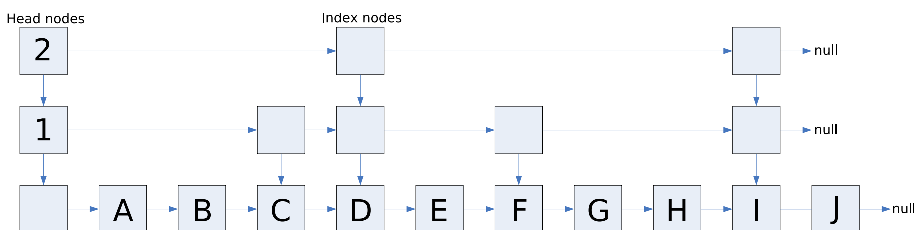
Figur 2.1: Rebalansering av AVL-tre

2.2.1 Balansert

Enkle tre-strukturer som f.eks binær-tre fungerer gjerne fint på tilfeldige data, men datastrukturen kan fort bli ubalansert når det f.eks settes inn delvis sorterte sekvenser av tall. Balanserte trær må derfor kjøre rebalanseringsalgoritmer for å unngå en ubalansert struktur som er kostbar å søke i. Kostnaden for å søke i treet er direkte knyttet til høyden i treet [BGMS97]. Balanseringsalgoritmer blir derfor kjørt når det oppdages at en sletting eller innsetting medfører at treet er ubalansert. Dette er illustrert i figur 2.1 der tallet 11 blir satt inn i et AVL-tre. Innsettingen medfører at treet blir ubalansert, som igjen utløser en rebalanseringsalgoritme. Som vi ser av figuren blir de fleste tallene byttet om, på store trær vil dette derfor medføre mye prosessering. På grunn av at store deler av treet blir omorganisert er det vanskelig å tillate samtidig tilgang. Lehman og Kung skisserte en tre-algoritme med en rebalanseringsalgoritme som tillot samtidig tilgang ved å kopiere ut deler av treet, rebalansere del-treet og etterpå koble det atomisk inn i det opprinnelige treet [KL80]. I systemer der sanntidsytelse er viktig er det uansett en uheldig egenskap å måtte rebalansere treet under aktivitet. Kuo *et al* har i [KWL99] foreslått en rebalanseringsalgoritme for sanntids-systemer som består av små transaksjoner med lav prioritet samtidig som det tillates et noe ubalansert tre, dette begrenser problemet endel, men eliminerer det ikke.

Skiplist bruker en probabilistisk framgangsmåte for å oppnå det samme. Den bruker en pseudo-tilfeldig tallgenerator til å bestemme hvordan data skal fordele seg i strukturen. Datastrukturen er derfor helt uavhengig av hvilke verdier som kommer inn. Like sekvenser av tall vil av denne grunn settes inn annerledes fra gang til gang. Det vil

derfor være lite sannsynlig at datastrukturen vil være veldig ubalansert. Å bruke en probabilistisk modell for å opprettholde en balansert struktur medfører også en enklere implementasjon enn å kjøre rebalanseringsalgoritmer når treet er ubalansert [Pug90b]. Datastrukturen som ligger i bunnen er en sortert enkeltlenket liste. I en enkeltlenket



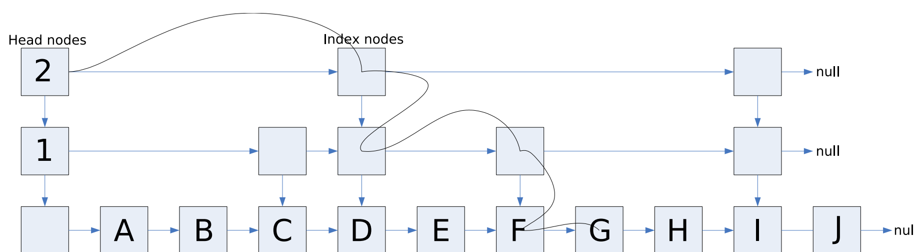
Figur 2.2: En typisk skiplist-struktur

liste har hver node en peker til neste node i rekken, dette i motsetning til dobbeltlenkede lister der hver node i tillegg har en peker til forrige node. Skiplist baserer seg på at man kan ha flere nivåer med indeksnoder over den lenkede lista med nøkler. En typisk skiplist vil se ut som i figur 2.2 og bestå av $L = log_{1/p}\ n$ nivåer. Her er p en konstant som settes til hvor stor andel av nodene på et nivå skal ha indeksnoder i nivået over og n er antall noder. Dess høyere p settes dess mindre er sannsynligheten for at en søking i treet tar vesentlig lengre tid enn forventet. Den ble f.eks satt til 1/2 i introduksjonen til Pugh [Pug90b] noe som gav en sannsynlighet på $\frac{1}{200000000}$ for at et søk skulle ta tre ganger lengre tid enn forventet.

2.2.2 Søking

Søking etter en nøkkel begynner på *head node* på øverste nivå, og fortsetter horisontalt langs indeksnodene så lenge de har en nøkkel som er mindre enn den det søkes etter. Når søket kommer til en indeksnode som er større, vil søket fortsette på et nivå lengre ned. Tilslutt havner søket på nederste nivå som er en tradisjonell lenket liste. Et typisk søk i Skiplist er illustrert i figur 2.3. Hver indeksnode har en minnereferanse til sin datanode på nederste nivå og denne brukes når nodens nøkkel sammenlignes med den det søkes etter.

Indeksnodene gjør at søket kan hoppe lengre frem enn bare den neste noden i den lenkede lista. Hvis hver node kan se to noder fremover er det i verste fall nok å søke igjennom $\lceil n/2 \rceil + 1$ noder, der n er antall noder i lista. Om man i tillegg gir hver fjerde node muligheten til å se 4 noder fremover trenger man bare traversere $\lceil n/4 \rceil + 2$ noder i værste fall. Gir man hver 2^i. node muligheten til å se 2^i noder fremover vil derfor søkekompleksiteten bli $\lceil log_2 n \rceil$. Her samsvarer i med antall nivåer i strukturen. Denne fremgangsmåten kan brukes på en statisk struktur, men i en dynamisk datastruktur der noder vil komme og falle fra vil dette bli umulig å opprettholde. Skiplist bruker derfor en pseudotilfeldig tallgenerator som distribuerer indeksene slik at 50% av nodene skal ha en indeks på nivå 1, 25% på nivå 2, 12.5% på nivå 3 osb. En typisk struktur kan da se ut som i figur 2.2.



Figur 2.3: Søking i Skiplist med indekser på to nivåer

2.2.3 Innsetting og sletting

Ved innsetting søkes det først som forklart i kapittel 2.2.2 helt til man finner riktig posisjonen i den lenkede listen. Når riktig posisjon er funnet, konsulteres den pseudotilfeldige tallgeneratoren som bestemmer hvor "høy" eller hvor mange nivåer indeksen skal være. Antall nivåer blir kalkulert slik at sannsynligheten for at strukturen skal bestå av i nivåer er p^i. Maksimalt antall nivå blir begrenset til $L = log_{1/p}\ n$. Selve innsettinga på hvert nivå innebærer å opprette noden og endre "neste" pekeren på noden som befinner seg like før i den lenkede lista. Det er foreslått at indeksnodene som må endres mellomlagres under det første søket slik at man slipper å søke etter de på nytt. Ikke alle implementasjoner gjør dette, implementasjonene gjort av Fraser [Fra04], Fomitchev [Fom03] og Fomitchev og Ruppert [FR04] mellomlagrer ikke indeksnodene, og må derfor gjøre et nytt søk når indeksnoder skal opprettes. Det samme gjelder `ConcurrentSkipListMap` som er implementert i Java 6.0.

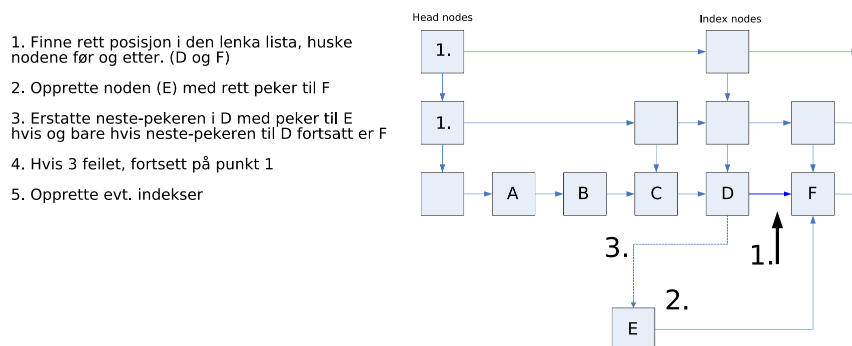
Sletting foregår på samme måte, men istedenfor å opprette nye indeksnoder slettes de. Det sjekkes samtidig om totalt antall nivåer med indekser kan reduseres.

2.2.4 Samtidighet

For å tillate samtidig tilgang til datastrukturen uten uheldige resultat utviklet Pugh også en variant av algoritmen som tillot flere prosesser eller tråder å samtidig oppdatere strukturen [Pug90a]. Denne algoritmen bruker kun skrivelåser og ikke eksklusive låser eller leselåser som blir brukt i mange trebaserte algoritmer. Disse ikke-eksklusive låsene blokkerer bare skrive-operasjoner, men det forutsettes her at en pekere og *integer*-verdier leses atomisk. Om denne forutsetningen ikke holder må man bruke leselåser i tillegg.

Det ble utført ytelsestester som konkluderte med at ytelsen var tilnærmet lik optimaliserte og ikke-rekursive implementasjoner av trealgoritmene AVL-tre, 2-3-tre og splay-tre [Pug90b]. Kostnadene ved *insert* og *delete* var noe lavere i Skiplist. Det ble også sammenlignet med rekursive implementasjoner av de ulike trealgoritmene, disse er vesentlig enklere å implementere enn de ikke-rekursive algoritmene. Resultatet viste da at Skiplist var 2-3 ganger raskere enn disse.

I senere tid har det blitt utvikler flere låsefrie algoritmer som er basert på skiplist. Låsefrie algoritmer er kjent for å være upraktiske på grunn av høy kompleksitet [Sun04], men i 2003 presenterte Sundell en låsefri algoritme basert på Skiplist som gav bedre ytelse enn eksisterende låsefrie algoritmer samtidig som enkelheten til Skiplist ble bevart [Sun04].



Figur 2.4: Innsetting av noder i Skiplist

Algoritmen til Sundell tillater flere prosessorer eller tråder å oppdatere en instans av skiplist uten noen form for låsing. Istenfor låser bruker han Compare-and-swap primitivet (CAS) som er tilgjengelig på de fleste pc-platformer. CAS er en instruksjon som prosessoren utfører atomisk, den sammenligner en gitt verdi med en verdi i minnet og endrer denne verdien til en gitt ny verdi om de er like. Er verdiene ulike endres ikke verdien i minnet, og instruksjonen returnerer med et negativt svar.

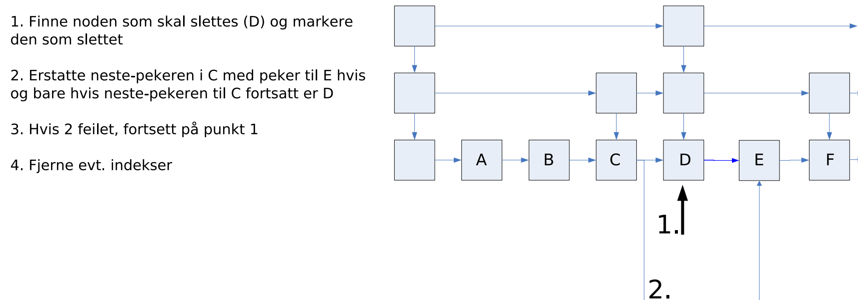
Innsetting av en ny node består av å lage noden med rett "neste"-peker, og utføre en CAS-instruksjon for å endre "neste" pekeren til noden til venstre for den nye noden. I figur 2.4 er innsetting av noden E illustrert, der blir neste-pekeren i D endret ved hjelp av CAS-instruksjonen.

En enkel implementasjon av sletting av en node som illustrert i 2.5 viser at neste-pekeren i C blir CAS-et til E. Dette medfører at D blir lenket ut av den horisontale lenkede listen. Dersom den ikke har flere referanser fra eventuelle indeksnoder vil noden bli fjernet av *garbage collection* eller en lignende funksjonalitet. Denne metoden har imidlertid et problem. Strukturen tillater samtidig sletting og innsetting uten noen form for synkronisering, dersom E innsettes samtidig som D slettes blir også node E slettet fordi D er den eneste som peker til den. Dette kan løses på flere måter, en måte er å bruke CAS2-instruksjonen[1] som atomisk kan sjekke to verdier før den utfører endringen men denne instruksjonen er det mange plattformer som ikke tilbyr. En annen måte er å legge inn spesielle hjelpenoder mellom hver normal node [Val95] som sørger for at punkt 3 (figur 2.4) under innsetting av en ny node feiler om noden før er under sletting. En tredje metode introdusert av Harris [Har01] er å markere pekere fra en node som er under sletting. Dette kan gjøres ved å utnytte en egenskap 32-bits pekere har, de må nemlig være delelig på 4. De to første bitene er derfor alltid satt til 0 og blir heller ikke sjekket ved vanlig bruk, men en endring vil medføre at en CAS-instruksjon feiler. Dette kan utnyttes til å hindre at en innsetting av en node vil lykkes om en sletting av noden foran har begynt. Ved sletting av en node er derfor det første som skjer å markere neste-pekeren.

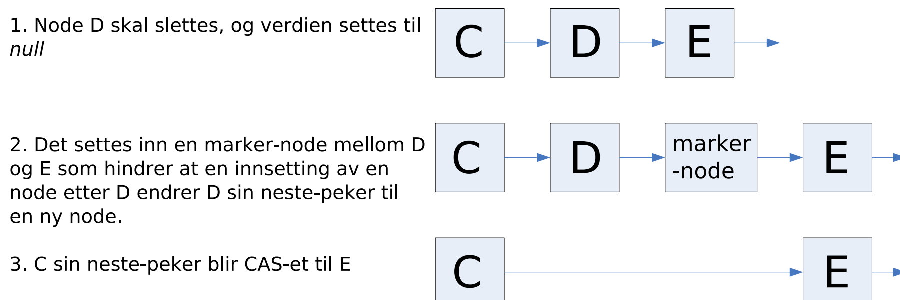
Doug Lea[2] [SM07] har implementert skiplist i Java 6.0 og bruker en kombinasjon av de to

[1]CAS2 støttes bare av et utvalg arkitekturer, den finnes bla. i den nyere IA-32 [Int04] men ikke i den gamlere x86-arkitekturen [HH01].

[2]med hjelp fra medlemmer av JCP JSR-166 Expert Group



Figur 2.5: Sletting av noder i Skiplist



Figur 2.6: Sletting av noder i Skiplist med markeringsnoder

metodene. Fra Java 5.0 har det vært mulig å bruke klassen `AtomicMarkableReference` for å markere minnereferanser, men Lea hevder at dette hadde medført en både mer plasskrevende og lite effektiv datastruktur. Istedenfor å bruke `AtomicMarkableReference` setter han inn en spesiell hjelpenode etter noder som skal slettes, og på denne måten hindre at CAS-instruksjonen til en evt. innsetting av en etterfølgende node skal lykkes. I tillegg til dette settes verdien til noden til *null*. Dette er illustrert i figur 2.6. Det eneste spesielle med hjelpenoden er at dataverdien til noden peker på seg selv, hver gang et søk traverserer en node må dette derfor sjekkes. Siden Java kjører regelmessige *garbage collection* vil slike ut-lenkede noder fort bli ryddet bort slik at algoritmen ikke trenger å gjøre annet enn å fjerne referansen. Dette er en av grunnen til at denne måten er mer effektiv enn å bruke `AtomicMarkableReference`.

Disse låsefrie algoritmene skal fungere like bra i et multitrådet miljø som i et multiprosessor miljø, alle operasjonen må derfor kunne stoppe når som helst uten å sette datastrukturen i en ikke-konsistent tilstand. En teknikk som er mye brukt her er "hjelpende kode" (eng: helping-out code). Det vil si at om en tråd har begynt å slette en node, men blir avbrutt før den har fullført, så vil en annen tråd som bare traverserer strukturen eller setter inn en ny node fortsette den påbegynte slettinga. På denne måten vil en operasjon aldri blokkere en annen operasjon. Dette er derfor en blokkeringsfri algoritme. Doug Lea bruker denne teknikken i implementasjonen av `ConcurrentSkiplistMap`.

Når en node slettes må eventuelle indekser på høyere nivåer også fjernes. Dette er en utfordring å få til i et multitrådet miljø uten noen form for samtidighetskontroll. Doug Lea løste dette ved å bruke hjelpende kode også under søking blant indeksnodene. Om

søket kommer over en indeksnode som har en verdinode med verdi lik null, indikerer dette at noden har blitt slettet eller er under sletting. Istedenfor å bare hoppe videre vil den hjelpe slettingen av indeksnoden ved hjelp av CAS-instruksjonen. Dette vil medføre noe dårligere responstid på transaksjoner som prøver å "hjelpe til" under indekssøk, men det vil ikke medføre blokkering som ville ha ført til en enda større forsinkelse.. Ved å unngå å bruke noen form for låsing slipper vi problemet med at låser på indekser på toppnivå fører til blokkeringer av alle samtidige transaksjoner. Dette er et kjent problem med minneresidente B-trær med høy samtidighet [Nie04].

2.3 Recovery i DRDB

Recovery i en database vil si å til enhver tid kunne gjennopprette databasen til en konsistent tilstand etter en transaksjonsfeil, systemfeil eller mediafeil. Med konsistent tilstand mener vi her transaksjonskonsistent. Det er mange måter å oppnå dette på, Hvasshovd presenterte fire måter å gjøre dette på i sin doktoravhandling [Hva92a]:

Blokk-orientert recovery er en gammel recovery-metode som forutsetter en DBMS[3] som bruker disk-blokker som fineste granularitet for logging og låsing. En ulempe er at loggvolumet er veldig høyt fordi før- og etterkopi av hele blokker blir brukt. Likevel tilbys det fleksible sjekkpunktingalgoritmer med akseptabel overhead ved normal transaksjonsaktivitet.

Post-orientert recovery eliminerer den største ulempen med den blokk-orienterte metoden ved at det her kan brukes post-logging isteden for en blokk-basert logging. Dette fører til en betydelig mindre disk I/O ved logging. Ved rollback av en transaksjon reverseres det som er gjort i en blokk ved å først låse den og etterpå reversere det som ble gjort i omvendt rekkefølge.

Kompensasjons-orientert recovery eliminerer den andre store ulempen ved de gamlere metodene ved å støtte radbasert låsing. Teknikkene som brukes her er latching av blokker mens en radbasert operasjon utføres, og logging av kompenserende operasjoner (CLRs[4]) for hver operasjon ved rollback. Fordelen av finere låsegranularitet medfører et noe høyere log-volum enn den post-orientert metoden og noe økt belastning ved gjennoppretting etter kræsj, men dette skjer så sjelden at i realiteten bare utgjør noen få prosent.

Tabell-orientert recovery er en ny teknikk som i sin helhet ikke blir brukt i noen kommersielle DBMS. Her introduseres en arkitektur der det er tabell-laget og ikke rad- og blokk-laget som sørger for samtidighetskontroll og recovery. I tillegg introduseres også en aksessmetodeserver i tabell-laget som mapper en rad med primary-key mot en post i en blokk. I tabell-laget låses radene på primary-key, mens aksessmetodeserveren sørger for å sette kortlivde låser på den tilhørende blokken. Recoveryserveren som også ligger i tabell-laget bruker en WAL[5]-basert logging, og logger data basert på primærnøkkel istedenfor poster. Denne fremgangsmåten tilbyr rad-basert logging og låsing i tillegg til blokkuavhengig *undo*-prosessering noe

[3]DataBase Management System
[4]CLR - Compensation log records
[5]Write-Ahead Logging

den kompensasjons-orienterte metoden ikke støtter. En største ulempen her er at *redo*-prosesseringen ikke kan gjøres blokkuavhengig, denne ulempen ble imidlertid utbedret i HypRa sin recovery-metode som baserer seg på denne teknikken [Hva92a].

Disse recovery-algoritmene fokuserer på å minimere disk I/O og samtidig påvirke samtidigheten i databasen minst mulig. Begge disse kravene er minst like viktig i en MMDB. For å optimalisere crash-recovery[6] lagres det jevnlig sjekkpunkter for å gjøre oppstart av databasen så rask som mulig. For en MMDB er dette en veldig viktig del, siden databasen primært ligger på et flyktig medium.

2.3.1 Logging

De forskjellige recovery-metodene har ulike krav til hvordan loggingen utføres. Blokk-orientert krever f.eks at før- og etterkopi av diskblokker som endres lagres i loggen, mens post-orientert trenger bare før- og etterkopi av posten som endres. Kompensasjons-orientert recovery krever i tillegg at det skrives CLR-poster ved hver *rollback*.

I tillegg til disse forskjellene kan logge-teknikker grovt deles inn i tre typer [GR93]:

Physical logging som også kalles *value logging* vil si å logge operasjoner på side-nivå. Den enkleste måte å gjøre dette på er å lagre før- og etterkopier av de endrede sidene, kopiene kan også gjenspeile bare endringene slik at det bare logges før- og etterkopi av de endrede delene av sidene. Fordelen med å gjøre det på denne måten er at UNDO og REDO og slike logg-poster er idempotent[7]. Et eksempel på en slik loggpost:

```
<pageupdate,filename=233,pagenum=12,offset=300,length=2000,
 old_value[length],new_value[length]>
```

En måte å redusere loggvolumet ved fysisk logging er å bare lagre endrede data. Man trenger da et nytt felt som forteller hvor de endrede dataene befinner seg, og hvor mye data det er. Dette blir kalt *partial logging*.

Logical logging , også kalt *operasjonslogging* vil si å logge selve operasjon som forårsaker side-operasjonen. Eksempler på dette er f.eks en INSERT-operasjon, den kan logges slik:

```
<insert operation,tablename=A,record value=r>
```

Når denne utføres vil dataene lagres i en side med ledig plass, indekser vil bli oppdatert, noe som igjen kan trigge f.eks flytting av data internt i en side og/eller en B-tree splitt. På denne måten kan én slik operasjon tilsvare flere ti-tals fysiske loggposter. UNDO av en INSERT-operasjon er rett og slett en DELTE-operasjon av posten som ble satt inn. Det er tydelig at dette vil spare store mengder logg-I/O i forhold til fysisk logging, men denne metoden har også svakheter. Det antas her at slike høynivå-operasjoner er atomiske, noe de ikke er. Om databasen krasjer midt i en slik operasjon, gjenspeiler ikke loggen hvilke av de fysiske operasjonene som er utført om som trenger UNDO. Et annet problem er også at disse operasjonene

[6]Gjenoppretting av databasen etter systemkræsj

[7]Det vil si at resultatet av en REDO eller UNDO er det samme selv om de blir utført flere ganger mot samme side

ikke nødvendigvis er idempotent, kjører man f.eks REDO av en INSERT-operasjon flere ganger vil den sette inn flere poster i databasen. Dette gjør at det ikke er så lett å implementere logisk logging, og har bare blitt brukt i et fåtall databaser. Clustra benyttet seg av logisk logging og gjorde den idempotent ved å inkludere en sekvensielt økende LSN i hver side.

Fysiologisk logging er et kompromiss mellom logisk og fysisk logging. Den bruker logisk logging der det er mulig, og fysisk logging der det trengs. Det vil si at de lagrer hvilken fysisk side endringene er gjort i tillegg til en logisk referanse til hvor i siden endringen er gjort. Denne metoden bruker mindre plass enn fysisk logging, fordi den ikke trenger å lagre tilstanden til siden, samtidig som den unngår konsistensproblemene ved logisk logging. Et eksempel på en INSERT mot en tabell med to indekser vil bli seende slik ut:

```
<insert operation,base filename=233,pagenum=12,record value=r>
<insert operation,index filename=543,pagenum=720,indexRecordValueOf(r)>
<insert operation,index filename=343,pagenum=45,indexRecordValueOf(r)>
```

Fysiologisk logging kan gjøres idempotent ved å bruke sekvensielt økende LSN[8] både i sidene og i loggen, og sammenligne PageLSN med LSN før REDO eller UNDO utføres.

2.3.2 Sjekkpunkting

Å ta regelmessige sjekkpunkter(*eng:* checkpoints) av databasen vil si å lagre nok informasjon på stabilt lager for når som helst å kunne gjenopprette databasen til en konsistent tilstand uten å lese igjennom store deler av loggen [Hva92a]. Dette gjøres av to grunner, for det første medvirker det til at gjenoppretting av en krasjet database går raskere. Dette fordi man bare trenger å lese siste lagret sjekkpunkt i tillegg til en begrenset mengde av loggen. Den andre grunnen er at etter lagring av et sjekkpunkt vet man hvor mye av loggen som trengs for å gjenopprette databasen og man kan derfor slette resten av loggen og frigjøre diskplass.

Det finnes også mange måter å utføre lagring av sjekkpunkt på, og vi vil her presentere fire kjente måter [HR83]:

Transaksjonsorientert sjekkpunkt

Transaksjonsorientert sjekkpunkt (TOC[9]) forutsetter at databasen bruker en FORCE [MHL+92] buffer-strategi. Dette innebærer at endringer som transaksjoner gjør i databasen blir tvunget til disk ved *commit*. Konsekvensen av dette er at vi slipper REDO-fasen ved recovery, og hver *commit* kan derfor sees på som om et sjekkpunkt. Om buffermanager samtidig har en STEAL [MHL+92]-strategi kan deler av endringene være skrevet til disk før *commit* og vi må kjøre UNDO ved recovery. Siden vi ikke har noen informasjon om hvilke transaksjoner som var aktive må hele loggen gjennomløpes. De største ulempene her er altså at det blir tvunget skriving til disk ved hver *commit* og at hele loggen må gjennomløpes om buffermanageren bruker en STEAL-strategi.

[8] LSN - Log Sequence Number som brukes til å knytte en loggpost til en bestemt versjon av disk-siden
[9] Transaction-Oriented Checkpoint

Transaksjonskonsistent sjekkpunkt

Transaksjonskonsistente (TCC[10]) vil si å lagre databasen i en tilstand der det ikke finnes noen delvis utførte transaksjoner. Dette gjøres ved å signalisere at man vil starte et sjekkpunkt, fra dette tidspunkt kan ikke noen nye transaksjoner startes. Når alle aktive transaksjoner har gjort seg ferdig startes sjekkpunkt-operasjonen. Her blir bufferet flushet[11] mot disken og sjekkpunktet notert i loggen. Etter dette fjernes skrivelåsen på databasen. Dette er en sjekkpunktmetode som passer dårlig for store databaser med mange brukere fordi skrivelåsen mens sjekkpunktet lagres medfører en for stor forsinkelse for innkommende transaksjoner. Forsinkelsen kommer av at bufferet flushes mot disken, noe som kan ta lang tid om bufferet er stort. En annen ting som bidrar til forsinkelsen er at sjekkpunktet må vente på at alle aktiv transaksjoner avsluttes før selve sjekkpunktet starte. Transaksjonskonsistent sjekkpunkt passer derfor stort sett bare en-bruker databaser.

Operasjonskonsistente sjekkpunkt

Operasjonskonsistente sjekkpunkt (ACC[12]) opererer i hovedsak på samme måte som TCC, men isteden for å vente til alle aktive transaksjonene er ferdige så bare låser vi databasen for skrive-operasjoner og begynner å lagre sjekkpunktet. Her slipper vi altså å vente til de aktive transaksjonene er ferdige, databasen blir derfor bare låst i perioden det tar å flushe bufferet og skrive sjekkpunkt-loggpost. Denne loggposten inneholder også den aktive transaksjonstabellen fordi det må kjøres *undo* på de operasjonene som tilhører uferdige transaksjoner ved en evt. recovery av databasen. ACC-sjekkpunkter har der for de samme svakhetene som TCC, men i noe mindre grad.

Fuzzy sjekkpunkt

Fuzzy sjekkpunkt (FCP[13]) ligner endel på ACC, men har den store fordelen av at den ikke hindrer at transaksjoner utfører operasjoner samtidig. Det finnes flere FCP-algoritmer, ARIES skisserer en veldig enkel algoritme der den begynner med å skrive en *BEGIN_CHKPT*-post i loggen, for senere å konstruere en *END_CHKPT*-post som inneholder transaksjons-tabellen, dirty-pages-tabellen. Denne skrives så fort den er konstruert. Da har recovery-algoritmen nok informasjon til å raskt bygge opp databasen til en konsistent tilstand, men for å få til det må det kjøres *redo* på alle transaksjoner som hadde skitne sider i bufferet og etterpå *undo* på alle aktive transaksjoner. En ulempe med denne algoritmen er at *hot spots*[14]-blokker medfører at recovery-algoritmen må veldig langt bak i loggen for å finne REDO-informasjon for operasjonene som først berørte blokkene. Lindsay *et al.* [Lin79] foreslo en måte å finne slike *hot spots*-blokker ved å sammenligne data fra forrige sjekkpunkt. Disse blokkene ble tvunget til disk og

[10] Transaction-Consistent Checkpoint

[11] Å flushe bufferet vil si å lagre blokkene som ligger i bufferet mot disken dersom blokken i bufferet er av nyere versjon eller markert skitten

[12] Action-Consistent Checkpoint

[13] Fuzzy Checkpoint

[14] Blokker i databasen som er brukt hyppig og befinner seg i lengre perioder i bufferet i “skitten” tilstand

reduserer derfor behovet for *redo* ved recovery. Hvasshovd [Hva92a] presenterer to andre algoritmer, kalt *basic FCP* og *penultimate FCP*. Begge disse algoritmene flusher alle skitne sider under sjekkpunkt-operasjonen, men for å sikre at sidene er konsistente og ikke delvis i en skriveoperasjon må det settes en lese-latch på de mens de flushes. Denne latchen fjernes like etter siden er flushet og vil bare i liten grad påvirke ytelsen til andre kjørende transaksjoner.

Basic FCP kjennetegnes av at det i loggen skrives en start- og en slutt-post for sjekkpunktet. I start-posten lagres gjerne transaksjonstabellen for aktive transaksjoner, slutt-posten lagres etter at alle skitne sider i bufferet er flushet til disken. Ved recovery er det nok å gå til det siste fullstendige sjekkpunktet.

Penultimate FCP kjennetegnes av at det ikke skrives en start- og slutt-post i loggen, men at det her bare skrives én logg-post for hvert sjekkpunkt. Den fungerer ved at man med jevne mellomrom flusher sider som er skitne og ikke har blitt flushet siden siste sjekkpunkt. Dette er mulig fordi slutten av et sjekkpunkt også fungerer som starten på det neste og kan derfor fortelle det neste sjekkpunktet hvilke skitne sider som ble flushet mot disken. Resultatet av denne metoden er at I/O-belastningen reduseres noe i forhold til *basic FCP*. Ved recovery må man gå til den nest siste sjekkpunkt-posten i loggen, noe som medfører mer recovery-arbeid enn med *basic FCP*.

Ved implementering av FCP at det fort gjort å bryte med WAL-prinsippet om man ikke er forsiktig [SGM89]. WAL-baserte systemer kan lagre endringer på samme sted hvor de ble hentet fra disken, før endringene blir skrevet til disken må det derfor antas at endringen er loggført i loggen [MHL+92]. Faren er at FCP-algoritmen flusher en side i bufferet mot disken før operasjonen mot sida har blitt loggført. Om databasen krasjer før loggen er skrevet blir ikke sida gjenopprettet. Måten dette tradisjonelt løses på er å sjekke om hver skitne side som flushes til disken har loggført alle endringer. Dette gjøres ved å sammenligne PageLSN med høyeste registrerte LSN ved starten av sjekkpunktet. Om en operasjon ikke er loggført blir alle loggposter opp til PageLSN flushet mot disken [HTBH95].

2.4 Recovery i main-memory databaser

Recovery-teknikkene i delkapittel 2.3.2 er ment for DRDB-er der dataene til enhver tid ligger delvis på disk og delvis i minnet. Dette i motsetning til en MMDB der hele tilstanden til databasen ligger lagret i minnet. Disk vil ofte bli brukt for å sørge for persistens av dataene slik at de kan gjenopprettes senere. Siden transaksjonen ikke commiteres før endringene er skrevet til loggen på disk blir det derfor ekstra viktig å redusere denne disk-I/O-en mest mulig for å gjøre responstiden så liten som mulig.

Vi vil her ta for oss de viktigste aspektene ved MMDB-recovery.

2.4.1 Logging

For å sikre persistens av dataene kreves det at alle operasjoner som oppdaterer databasen må loggføres på et stabilt laget. Dette kravet ødelegger ofte for den potensielt veldig lave

responstiden en minnebasert database kan tilby.

Selve loggingen kan gjerne skje på samme måte i en MMDB som i en DRDB, men loggingen kan også endres for å utnytte egenskapene til en MMDB. For det første er det viktig å minimere loggingen mot disk mest mulig fordi dette påvirker ytelsen mer enn hos tradisjonelle DRDB-er. Teknikkene som blir brukt til dette er som nevnt for DRDB i 2.3.1.

M. Eich *et al* [LE92] presenterte f.eks en ny logge-strategi kalt LAW (*Logging After Writing*) for MMDB-er i motsetning til WAL (*Write Ahead Logging*) som vanligvis benyttes i DRDB-er. LAW-strategien går ut på å utføre operasjonen i stabilt lager før REDO-delen logges. På denne måten er man sikret at alle operasjoner som utføres etter BEGIN_CHECKPOINT også blir loggført etter BEGIN_CHECKPOINT. Dette gjør at man ved recovery ikke trenger å gå lengre tilbake enn til BEGIN_CHECKPOINT-loggposten før man starter REDO-fasen.

For en MMDB'er som logger til disk vil flushinga av loggposten være det største bidraget til responstid. Det er presentert flere måter å unngå denne vesentlige forsinkelsen på. En teknikk som kan brukes er å bruke et loggbuffer i ikke-flyktig minne, dette vil redusere responstiden vesentlig. Den maksimale throughput vil ikke øke tilsvarende mye fordi loggdataene likevel må skrives til disk. Ulempen her er selvfølgelig den økonomiske kostnaden ved ikke-flyktig minne som ikke er hyllevare. Clustra benyttet en annen teknikk kalt *log-to-neighbor* [HTBH95]. Der blir loggpostene sendt to nabo-noder på uavhengige krasjdomener, så istedenfor å vente på at loggposten er lagret på disken som typisk tar er det nok å vente på positivt svar fra to nabo-noder. Dette gir en vesentlig lavere responstid samtidig som det gir tilstrekkelig sikkerhet for at loggpostene ikke forsvinner.

2.4.2 Sjekkpunkting

Sjekkpunktsalgorimer for MMDB-er fokuserer på å påvirke pågående transaksjoner minst mulig og samtidig sørge for at gjenoppretting av databasen går så fort som mulig. Gruenwald *et al* [GD96] klassifiserte sjekkpunktalgoritmene i tre grupper; *fuzzy*, *ikke-fuzzy* og logg-drevet.

Fuzzy sjekkpunkting

Hagmann var den første som introduserte fuzzy sjekkpunkting for MMDB-er [Hag86]. Denne har mange likhetstrekk med den som var brukt for DRDB-er, men har noen vesentlige forskjeller. I en MMDB må hele databasen dumpes til en backup på disk, her gjøres dette i segmenter eller sider. Hagmann gjorde dette ved å dumpe alle sidene i databasen, uansett om de var skitne[15] eller ikke. Dumpingen ble gjort sekvensielt uten noen annen form for synkronisering enn å skrive en loggpost i loggen for hver side som ble flushet til disk. Dette kalte han en "fuzzy" dump fordi den verken var operasjon- eller transaksjonskonsistent. Resultatet var en sjekkpunktalgoritme som både var rask og krevde veldig liten samkjøring med resten av databasesystemet.

[15] Det vil si endret siden siste sjekkpunkt

Et annet viktig poeng var at en side ikke må overskrive den gamle versjonen av siden, dette forhindrer at en side ødelegges om den bare blir delvis overskrevet. Dette løste Hagmann ved å bruke et sirkulært buffer på disken som hadde plass til minst to sjekkpunkt. På denne måten var det til enhver tid minst ett konsistent sjekkpunkt på disken. Dette ble kalt "sliding monoplexed backups" av K. Salem *et al* i [SGM87].

K. Salem *et al* presenterte i [SGM87] en ny metode å utføre fuzzy sjekkpunting på som de kalte "ping-pong" fuzzy checkpointing. Forskjellen her er at under kopieringa av databasen fra minnet brukes det to backup-databaser som det veksles på å skrive mot. Her blir bare skitne sider skrevet, mens de som ikke var endret siden siste sjekkpunkt ikke blir skrevet.

For å sikre at de skitne sidene blir skrevet til begge backupene på disk, blir det brukt to bits på hver side som indikerer om en side er skrevet mot hver backup. Når en side markeres som skitten blir begge bit satt til 1. De skitne sidene blir først skrevet mot den ene backupen, mens ved neste sjekkpunkt blir de samme skitne sidene skrevet mot den andre backupen. For hver gang en side blir tatt backup av, blir ett av bit-ene satt til 0. Poenget med denne metoden er at man til enhver tid har minst én konsistent backup på disk, og man sparer samtidig mye disk I/O ved å bare skrive ut skitne sider.

Lin og Dunham foreslo en utvidelse til fuzzy sjekkpunkting kalt *Segmented Fuzzy Cehckpointing* [LD96]. Denne metoden går ut på å dele databasen opp i segmenter og utføre sjekkpunkting på hvert av disse segmentene. På denne måten trenger man bare forkaste siste segment istedenfor hele databasen om siste sjekkpunktet blir avbrutt. Her skrives det til loggen for hvert segment som sjekkpunktes noe som fører til noe mer loggskriving, samtidig vil gjenoppretting av databasen gå raskere fordi man leser mindre av loggen og kjører mindre REDO.

Li *et al* viste at man kunne redusere gjenopprettingen ytterligere ved å dele databasen i partisjoner som hver hadde sin egen loggedisk [XL95]. Ved å distribuere loggen vil gjenopprettelsestiden reduseres enda mer.

Ikke-fuzzy sjekkpunkting

Ikke-fuzzy sjekkpunkting medfører vanligvis mer overhead på vanlig transaksjonsutførelse. Dette kommer av at objektene som det tas sjekkpunkt av må låses for å få et transaksjons- eller operasjons-konsistent sjekkpunkt. Dali brukte opprinnelig en operasjonskonsistent sjekkpunktalgoritme foreslått av Jagadish *et al* i [JSS93]. Den gikk ut på å låse én og bare én side om gangen, flushe sida til disken og samtidig skrive evt. UNDO-loggposter som berørte sida. REDO-log ble skrevet som normalt. Senere ble denne metoden forkastet pga dårlig ytelse fordi hver transaksjon måtte vedlikeholde en egen UNDO-log og REDO-log for hver side den aksesserte i tillegg til den globale REDO-loggen.

Logg-drevet sjekkpunkting

Logg-drevet sjekkpunkting går ut på å regelmessig "oppgradere" backupen på disk ved hjelp av logg-data. Fordelen med dette er at man ikke trenger å aksessere selve

databasen som ligger i minnet. Denne metoden ble opprinnelig brukt for å distribuere en backup av en database over et nettverk [GMP90, Lyo90] og har senere blitt brukt i MMDB-er [LS92, LN88, Eic86]. Gruenwald konkluderer imidlertid med at den høye prosesseringshastigheten som MMDB-er tillater kan medføre at størrelsen på loggen øker veldig raskt. Dette vil gjøre det lite effektivt å mellomlagre loggen for senere å bruke den til å generer et sjekkpunkt istedenfor å bruke *fuzzy* sjekkpunkting og lagre data direkte fra minnet.

2.4.3 Gjenoppretting

Ved systemkrasj mister man all data i minnet, og databasen må gjenopprettes ved oppstart. Systemkrasj vil typisk være strømsvikt, krasj i operativsystem, krasj i databasesystemet eller hardware-feil.

Gjenoppretting eller innlasting av en MMDB er en noe mer omstendelig prosess enn for DRDB-er siden data ligger primært i minnet. Databasen kan derfor ikke utføre databaseoperasjoner før hele eller deler av databasen er lastet inn fra stabilt lager.

Gjenopprettings-metodene kan grovt deles inn i to kategorier, enkel gjenoppretting (eng: *simple reloading*) eller samtidig gjenoppretting (eng: *concurrent reloading*).

Enkel gjenoppretting

Enkel gjenoppretting vil si å først utføre gjenopprettinga i sin helhet før man åpner databasesystemet for eksterne databasetransaksjoner. Dette er den raskeste og enkleste måten å gjenopprette en database på, men medfører at databasen er blokkert i tiden gjenopprettinga tar [GD96]. Det er mulig å dele gjenopprettinga inn i to faser; først lastes siste backup inn i minnet, så brukes loggen til å bringe dataene til siste transaksjonskonsistente tilstand. Margaret Eich presenterte i [Eic87] en måte å gjøre disse fasene i parallell. Mens den ene prosessoren laster inn siste backup, prosesserer den andre prosessoren loggen og lagrer modifiserte sider i ikke-volatilt minne. Ved transaksjonsprosessering sjekkes først det ikke-flyktig minnet for å finne eventuelle endrede sider før siden i volatilt minne sjekkes. En ulempe her er selvsagt at ikke-flyktig minne kreves, samtidig som ytelsesøkningen er ubetydelig på store database siden tiden det tar å laste inn sjekkpunktet er mye høyere enn prosesseringa av loggen [Hag86].

Samtidig gjenoppretting

Samtidig gjenoppretting vil si å utføre gjenoppretting av databasen samtidig som det tillates eksterne databasetransaksjoner. Etter at databasesystemet har gjenopprettet systemtabellene åpnes det for eksterne transaksjoner. Databasen er her delt opp i partisjoner, gjerne i store biter som tabeller og indekser. Når en transaksjon trenger data som ikke ligger i minnet, må den vente til den aktuelle partisjonen er lastet inn. Dette kan ta lang tid om partisjonen er stor, og det hindrer samtidig andre transaksjoner som er avhengig av andre partisjoner. Gruenwald har foreslått tre forskjellige algoritmer: *Ordered Reload with Prioritization (ORP)*, *Smart Reload (SR)* og *Frequency Reload*

(FR) [GD96]. Forskjellen mellom de er måten de behandler transaksjoner under gjenopprettingen, de tar ulike hensyn til bla. strukturen på disk, historisk aksessmønster mot partisjonene, ulik prioritet på transaksjoner i ulike stadier og mengde data som må lastet for å gjennomføre en transaksjon.

2.5 Tidligere arbeid

Det har blitt utviklet mange MMDB'er, og vi vil her presentere de mest relevante egenskapene til de mest kjente. Dette kapittelet er en noe modifisert utgave av funnene vi gjorde i [Sol06].

2.5.1 Starburst

I 1984 startet IBM databaseprosjektet Starburst. Målet med dette prosjektet var å bygge en fungerende prototype av en relasjonsdatabase som enkelt tillot brukere å lage sine egne utvidelser til databasen. Siden man tenkte på utvidbarhet gjennom hele designet av Starburst medvirket dette sterkt til at brukere og forskere enklere kunne eksperimentere med ny databaseteknologi. Starburst tillot bla. brukerne å lage egne lagerkomponenter, nye aksessmetoder og datatyper.

Blant de mange utvidelsene som ble utviklet var ”Starburst's Memory Resident Storage Component” [LSC92]. Dette var en ganske stor utvidelse av sentrale deler i Starburst og bestod av en ”Main Memory Relation Manager (MMM)” og to forskjellige indeksstrukturer, T-Tree og modified linear hash index. Utvidelsen bestod av ca 20000 linjer kode og erstattet rundt 400000 linjer kode i Starburst-kildekoden og tok ca et årsverk å utvikle. Dette var altså en ganske stor forenkling av databasen.

Ytelsestestene [LSC92] viste at den gjennomsnittlige utføringstiden for lese-spørringer var 1.5-4.8 ganger raskere enn den originale lagerkomponenten. Det som bidrog mest til dette var elimineringen av buffermanager, mer effektive data- og indeks-strukturer og generelt en mindre kompleks database.

2.5.2 TimesTen

TimesTen In-Memory Database [Ora06] er en kommersiell MMDB utviklet av selskapet TimesTen. De ble i 2005 kjøpt opp av Oracle og de selger nå produktet som Oracle TimesTen In-Memory Database. Denne relasjonsdatabasen ble utviklet for å gi sanntidsytelse og høy tilgjengelighet. Systemet baserer seg på å lagre data i minnet, og skrive logg til disk. Ellers har den en tradisjonell databasearkitektur.

Den tilbyr to typer indekser: en som er basert på hashing og en på t-tre. På *primary key* opprettes det automatisk en hashbasert indeks for å sikre raskt oppslag med likhetsoperator på primærnøkkel. Man kan opprette en t-tre-basert indeks ved hjelp av en CREATE INDEX setning.

Som navnet sier, hevder de at den er ti ganger raskere enn en DRDB. Uavhengige ytelsestester har derimot vist en noe mindre økning. I en TPC-W benchmark, som har

en belastning lik typiske web-applikasjoner ble det konkludert med at TimesTen hadde ca dobbelt så høy throughput som Oracle 8i [CLK+06].

2.5.3 MySQL Cluster

MySQL Cluster [AB06a] er en utvidelse av MySQL-databasen som tilbyr et "shared-nothing"-cluster som baserer seg på lagring av databasen i minnet. Arkitekturen er designet for høy ytelse, høy tilgjengelighet og tilnærmet lineær skalerbarhet. Den bruker logging mot disk på hver node for å oppnå durability av dataene.

Den er implementert som en alternativ lagerkomponent kalt NDB (**N**etwork **D**ata**b**ase) og har vært tilgjengelig siden versjon 4.1 (November 2004). Databasen støttet i første omgang bare en hash-basert index, men fikk senere støtte for en T-tre-basert index som gav støtte for områdesøk. En hash-basert index på primary-key var imidlertid 20-30 % raskere enn T-tree på inserts, dette på grunn av at det kreves mindre minne og mindre CPU-tid [AB06b].

Den krever at alle tabeller har en *primary key*, men støtter foreløpig ikke *foreign keys*. For å spre data i clusteret bruker den hashpartisjonering på *primary key*, om det ikke finnes en *primary key* lages denne automatisk.

Ytelsene skal være 5-10 ganger raskere enn en DRDB. Tester utført av Japan OSS Promotion Forum [For05] viser at Mysql Cluster er ca. 4 ganger raskere enn Mysql med InnoDB på *Insert* og *Update*, men bare dobbelt så rask på *Select*.

2.5.4 Dali/Datablitz

Dali [JLR+94] var et forskningsprosjekt ved Lucent's Bell Labs som utviklet en veldig rask MMDB, produktversjonen av denne databasen ble kalt DataBlitz [Ew97]. DataBlitz er et lagringssystem for persistent data optimalisert for å være minne-resistent. DataBlitz bruker en memory-mappet arkitektur der databasene på serveren blir mappet til et virtuelt adresserom til de kjørende prosessene.

Den kan også lagre data utenfor minnet, men er optimalisert for en database der alle data ligger i minnet. Spesielt recovery-mekanismen er designet for å levere høy ytelse når databasen får plass i minnet. Dette er deler av arkitekturen som skiller DataBlitz fra en vanlig DRDB:

- Bruker recovery-mekanismene skissert av Jagadish *et al.* i [JSS93]. Dette innebærer bla. at kun redo-records blir skrevet til disk, mens undo-records blir holdt minneresistente og blir forkastet når en transaksjon committer. Checkpointing er også forenklet her, ved at den er en atomisk operasjon og lagrer bare undo-loggen, selve main-memory-databasen, aktive transaksjoner og siste commit-sekvensnummer. Dette fører til at checkpoint-algoritmen blir både raskere og gir mindre påvirkning på samtidigheten i systemet enn f.eks algoritmen foreslått av Lehman og Carey [LC87]. Recovery blir også en forholdsvis rask operasjon fordi den trenger bare å lese igjennom loggen en gang.

Id	Name	Postal Code	Date of Birth
1	John	2345 BP	17-09-1976
2	Jane	6146 TY	21-04-1959
3	Bob	8127 PR	04-04-1990

Id	Name
1	John
2	Jane
3	Bob

Id	Postal Code
1	2345 BP
2	6146 TY
3	8127 PR

Id	Date of Birth
1	17-09-1976
2	21-04-1959
3	04-04-1990

Figur 2.7: Vertikal fragmentering av data som brukt i Monet

- Databasen er partisjonert i *databasefiler* som kan mappes til et delt minne område. Dette har flere fordeler, blant annet kan flere brukerprosesser adressere dataene direkte og bare deler av databasen trenger å mappes til minnet.

- Databasen har innebygt toleranse for softwarefeil. Siden data i minnet er delt av flere prosesser, er det viktig å kunne tilby påvisning og recovery av korrupterte data.

- Er veldig konfigurerbar, bla. kan samtidighetskontroll og logging slåes av om det ikke trengs. Databasen tilbyr i tillegg remote backup og hot spares.

- Tilbyr komprimering av enkeltdata der headeren er av fast størrelse og en komprimert datadelen som er av variabel størrelse.

Som de fleste MMDB'er støtter DataBlitz indekser basert på T-trær og i tillegg tilbyr den to typer hashbaserte indekser.

2.5.5 Monet

Monet [BK95] er et databasesystem utviklet hos CWI og University of Amsterdam og har vært under utvikling siden 1992. Monet har en databasekjerne spesielt laget for spørreintensive oppgaver som f.eks OLAP(Online Analytical Processing) og datavarehusapplikasjoner. Det skaperene av Monet har observert er at mens ytelsen til de fleste type hardwarekomponenter har økt eksponentielt opp gjennom tidene, har I/O-kostnaden og minnelatency ikke gått like fort ned. Resultatet er en eksponentielt økende flaskehals. For å unngå mest mulig I/O under spørringer, bruker den vertikal fragmentering av data i motsetning til horisontal fragmentering som vanlige databasesystemer bruker. Dette er illusertert i figur 2.7. Undersøkelse gjort av P.A Boncz [Bon02] viser at spørre-intensive aksessmønster har nytte av vertikal fragmentering av data. Det medfører bla mindre I/O og øker treffsikkerheten for cache-algoritmer.

2.5.6 Stephen Fitch sin MemoryStorageFactory

Stephen Fitch har tidligere jobbet med en minnebasert lagringsmodul til Derby [Fou07]. Denne modulen kalt MemoryStorageFactory implementerer StorageFactory og kan brukes av RawStore i Storage-laget(se seksjon 3.1) som et alternativ til for eksempel DirStorageFactory som lagrer data som filer i filsystemet. En StorageFactory må tilby aksessering av `Container`-objekter som inneholder data for en Conglomerate. I DirStorageFactory mappes et Container-objekt mot en fil i filsystemet som består av flere Page-objekter, mens i MemoryStorageFactory mappes en Conglomerate mot en minneresident byte-tabell som inneholder flere serialiserte Page-objekter. I Derby kan bare én StorageFactory brukes per database, man kan derfor ikke bruke MemoryStorageFactory samtidig som DirStorageFactory.

Fordelene ved å gjøre det på denne måten er bla:

Lite kode å endre - Man trenger bare å implementere en ny `StorageFactory`, og resten av databasen er helt uberørt.

Rask tilgang til kalde data - *Kalde* data, data som sjelden blir brukt vil være mye raskere å hente her siden alle data ligger i minnet. I en database som har all data liggende på disk tar det vesentlig lengre tid å hente *kalde* data enn *varme*. Dette i motsetning til *hot spots* (varme data) som vil ligge i minnet i begge tilfeller.

Ingen disk I/O - I noen tilfeller vil man minimere disk IO, og denne måten å lagre data på vil ikke generere I/O direkte. Om man ønsker persistens må loggen fortsatt skrives til disk ved å spesifisere DirStorageFactory, om dette ikke ønskes kan MemoryStorageFactory også brukes til logging og eliminere disk-I/O helt.

Å gjøre det så enkelt som å lagre `Page`-objektene i minnet istedenfor disken har også flere ulemper:

Fortsatt bufring - Databasen vil forsatt tro at data blir lagret til eller hentes fra disken, og vil mellomlagre `Page`-objektene i bufferet til `CacheManager`. Dette fører til dobbeltlagring av data i minnet, noe som verken er nødvendig eller særlig heldig med tanke på at det er en begrenset ressurs. I en DRDB er også sannsynligheten stor for at diskblokker blir mellomlagret i filsystemet eller på disken sin innebygde buffer. Denne mellomlagringen er derfor unødvendig kostbar siden data allerede finnes i minnet. Dette kan til en viss grad løses ved å sette størrelsen på bufferet til et minimum.

Checkpointing - Derby utfører checkpointing ved å skrive alle skitne sider i bufferet til `StorageFactory` for data, og lagre redo- og undo-logg til evt. en annen `StorageFactory`. Når `StorageFactory` for data også er volatilt betyr det at all data i databasen forsvinner ved krasj og vil gjøre checkpoint'et ubrukelig. Recovery vil da feile om Derby ikke har tatt vare på all logg-data, recovery vil da i tilfelle ta veldig lang tid fordi alle loggfilene siden databasen ble opprettes må traverseres.

Optimalisereren kan ta feil - Optimalisereren kan ta avgjørelser på feil grunnlag siden en minneaksess har andre egenskaper enn en diskaksess. I `MemoryStorageFactory` vil henting av data utenfor cache ha samme kostnad som å hente fra cachen.

Overhead på lagring - Det koster endel å serialisere objekter, men dette må gjøres for å lagre objektene på disken. Dette er derimot ikke nødvendig for å lagre objektene i minnet og vil her utgjøre en unødvendig overhead.

Ineffektive aksessmetoder - Aksessmetodene er utviklet med hensyn på egenskapene til disk, det er derfor sannsynlig at dette ikke er den mest effektive måten å aksessere data i minnet på.

Ut fra dette kan man trekke den konklusjonen at dette er en enkel men ikke den mest effektive måten å implementere et minnebasert lager i Derby. Hvis man har nok minne til å romme hele databasen, kan det faktisk være like greit å bare øke størrelsen på bufferet til cachen slik at hele databasen får plass der. Da vil databasen finne alle data i cachen, istedenfor å først lete i cachen for deretter å lete i `MemoryStorageFactory` om dataene ikke finnes der. Om persistens ikke ønskes kan man deaktivere logging i RawStore og flushing av Page-objekter i CacheManager og få en minst like høy ytelse som MemoryStorageFactory.

I tilfeller hvor dette kan være fornuftig er hvis man har flere databaser kjørende, og ønsker å ha en av de i minnet hele tiden. Men man bør da prøve å få cache-manageren til å la være å cache sider tilhørende den minneresistente databasen.

I høstprosjektet vårt i 2006 [Sol06] utførte vi en liten test for å sammenligne MemStorageFactory og DirStorageFactory. Resultatene viste at for innsettinger var ytelsen for MemStorageFactory 100% høyere enn for standard Derby, mens for søk var ytelsen den samme. Vi har grunn til å tro at økningen i thoughput på innsettinger først og fremst skyldes at loggen også blir skrevet til minnet.

2.5.7 Clustra

ClustRa [HTBH95] er en database utviklet av ClustRa AS i 1995. De ble kjøpt opp av Sun Microsystems i 2002 og databasen skiftet navn til HADB. Dette er en database utviklet for bruk i telekommunikasjonstjenester. Her er kravene til responstid, throughput og ikke minst tilgjengelighet viktige. For å møte disse kravene ble det brukt flere både nye og gamle teknikker for å øke ytelse og tilgjengelig i forhold til tradisjonelle databaser. Det ble brukt bla. flere *shared-nothing* noder spredd på minst to lokasjoner med replikering og online self-repair for å øke tilgjengeligheten. Main-memory-logging med logg-shipping til nabonoder [Hva92b] og parallell transaksjonsbehandling ble brukt for å bedre responstiden da en transaksjon ikke trenger å vente på at loggen blir skrevet til disk. Databasen var også designet slik at throughput skalerte veldig bra med antall noder.

ClustRa er en tradisjonell DRDB i den forstand at data ligger på disk og mellomlagres i buffermanageren, men den har også muligheter for å deklarere at tabeller i sin helhet skal ligge i minnet. For indeksering brukes det B-trær med en modifikasjon som hindrer mange side-splittingen under utførelse av innsetting. Det tillates bare én side-splitting i den tidskritiske fasen av en transaksjon, eventuelle resterende splittinger utføres senere med lavere prioritet.

2.5.8 C-store

C-Store [SAB+05] er en kolonneorientert DBMS som et team på MIT ledet av Michael Stonebraker begynte utviklingen av i 2005. Bakgrunnen for C-store var å lage et databasesystem som var lese-optimalisert istedenfor skrive-optimalisert som de fleste DRDB-systemer er. De er skrive-optimalisert fordi de lagrer radene fortløpende i lageret, det er derfor tilstrekkelig med én skriveaksess for å lagre raden på disken. For applikasjoner med arbeidsbelastning lik OLTP er dette effektivt, men for datavarehus, CRM-systemer, bibliotekdatabaser og systemer som krever mange tilfeldige og begrensede søk i store databaser er det mer effektivt å optimalisere for lesing. C-store er optimalisert for lesing og lagrer dataene kolonnebasert eller vertikalt på samme måte som Monet.

Datalageret består av to deler, *Writeable Store (WS)* som tillater innsetting og oppdateringer med høy ytelse og *Read-optimized Store (RS)* som er den største av de to. Disse er knyttet sammen ved hjelp av en *tuple mover* som flytter tupler fra WS til RS. Innsettinger blir sendt til WS mens forespørsler om sletting blir sendt til RS der tuplene blir markert før *tuple mover* vil fjerne de etterhvert. Oppdateringer blir behandlet som en innsetting og en sletting. Dataene i RS er kolonnebasert på samme måte som i Monet og figur2.7, de er også komprimert for å redusere plassforbruket. Komprimeringsteknikkene er laget slik at sorteringsrekkefølgen bevares, dataene trenger derfor vanligvis ikke dekomprimeres før de blir presentert for brukeren. Kolonnene knyttes sammen til tabeller ved hjelp av egne join-indekser.

Bufferstrategien er NO-FORCE/STEAL [GR93], men skiller seg noe fra tradisjonelle implementeringer. Blant annet logges bare UNDO-poster, recovery foregår ved å konsultere andre noder istedenfor loggen. *Commit*-protokollen 2PC [GR93] er også endret slik at PREPARE-fasen ikke er nødvendig. Det benyttes logisk logging fordi fysisk logging ville resultert i mange flere loggposter på grunn av datastrukturen i WS.

For indeksering i RS brukes tettpakkede B-trær som har en fyllingsgrad på 100%. Dette kan gjøres fordi RS ikke trenger online oppdateringer. I tillegg kan det brukes bitmap-indekser på kolonner. Dette er en rekke bits som representerer dataene i kolonnen og kan brukes til å besvare de fleste spørringer bare ved hjelp av logiske operatorer. Dette gir betydelige ytelsesforbedringer i forhold til tradisjonelle DBMS-er. Ytelsestester viser at C-Store i gjennomsnitt er 164 ganger raskere enn en radbasert kommersiell DBMS og 21 ganger raskere enn en kolonnebasert DBMS [SAB+05].

Å benytte kolonnebasert lagring i Derby er ganske uaktuelt fordi det vil kreve for store endringer i systemer.

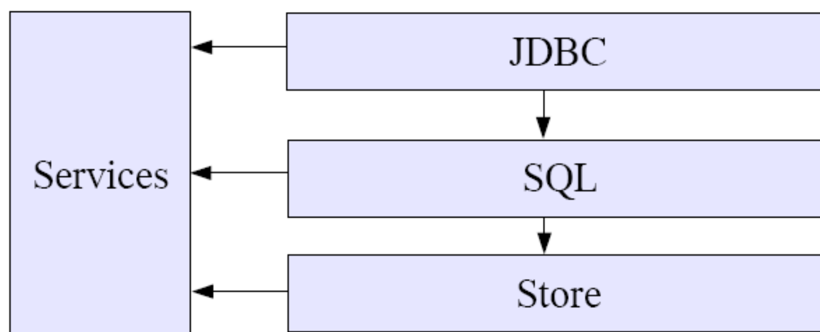
Kapittel 3
Derby

Vil her gå kortfattet igjennom arkitekturen i Derby og identifisere hvilke deler som berøres når vi vil optimalisere ytelsen for en database som i sin helhet ligger i minnet. Dette kapittelet er en noe modifisert utgave av et tilsvarende kapittel hentet fra vårt tidligere prosjekt høsten 2006 [Sol06].

3.1 Introduksjon til derby

Derby er sterkt lagdelt som vist i figur 3.1, og består i grove trekk av disse lagene: JDBC, Query Execution og Store.

Store-laget er delt i to deler, access og raw. Access-laget tilbyr et radbasert interface mot Query Execution-laget. Det hånterer tabell-skanning, index-skanning, indeksoppslag,



Figur 3.1: Arkitekturen til Derby

indeksering, sortering, låsepolitikker, transaksjoner og isolasjonsnivå. I dette laget finnes det forskjellige moduler som tar seg av de ulike oppgavene og som kommuniserer med Raw-delen i Store-laget. Raw-delen tilbyr et sidebasert interface for lagring av rader i sider som igjen blir lagret i filer på disken. Accesslaget må derfor ha logikk for å finne ut hvilken side en rad skal lagres i eller hentes fra.

Skal man f.eks lage en ny låsemanager-modul lager man først en implementasjon som implementerer interfacet LockFactory og registrerer man den i fila *modules.properties* slik: `derby.module.lockManager2=java class name` og kommentere ut den gamle modulen.

3.2 Moduler og monitor

En kjørende Derby database består av en monitor og en samling av *services* eller tjenester som hver har sin oppgave [Dera]. En slik tjeneste består av en eller flere moduler som tilsammen implementerer funksjonaliteten til tjenesten. Eksempel på slike tjenester kan være loggemanager, buffermanager og låsemanager. En av tjenestene er helt spesiell, og kalles *System Service*

I figur 3.1 illustreres det at *Service*-laget brukes av alle lagene i Derby.

En modul kan f.eks være en låsemanager, JDBC-driver, lagermodul eller mindre deler av disse. Modulen må oppfylle et interface for oppgaven den skal utføre, f.eks må låsemanager-modulen oppfylle interfacet `org.apache.derby.iapi.services-.locks.LockFactory`. I tillegg må en modul ha en identifikator som kan være enten en String, UUID eller *null*.

Moduler kan enten være delt for hele systemet, f.eks logging eller den kan bare være delt for en database, f.eks en låsemanager.

Alle kall etter nye moduler går igjennom modulen `Monitor`. Dette er en systemmodul som er felles for alle databaser som kjører under en instans av Derby. Denne modulen må derfor sørge for å ha oversikt over modulene og kunne starte og stoppe de etter behov. Den søker etter moduler slik [Derb]:

- Sjekker om en potensiell modul oppfyller kjente modulinterfacer, hvis ikke hopp over til neste potensielle modul.

- Sjekke om den etterspurte implementasjonsidentifikatoren er lik. Hvis ikke hopp over til neste.

- Sjekker vha `BaseMonitor.canSupport()` om den potensielle modulen kan tilby spesielle krav gitt i et Property-set som den som kaller modulen gir. F.eks kan RawStore-modulen bli stilt krav til tilby recovery. Om slike krav finnes og modulen ikke oppfyller de hopper den til neste potensielle modul.

3.3 Låsing og samtidighet

Låsing av databasen eller deler av den er en mekanisme som brukes for å begrense en transaksjons tilgang til delte data. Dette må gjøres for å hindre inkonsistens i databasen, en av de fire ACID egenskapene. Konsistens kan deles opp i fire grader [GLPT94], som vist i listen under. De tilsvarende isoleringsnivåene til JDBC-interfacet er også navngitt [Der06c].

Grad 0 beskytter andre brukere mot dine oppdateringer. Kalles TRANSACTION_READ_UNCOMMITED i JDBC og READ_UNCOMMITTED (UR) i Derby.

Grad 1 gir ytterligere beskyttelse mot å miste oppdateringer, kommiterer ikke data før alt er skrevet til stabilt lager. Kalles TRANSACTION_READ_COMMITTED i JDBC og (CS) i Derby.

Grad 2 beskytter ytterligere mot lesing av ukorrekte data på en slik måte at `select`-setninger som utføres flere ganger vil uansett gi samme resultat i samme transaksjon. Kalles TRANSACTION_REPEATABLE_READ i JDBC og REPEATABLE_READ (RS) i Derby.

Grad 3 beskytter ytterligere mot å lese data med ukorrekte forhold blant dataelementene (Total beskyttelse). Kalles TRANSACTION_SERIALIZABLE i JDBC og SERIALIZABLE (RR) i Derby.

Derby tilbyr altså fire isolasjonsnivåer for transaksjoner gjennom det standardiserte JDBC-interfacet, men den bruker internt en litt annen syntaks på nivåene. Skal man sette isolasjonsnivået selv, må man bruke Derby-syntaksen som man gjør det via SQL-parseren eller bruke JDBC-syntaksen om man gjør det via JDBC.

Isolasjonsnivåene kan oppfylles ved hjelp av låsing av databasen. Disse låsene kan ha ulik granularitet, alt fra hele databasen til en rad eller kolonne i en tabell. Det enkleste er selvsagt å bare ha én type lås som låser hele databasen når en transaksjon leser eller oppdaterer databasen, da er høyeste isolasjonsnivå oppfylt. Dette får selvsagt store konsekvenser for samtidigheten i databasen, derfor er en lås av mindre granularitet å foretrekke. En rad-basert låsemekanisme vil øke samtidigheten, men vil gi en viss overhead i form av høyere minneforbruk og mer beregningstid sammenlignet med låser av høyere granularitet. Dette fordi det må settes flere låser. I Derby er standard isolasjonsnivå READ_COMMITTED.

Derby tilbyr som de fleste databasesystemene idag både tabell- og rad-basert låsing. Som standard bruker den rad-basert låsing, men den har mulighet til å eskalere granulariteten til tabellbasert om antall låser per transaksjon i en tabell overstiger en konfigurerbar terskelverdi for antall låser [Der06b]. Optimalisereren kan også velge å bruke tabell-låsing for spørringer som ikke gir betydelig mer samtidighet ved radbasert låsing, f.eks en spørring med et WHERE-predikat som ikke bruker en index må uansett søke igjennom alle radene i tabellen. Dette er stikk i strid av hvordan MMM-utvidelsen [LSC92] til Starburst gjør det, den begynner med tabell-låsing og de-eskalerer til rad-basert låsing når en eller flere transaksjoner er blokkert. Grunnen til denne forskjellen kan være at Derby først og fremst prioriterer samtidighet mens Starburst sin MMM [LSC92] prioriterer ytelse. Et annet poeng kan være at Derby sin låsemekanisme er "dyrere"

enn den brukt i MMM slik at det er viktigere å holde totalt antall låser nede.

Selve låseinformasjonen må ligge i en ekstern datastruktur for å få rask tilgang til statusen på låsen, i Derby ligger den i en hashtabell. I en MMDB der alle dataene ligger i minnet trenger man ikke en ekstern hashtabell, men kan lagre låsestatus i samme datastrukturen som dataelementet slik de gjorde i MMDB-utvidelsen til Starburst [LSC92].

	Single-record select (10 clients)		Single-record update (20 clients)		Join (4 clients)	
Module	**System & user CPU**	**Wall clock time**	**System & user CPU**	**Wall clock time**	**System & user CPU**	**Wall clock time**
Network server	14.5%	47.0%	10.8%	6.0%	1.3%	1.0%
JDBC	4.1%	1.4%	1.0%	0.0%	2.4%	1.0%
Execution	7.9%	2.6%	8.1%	0.4%	4.7%	2.7%
Access	18.1%	7.0%	13.2%	0.8%	32.5%	24.6%
Lock manager	19.7%	30.9%	18.8%	10.0%	13.5%	40.6%
Buffer manager	6.8%	2.3%	5.5%	0.3%	1.3%	0.7%
Logging	0.0%	0.0%	11.0%	81.1%	0.0%	0.0%
Data store	23.6%	7.2%	24.4%	1.1%	33.4%	22.2%
Transaction control	5.3%	1.8%	7.3%	0.3%	10.8%	7.2%

Figur 3.2: Ressursbruk i Derby [AD06]

For å sikre fysisk konsistens i datastrukturene på disk og i minnet, bruker Derby som de fleste andre databasesystemer latcher for å låse strukturen mens man oppdaterer de. Dette er raske og kortlivde låser som i de fleste databasesystemer bare koster en tiendel av hva en ekslusiv lås koster [GL92][MHL+92]. Derby bruker samme låsemanager til både latcher og låser, grunnen til dette var bla. gjenbruk av kode, enklere feilsøking og enklere brukerstøtte [ddml06]. Det ble antydet i [ddml06] at å sette en latch i Derby koster mellom 50% og 60% av å sette en lås. Som det fremgår i figur 3.2, utgjør låsemanageren en stor del av kjøretiden.

3.4 Caching

Henting av data fra disk er en stor kostnad i et databasesystem, det er derfor et ønske om å gjøre dette så sjelden som mulig. Det optimale hadde vært å lagre all data i minnet, men siden dette ikke alltid er mulig må man finne en annen løsning. Caching er en design teknikk som utnytter seg av minnet som et mellomlager og man får fordelen av å lese store deler av data fra minnet samtidig som man bare bruker en mindre mengde minne. Caching utnytter prinsippet om *lokalitet* som sier at en applikasjon ofte vil lese dataposter sekvensielt og at en stor del av leste poster vil bli leste igjen senere.

Caching i Derby behandles av modulen `CacheManager`, denne kan ha flere implementasjoner. I standard Derby oppfylles denne imidlertid av en clock-algoritme [OE06] i klassen `Clock`. Se klassediagram i figur 3.3 som viser hvordan cache-arkitekturen er bygd opp i Derby. Derby bruker en sidebasert caching av data i databasen.

Som det fremgår i figur 3.3 behandler `CacheManager` ulike `CachedItem`-instanser som igjen holder en referanse til objekter som implementerer `Cacheable` interfacet. I dagens

Figur 3.3: Klassediagram av cache-tjenesten i Derby [Cer06]

Derby implementasjon vil det si følgende objekter [Derb]:

- `FileContainer` - kontainer som inneholder alle sidene til en tabell.
- `CacheablePages` - sider på disken
- `CacheableConglomerate` - instanser av `Conglomerates` (se seksjon 3.5).
- `GenericPreparedStatement` og dens subklasser.

Det er forsket på flere implementasjoner av CacheManageren i Derby, dette er bla. undersøkt i [Sou06] og [OE06]. Begge konkluderer forøvrig med at det finnes cache-implementeringer som er mer effektiv enn Clock-implementeringa som er standard i Derby.

3.5 Conglomerate

Et begrep som går igjen mange steder i kildekoden til Derby er Conglomerate. Direkte oversatt betyr det "sammenklumpet" eller "hope sammen". I Derby er en conglomerate en abstrakt lagringsstruktur og tilsvarer en aksessmetode [Derb]. Alle aksessmetodene må derfor implementere et `Conglomerate`-interface. Et `Conglomomerate`-objekt blir lagret på disk i en egen kontainer gjennom `Storable`-interfacet. Når et `Conglomerate`-objekt blir tatt i bruk mellomlagres det i `CacheManager` for senere bruk.

I standard Derby er det implementert to typer `Conglomerate`:

`B2I` Dette er en indeksstruktur som er basert på B+-tre. Brukes på primærnøkler og indekser som brukeren lager selv. Den er implementert under pakken `org.apache.derby.impl.store.access.btree`.

`Heap` En enkel aksessmetode som søker sekvensielt igjennom rader i en tabell. Den cacher sider funnet på disken underveis ved hjelp av CacheManager, dette sikrer rask tilgang til de samme dataene senere. Den er implementert under pakken `org.apache.derby.impl.store.access.heap`.

I resten av oppgaven blir Conglomerate og aksessmetode brukt om hverandre.

3.6 Persistens

Derby er en klassisk disk-basert database i den forstand at den lagrer rader i blokker som er optimalisert for diskaksess og bufrer disse i minnet etter behov. Hver conglomerate i Derby tilsvarer en Container i access-laget som igjen mappes til en fil i raw-laget. Her lagres f.eks alle radene i en tabell eller dataene i et B-tree. Når en rad i en tabell endres, må først filen tabellen tilhører leses før siden som raden inneholder lastes inn i bufferet. Her blir dataene endret og en logg-post som inneholder hvilke endringer som er gjort blir laget og skrevet til loggen. I Derby brukes det fysiologisk logging som forklart i kapittel 2.3.1. På grunn av at Derby bruker en STEAL/NO-FORCE-bufferstrategi blir endringene utført på siden mens den ligger i minnet før siden evt. senere blir flushet mot disken. Ved systemkrasj vil derfor ikke filene på disken være transaksjonskonsistente.

Derby sin recovery-strategi bygger på ARIES [MHL+92] med noen modifikasjoner. Det benyttes i begge tilfeller *basic* 'fuzzy'-sjekkpunkting, men istedenfor å bare lagre tabellen over skitne sider flusher Derby alle sidene mot disken. Dette gjør at gjennopprettinga av databasen går fortere fordi mindre deler av loggen trenger leses (*Analysis-fasen* trengs ikke) og mindre arbeid i REDO-fasen. En annen forskjell er hvordan LSN[1] blir brukt. Både Aries og Derby benytter LSN for å identifisere loggposter, men i Derby kalles de LogInstants. Forskjellen ligger i at at Derby ikke lagrer LSN på siden, men bruker heller en annet nummer kalt PageVersion som lagres i loggposten for å sikre at operasjoner blir utført idempotent. Årsaken til dette valget var at det var mer fleksibelt og enklere og kode og feilsøke. Når vi skal implementere vår MMDB med eget loggformat er dette bare en fordel.

3.7 MemStore

I prosjektet som vi utførte høsten 2006 [Sol06], utviklet vi en minnebasert lagringsmodul som vi valgte å kalle MemStore. Denne består av en helt enkel lagringsstruktur for tabelldata i minnet, og to tilhørende aksessmetoder kalt MemHeap og MemHash. MemStore er implementert i pakken `org.apache.derby.impl.store.mem` og består av disse klassene:

MemStoreImpl - Dette er implemetasjonen av interfacet `org.apache.derby.iapi.-store.mem.MemStore` som `Monitor` [Derb] bruker for å finne rett implementasjon. Denne modulen blir bootet under System Service [Derb], og er derfor felles for alle databasene som kjører.

[1] Log Sequence Numbers - brukes som unik identifikasjon for logg-poster

MemInsertOperation - Ethvert objekt av denne klassen representerer en *insert*-operasjon mot minnet. Operasjonen blir logget ved å serialisere objektet og lagre det i en log-record. Etter at operasjonen er logget, blir `doMe()` metoden på objektet blir kalt.

MemUpdateOperation - Samme som over, men for SQL-operasjonen *update*.

MemUndoOperation - Kompanserende operasjon for *insert* og *update*, nåværende implementasjon av denne fungerer ikke.

Container - På samme måte som vanlig Derby, samsvarer en container med en tabell i databasen. Den deler opp en container i partisjoner.

Partition - En partisjon inneholder som standard 10000 `Slot`-objekt.

Slot - Et `Slot`-objekt inneholder en rad i en tabell i formatet `DataValueDescriptor[]`, dette kan være et array av kolonner. I tillegg implementerer den `Lockable` og `RecordHandle` slik at objektet kan brukes direkte i låsemanageren. En operasjon som må låse en rad kan da kalle metoden `lockRecordForWrite()` på en `LockingPolicy` [Derb] med `Slot` som argument istedenfor et eksternt objekt som implementerer `Lockable` og `RecordHandle`.

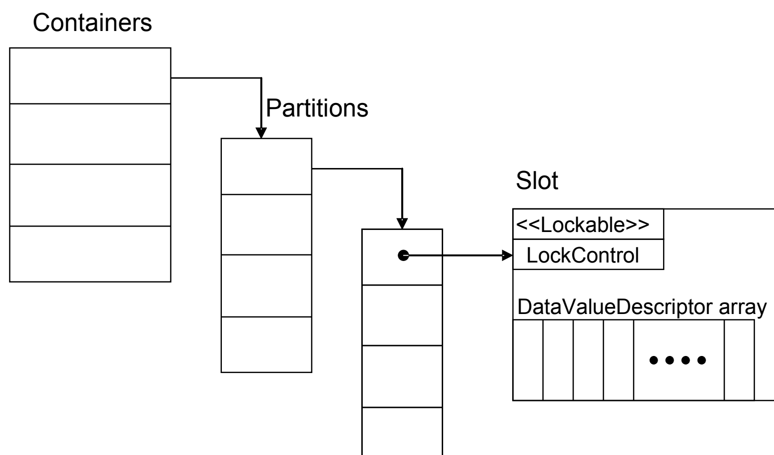
Det er også satt av et felt for et `Control`-objekt. `Control` [Derb] er et interface som implementeres av `Lock` og `LockControl`, og kan inneholde låseinformasjon om en eller flere låser som er knyttet til raden. Låsemanageren kan derfor endres til å lagre låseinformasjonen her istedenfor i en ekstern hashstruktur. Se figur 3.4 for illustrasjon.

Lagringsstrukturen i den gamle MemStore var delvis inspirert av MMM utvidelsen til Starburst [LSC92] og dagens struktur i Derby. Istedenfor å latche sider som i standard Derby, ble det i operasjonene mot MemStore synkronisert på `Container`-objektet. Grunnen til dette var en undersøkelse av T. J. Lehman *et al.* [GL92] som konkluderer med at tabell-latcher ikke medfører vesentlig dårligere samtidighet i en MMDB fordi operasjoner mot minnet er mye raskere enn operasjoner mot disk. I et multiprosessormiljø er dette derimot mindre lurt siden en tråd vil blokkere en annen tråd som vil lese fra samme tabell.

3.7.1 MemHeap

Denne aksessmetoden ble plassert i `org.apache.derby.impl.store.access.memheap`, og er en aksessmetode som tilbyr samme funksjonalitet som aksessmetoden `org.apache.derby.impl.store.access.heap` i standard Derby. Funksjonaliteten er ganske enkel og blir brukt til å aksessere data som om de skulle være lagret i en heap. Den tilbyr ingen funksjon for direkteoppslag på en primærnøkkel uten at en indeksstruktur hjelper til. Bruker man denne aksessmetoden på en tabell uten indeks vil den sekvensielt søke gjennom alle dataene i den rekkefølgen de er lagret på disken eller i dette tilfellet i minnet.

Til forskjell fra Heap [Derb] bruker ikke MemHeap RawStore for å lagre data. Den bruker isteden MemStore(section 3.7) som er en radbasert lagringsmodul. Siden dataene



Figur 3.4: Gamle MemStore sin lagringsstruktur

allerede ligger i minnet går ikke MemHeap omveien gjennom buffermanager når den skal aksessere data, man slipper derfor overheaden dette gir.

Ellers er Memheap og Heap ganske like, de bruker begge RawStore for logging, transaksjonsbehandling og låsing. Istedenfor å bruke Derby sin latche-mekaniske som Heap bruker når den endrer en side og logger evt. endringer [Der06a], bruker vi her heller `synchronized`-primitivet som Java tilbyr for dette. Derby latcher på sider, mens vi har valgt å bruke gjøre mutexer på tabeller, grunnen til dette er at det i [GL92] konkluderes med at tabell-latcher ikke medfører vesentlig samtidighetsreduksjon. En annen grunn er at det er påvist at latche-mekanismen i Derby er forholdsvis kostbar [ddml06], se også figur 3.2.

`MemHeap` implementerer alle interfacene som `Heap` gjør. I tillegg må den implementere de interfacene som `GenericConglomerateController` og `GenericScanController` implementerer siden den ikke kan arve disse klassene fordi de inneholder stort sett sidebasert kode.

3.7.2 MemHash

MemHash er en indeksstruktur som kan brukes sammen med MemHeap for å gjøre direkteoppslag i tabelldata. Den ble implementert i `org.apache.derby.impl.store.-access.memhash`, og ble laget for å ha en enkel indeksstruktur til MemHeap. På grunn av at den bruker en hashtabell har MemHash bare støtte for eksakt søk, og ikke områdesøk slik som BTree, den eneste indeksstrukturen til Heap. Bakgrunnen for at vi valgte en hashbasert aksessmetode var å lage en enkel indeksert aksessmetode slik at vi enklere kunne sammenligne MemStore og RawStore.

Ved innsetting av en rad i en MemHeap-tabell med MemHash som indeksstruktur vil raden først innsettes i MemHeap. MemHeap returnerer hvor raden ble lagret ved hjelp av et `MemHeapRowLocation`-objekt, og MemHash lagrer så `RowLocation`-objektet i en hashstruktur med den indekserte nøkkelen som nøkkel.

Kapittel 4
Design og implementasjon

Vi vil her gå igjennom endringene som er gjort i forhold til den MemStore som vi utviklet i [Sol06], og hvilke nye utvidelser som er gjort. Utgangspunktet for implementeringa er Derby 10.2.2.0, siste stabile versjon når vi startet utviklinga. Koden for implementeringen vi har gjort ligger i det digitale vedlegget og på `http://issues.apache.org/jira/browse/DERBY-2798`.

4.1 Ny datastruktur i MemStore

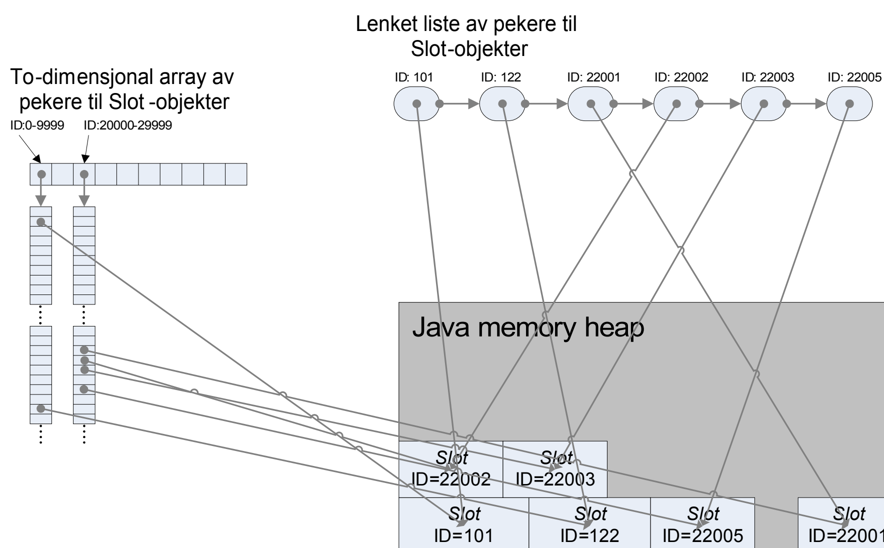
I MemStore-utvidelsen som vi utviklet høsten 2006 [Sol06] brukte vi bare en enkel array for å lagre referanser til Slot-objektene[1]. Dette var godt nok siden vi ikke hadde implementert noen form for sletting av elementer. Siden målet nå var å implementere sletting var det naturlig å gå bort i fra denne statiske måten å lagre elementene. Å bruke en array ville medført mye ekstraarbeid for å forhindre fragmentering og dårlige ytelse ved sekvensiell søking.

Datastrukturen i MemStore vil i hovedsak ha tre krav:

- sørge for raskt oppslag vha RowLocation-objekter som blir lagret i indeksen
- sørge for rask sekvensiell lesing
- den er subjekt for recovery, derfor må hvert Slot-objekt kunne adresseres direkte med slotID for å kunne gjenopprette radene i et sjekkpunkt til en konsistent tilstand under REDO- og UNDO-fasen

Vi valgte å bruke en lenket liste som den grunnleggende strukturen. Det første kravet løser vi på samme måte som tidligere ved å tillate at RowLocation-objektet inneholder en referanse til Slot-objektet. På denne måte trenger man ikke bruke den lenkede listen i

[1] Et Slot-objekt inneholder en rad i en tabell



Figur 4.1: Ny lagringsstruktur i MemStore

det hele tatt for oppslag, man følger bare referansen til objektet i minnet. En lenket liste vil også oppfylle det andre kravet ved å sørge for rask sekvensiell lesing, samtidig vil den håndtere sletting bra. Det tredje kravet er litt verre, en lenket liste støtter ikke direkte oppslag til en kjent posisjon. For at REDO- og UNDO-fasen ved recovery skal kunne operere på bestemte rader i minnet, må vi kunne finne Slot-objekter som skal endres raskt og effektivt. Vi lar derfor hvert Slot-objekt få en unik slotID, og en array-struktur av pekere som sørger for direkteoppslag mot objektene. Ved recovery vil Slot-objektene først bli hentet fra et eventuelt sjekkpunkt og lagret i den lenkede listen. Samtidig vil en peker til hvert Slot-objekt bli lagret i array-strukturen på en posisjon bestemt av slotID. REDO- og UNDO-fasen av recovery vil deretter bruke array-strukturen for direkteoppslag til Slot-objekter som berøres. Det samme gjelder transaksjonsrecovery, om det kjøres en *rollback* vil det genereres en *undo*-operasjon som sletter Slot-objektet den finner i arrayen med gitt slotID.

Lagerstrukturen er illustrert i figur 4.1. Av figuren kan vi se at selve dataene blir lagret i minneområdet som bare JVM[2] har kontroll over, vi har derfor ikke noen kontroll over den fysiske plasseringa av dataene i minnet. Ved innsetting og sletting må vi oppdatere *arrayen* i tillegg til den lenkede lista.

Sletting foregår ved å følge en *lazy deletion*-strategi som ofte er brukt i lenkede lister og tre-strukturer [CT76][Hor84][LPR93]. Denne strategien går ut på å bare markere nodene for slettet, og ved en senere anledning gå igjennom å slette nodene. Fordelen med dette er at sletting vil gå veldig fort, men vil påvirke ytelsen ved søk noe i tillegg vil ikke minnet bli like for tilgjengelig. I MemStore kjører vi en tråd som regelmessig sjekker om noe er slettet, og om dette er tilfelle løper den igjennom den lenkede listen og sletter noder som er markert for slettet. Samtidig slettes referansen i den to-dimensjonale tabellen.

Vi gjorde denne lenkede listen låsefri ved å bruke de samme teknikkene som Doug Lea[SM07] brukte i sin Skiplist-implementasjon. For å gjøre den kompatibel med Java

[2] Java Virtual Machine

1.4 brukte vi en synkronisert metode for CAS, denne kan med enkle grep endres til å bruke `AtomicReferenceFieldUpdater` senere.

4.2 Endring av eksisterende aksessmetode

Det ble også gjort mindre endringer i MemHeap angående låsing og mulighet for persistering. Endringene i låsingen innebærer at det nå settes en S-lås på tabellen når et søk bruker sekvensiell lesing ved hjelp av MemHeap. Dette gir mening for samme hva det søkes etter i MemHeap må det søkes igjennom hele tabellen. Det er derfor lite gunstig å låse hver rad for seg siden dette er en forholdsvis høy kostnad i Derby. Brukes indeksen MemSkiplist og det søkes etter en indeksert nøkkel vil den også benytte MemHeap til å hente dataene. Siden indeksen vet nøyaktig hvor dataene befinner seg trenger vi ikke låse hele tabellen, det settes da en IS-lås på tabellen og en S-lås på raden. Dette er vist i liste 4.1 der MemHeap får et RowLocation-objekt fra MemSkiplist. RowLocation-objektet vil her, i motsetning til RawStore inneholde en direktepeker til Slot-objektet med den aktuelle raden (linje 8).

```
1  public boolean fetch(RowLocation loc, DataValueDescriptor[] row,
2      FormatableBitSet validColumns) throws StandardException {
3    if (loc == null) {
4      return false;
5    }
6    MemHeapRowLocation memHeapRowLocation = ((MemHeapRowLocation) loc);
7    //use this slot-object for row based locking:
8    Slot2 slot = (Slot2) memHeapRowLocation.getRecordHandle();
9    boolean waitForLock = true;
10   open_conglom.lockSlotForRead(slot, waitForLock, open_conglom
11       .isForUpdate()); //Set S-lock on row, assumes IS-lock on table
12   DataValueDescriptor[] rowFromLocation = memHeapRowLocation
13       .getRecordHandleRow(); // get pointer to row in heap
14   int j = 0;
15   if (row[0] == null)
16     j = 1; //in this case the key in the index is used as row[0] and
          inserted by the join-operation
17   if (rowFromLocation != null) {
18     for (int i = 0; i < rowFromLocation.length; i++) {
19       if (validColumns.get(i)) {
20         row[j++] = rowFromLocation[i];
21       }
22     }
23     open_conglom.unlockSlot(slot, open_conglom.isForUpdate());
24     return true;
25   }
26   open_conglom.unlockSlot(slot, open_conglom.isForUpdate());
27   return false;
28 }
```

Liste 4.1: MemHeap.fetch(...), her blir data hentet ut fra et RowLocation-objekt som er hentet fra den sekundære indeksen MemSkiplist.

4.3 Ny aksessmetode

I MemStore har vi tidligere utviklet en hash-basert aksessmetode kalt MemHash. Hash-baserte aksessmetoder er effektive på direkteoppslag på indekserte nøkkelverdier, men de støtter ikke områdesøk. De er brukt av flere databasesystemer, men bare i sammenheng med andre indeksmetoder som støtter områdesøk. Det vil da være opp til optimalisereren å velge hvilken aksessmetode den skal bruke. Vi valgte en hash-basert aksessmetode fordi det var enklest å implementere, og fordi det var nok å ha en aksessmetode som bare støttet direkteoppslag for å sammenligne ytelsen til MemStore med RawStore.

I dette prosjektet ønsker vi å utvide funksjonaliteten til MemStore med en ny aksessmetode som støtter områdesøk. Vi valgte å erstatte MemHash med en ny aksessmetode fordi det ville medført unødvendig høy kompleksitet å støtte mer enn én indeksbasert aksessmetode, spesielt siden dette aldri har vært gjort i Derby.

Valget falt på SkipList som er en ny og ganske enkel datastruktur å implementere. Den er enkel å implementere fordi den ikke krever noen form for rebalanseringsalgoritmer og samtidig tilnærmet lik søkekostnad som B-trær. Vi ønsket også å bruke en variant som støttet høy samtidighet i et multi-tråd og multi-prosessormiljø.

Vi tok utgangspunkt i Skiplist-implementasjonen som finnes i Java 1.6, men siden Derby bør være kompatibel med versjoner fra 1.4 måtte vi modifisere den noe. Blant annet finnes ikke den atomiske CAS-instruksjonen i Java 1.4, vi erstattet derfor den med en synkronisert metode som gjorde det samme. Ytelsestestene viste at bruk av vår metode ikke var veldig mye tregere enn CAS-instruksjonen som tidligere antatt. Ønskes høyest mulig ytelse bør Java 1.6 med CAS brukes.

4.3.1 MemSkiplist

Den nye aksessmetoden kalte vi MemSkiplist og vi implementerte den i pakken `org.apache.derby.impl.store.access.memskiplist`. Den består av disse klassene:

MemSkiplist - Dette objektet korresponderer til en instans av en MemHeap- conglomerate, den mellomlagrer informasjon slik at det går fort å åpne (`MemSkiplistController` og `MemSkiplistScanController`) fra den. Den arver fra `GenericConglomerate` og implementerer interfacene `Conglomerate` og `StaticCompiledOpenConglomInfo` på lik linje med Heap. En serialisert versjon av dette objektet må lagres sammen med tilhørende MemContainer for at gjenoppretting av en MemStore-tabell skal lykkes.

MemSkiplistController - Ved innsetting blir denne klassen kalt for å lage et innslag i indeksen etter at raden har blitt lagret ved hjelp av `MemHeapController`. Denne fungerer på samme måte som MemHash ved at den kaller `RawTransaction.logAndDo(new MemSkiplistInsertOperation(..))` ved innsetting.

MemSkiplistCostController - Optimalisereren i Derby velger aksessmetode (MemHeap eller MemSkiplist) ut ifra kostnaden som denne klassen og tilsvarende klasse for MemHeap estimerer.

MemSkiplistFactory - Det er via denne klassen Derby får tilgang til denne aksessmetoden. Den blir registrert av RAMAccessManager under implementasjonsid `memskiplist`. Den fungerer som en factory [Coo00] for nye instanser av MemSkiplist.

MemSkiplistScanController - Denne klassen brukes for å finne en eller flere rader i indeksen. Den returnerer MemRowLocation-objekter som inneholder en minnereferanse direkte til Slot-objektet der hver enkelt rad ligger.

MemSkiplistStaticCompiledInfo - Denne klassen inneholder statiske data om en MemSkiplist-conglomerate som kan brukes om igjen ved senere utførelser. Dette vil stort sett si selve MemSkiplist-objektet og tilsvarende statiske data om conglomerate-objektet den er indeks for, her MemHeap.

Selve skiplist-algoritmen ligger i pakken `org.apache.derby.impl.store.access.-memskiplist.skiplist` og består av disse klassene:

- HeadIndex
- IndexNode
- Node
- SkipListEnumerator
- SkipListOptimistic

Skiplist-algoritmen er tatt ifra `ConcurrentSkiplistMap` som ligger i Java 1.6. Vi gjorde noen endringer i den for å gjøre den bakoverkompatibel med Java 1.4 og mer egnet for bruk i Derby. For det første finnes det ikke støtte for `AtomicReferenceFieldUpdater`, vi kan derfor ikke bruke CAS-instruksjonen (figur 4.2). For å kunne bruke denne implementasjonen i Java 1.4 brukte vi en synkronisert metode(figur 4.3) som gjorde det samme. Dette gir sannsynligvis en dårlige ytelse, men det medfører lite kode å endre og det gjør det enkelt å bruke AtomicReferenceFieldUpdater senere. Det fører også til mindre synkronisert kode enn om vi skulle brukt blokkerende kode slik som Pugh foreslo i sin versjon. For det andre støttes ikke *generics*[3] før i Java 1.5, kode som brukte dette måtte derfor endres.

I Derby arver alle datatypene fra klassen DataValueDescriptor, vi kan derfor erstatte alle *generics*-datatypene med denne klassen. Objekter av denne klassen har også logikk for å sammenligne verdier med hverandre, dette ble utnyttet i skiplist-algoritmen.

Vi trengte også en egen enumerator[4] som var mer databasespesifikk, vi utvikler derfor *SkipListEnumerator* som blant annet har støtte for de samme søkeoperatorene som Derby bruker.

Denne Skiplist-implementasjonen er basert på en enkeltlenket liste, i databasesammenheng er det også ofte ønskelig å kunne søke baklengs i indeksen. Vi utvidet derfor Skiplist-implementasjonen med en dobbellenket liste i bunnen med samme

[3]en måte å bestemme hvilke objekter en klasse skal benytte ved instansering av objektet.

[4]Grensesnitt for å hente ut data

```
1 static final AtomicReferenceFieldUpdater<Node,Node> nextUpdater =
      AtomicReferenceFieldUpdater.newUpdater(Node.class,Node.class, "next");
2
3
4 boolean casNext(Node<K,V> cmp, Node<K,V> val) {
5         return nextUpdater.compareAndSet(this, cmp, val);
6 }
```

Liste 4.2: Bruk av CAS-instruksjon i ConcurrentSkiplistMap

```
1 public boolean casNext(Node node, Node newNode) {
2         synchronized (this) {
3                 if (nextNode == node)
4                         nextNode = newNode;
5                 else
6                         return false;
7         }
8         return true;
9 }
```

Liste 4.3: Vår CAS-metode i MemSkiplist

indeksstruktur som før. Senere fant vi ut at Derby ikke kan utnytte baklengssøk i indekser fordi optimalisereren ikke har støtte for det. Vi beholdt derfor bare den enkeltlenkede listen. Bruk av f.eks `SELECT MAX(t1)` vil gå forholdsvis tregt siden optimalisereren ikke tar hensyn til baklengssøk, denne operatoren behandles derfor spesielt i Derby. Når MAX() brukes mot en indeksert kolonne kalles Conglomerate.fetchMaxOnBTree() istedenfor et vanlig søk, i RawStore brukes søkekontrolleren B2IMaxScan her, mens vi i MemStore valgte å lage en ny `getMax()`-metode i Skiplist som blir kalt gjennom enumeratoren vår. Denne løsningen er ikke helt optimal for noen av implementasjonene, men det forventes at Derby vil støtte baklengssøk i indekser ved senere versjoner.

Det er optimalisereren i Derby som bestemmer når MemSkiplist eller MemHeap skal brukes. Dette gjør den på grunnlag av informasjon om tilgjengelige indekser og kostnadsestimat gitt fra MemSkiplistCostController og MemHeapCostController. Vi må derfor sørge for at disse klassene gir riktig estimat. Målet er at spørringer av typen `SELECT * FROM TABLE` bør besvares med MemHeap, mens spørringer av typen `SELECT * FROM TABLE WHERE UNIQUE=?` besvares ved hjelp av MemSkiplist. Dette løste vi ved å la MemSkiplist estimere at det bare skal hentes én rad dersom spørringen setter start- eller stoppverdier. Estimeringsverdiene vi genererer her kan ikke direkte sammenlignes med Heap og BTree sine verdier siden de også tar hensyn til antall sideaksesser.

4.3.2 Implementering i Derby

For at Derby skal ta i bruk vår implementasjon av MemStore, må det gjøres noen endringer i koden. De nye aksessmetodene MemHeap og MemSkiplist blir registrert ved at de tilhørende ConglomerateFactory-objektene plugges inn i Derby. Dette gjorde vi ved å legge til filnavnene i *modules.properties* slik at de blir funnet av *Monitor*. For å legge til de to nye ConglomerateFactory-objektene redigerte vi `org.apache.derby.impl.store.-`

`access.RAMAccessManager`, her blir Monitor brukt for å lage en instans av objektene. Her blir også MemStore startet opp som en *system service*. RAMAccessManager har nå fått muligheten til å bruke de nye conglomerate-objektene, men siden RAMAccessManager ligger i access-laget må også sql-laget endres. Det er kun ved oppretting av nye conglomerate-objekter at vi trenger å spesifisere hvilken conglomerate det skal brukes. Dette skjer i `org.apache.derby.impl.sql.execute.CreateTableConstantAction` og `org.apache.derby.impl.sql.compile.TableElementList`, vi gjør dette ganske enkelt ved å sjekke om navnet på tabellen begynner med 'MEM_' så blir det laget en MemHeap eller MemSkiplist-conglomerate. Etter opprettingen vil det bare bli referert til Conglomerate-objektene via Conglomerate-ID'er som kan mappes direkte til en type conglomerate ved hjelp av en logisk operasjon.

Vi bruker Externalize-interfacet for serialisering, for at dette skal fungere må alle objektene som skal kunne serialiseres registreres under en unik FormatID i filene `org.apache.derby.iapi.services.io.RegisteredFormatIds` og `org.apache.derby.iapi.services.io.StoredFormatIds`.

I tillegg måtte vi også endre tilgangen fra `private` til `protected` i OpenConglomerate som MemOpenHeap og MemOpenSkiplist arvet fra. Grunnen til dette var at mye av koden i OpenConglomerate er side-spesifikk.

4.4 Logging

I den gamle MemStore implementerte vi bare en ikke-fungerende dummy-logg for å utføre ytelsestester med lik belastning som RawStore. En konsekvens av dette var at MemStore ikke hadde full transaksjonsstøtte fordi den ikke kunne abortere transaksjoner etter at de hadde startet. Årsaken er at Derby lagrer en serialisert versjon av operasjonene i loggen under utførelse, og ved en eventuell *rollback* blir de serialiserte operasjonsobjektene lest fra loggen igjen før hver operasjon angres. Vi må derfor få loggingen til å fungere for at MemStore skal få full støtte for transaksjoner.

Siden Derby bruker WAL-logging blir loggpostene skrevet til disken før en transaksjon er ferdig med *commit*-fasen. Som vi skrev i kapittel 2.4.1 blir det hevdet at LAW-logging er en bedre teknikk for MMDB-er fordi det gjør *REDO*-fasen av *recovery* enklere, men siden Derby uansett vil bruke WAL for RawStore-tabeller og en implementasjon av LAW ville medført mye ekstraarbeid bruker vi også WAL-logging for MemStore-tabeller.

Alle databaseoperasjoner i Derby kan oversettes til en eller flere `Operation`-objekter som på hver sin måte manipulerer datalageret. For at MemStore skal fungere på samme måte som RawStore, må alle operasjoner mot MemStore skje via tilsvarende operasjonsobjekter, der hvert objekt blir serialisert og klargjort for loggen før de utføres.

Loggeformatet (se figur 4.2) til Derby baserer seg på serialisering av operasjonsobjektene og kan derfor brukes uten endringer i MemStore. Operasjonsobjektene ble implementert i pakken `org.apache.derby.impl.store.mem.operations`, og består av:

- MemDeleteOperation
- MemDeleteUndoOperation

Type	Beskrivelse
int	format_id - denne verdien blir satt til LOG_RECORD av FormatIdOutputStream som en identifikator for hvor hvilket objekt som blir skrevet.
CompressedInt	loggable_group - brukes for å identifisere hvilken del av Derby logg-posten tilhører. RawStore har f.eks en egen loggable_group. MemStore bruker den samme idenfikatoren.
TransactionId	XactId - knytter logg-posten til en transaksjon.
Loggable	Op – Her lagres operasjonen.

Figur 4.2: Loggformat i Derby

- MemInsertOperation
- MemInsertUndoOperation
- MemUpdateOperation
- MemUpdateUndoOperation
- MemSkiplistDeleteOperation
- MemSkiplistDeleteUndoOperation
- MemSkiplistInsertOperation
- MemSkiplistUndoOperation

En innsetting av en indeksert post i MemStore vil bestå av disse stegene:

- Derby finner ut at posten tilhører MemHeap ved å undersøke conglomerate-id'en
- Finner den tilhørende indeksen MemSkiplist
- Setter inn raden i MemHeap vha MemHeapController.insertAndFetchLocation(..) i kodeliste 4.4. Det blir her generert et MemInsertOperation-objekt på linje 32 som brukes når Transaction.logAndDo(MemInsertOperation operation) kalles. Til slutt returneres posisjonen til posten ved at MemHeapRowLocation-objektet får en referanse RecordHandle-objektet til posten på linje 5.
- Lager indekspost ved å kalle MemSkiplist.insert(..) med posisjonen til den innsatte posten som argument. Her genereres det et MemSkiplistInsert-objekt og Transaction.logAndDo(MemSkiplistInsert operation) kalles igjen.

Når `Transaction.logAndDo(Loggable operation)` kalles blir operasjonen først skrevet til loggen i `FileLogger.logAndDo(..)` før den utføres ved å kalle `doMe()`-metoden på det samme operasjonsobjektet. I motsetning til RawStore kreves det ingen form for serialisering eller deserialisering av objekter under utførelse av `doMe()`-metoden, men for at recovery og transaksjonsrecovery skal fungere blir operasjonen serialisert når den logges. Skrivingen av operasjonsobjektet som må implementere Loggable-interfacet

```
1
2  public void insertAndFetchLocation(DataValueDescriptor[] row,
3      RowLocation templateRowLocation) throws StandardException {
4    RecordHandle rh = doInsert(row);
5    MemHeapRowLocation hrl = (MemHeapRowLocation) templateRowLocation;
6    hrl.setFrom(rh);
7    hrl.setFrom(open_conglom.getContainerId());
8  }
9
10 private RecordHandle doInsert(DataValueDescriptor[] row)
11     throws StandardException {
12   Conglomerate c = open_conglom.getConglomerate();
13   ContainerKey id = null;
14   if (c != null) {
15     id = c.getId();
16   } else {
17     id = new ContainerKey(0, 1);
18   }
19   ContainerHandle memcontainer = open_conglom.getContainer();
20   RecordHandle rh = null;
21   // Setter IX-lås på container:
22   xact.getDefaultLockingPolicy().lockContainer(xact, memcontainer, true,
23       true);
24   // Setter X-lås på rad:
25   do {
26     // loop until we get a new record id we can get a lock on
27     rh = memstore.newSlotAndBump(open_conglom.getContainer().getId()
28         .getContainerId());
29   } while (!xact.getDefaultLockingPolicy()
30       .lockRecordForWrite(xact, rh, true /* lock is for insert */,
31           false /* don't wait for grant */));
32   MemInsertOperation memInsertOperation = new MemInsertOperation(
33       open_conglom.getRawTran(), id, open_conglom.getContainer(),
34       row, rh, memstore);
35   rawTran.logAndDo(memInsertOperation);
36   return rh;
37 }
```

Liste 4.4: Innsetting av rad i MemHeap, der posisjon returneres for oppdatering i indeks.

gjøres ved at metoden writeExternal(outputstream) kalles på objektet. Dette er vist for MemInsertOperation i kodeliste 4.5.

Ved en eventuell *rollback* av transaksjonen må alle operasjonene som tilhørte transaksjonen hentes fra loggen. Hver operasjon som trenger å gjøre om på data i MemStore genererer da en ny undo-operasjon eller kompenserende operasjon i metoden `operasjon.generateUndo(..)`. Denne operasjonen blir utført på samme måte som vanlige operasjons-objekter. Det er imidlertid ikke mulig å angre en undo-operasjon, dette er i henhold til ARIES [MHL+92].

Det kan være ønskelig å bruke MemStore helt uten persistering mo disk. Dette kan gjøres ved å hindre loggeren i Derby i å skrive loggposter som tilhører MemStore til disk. Det er flere måter å gjøre dette på. En måte er å sjekke om en loggpost tilhører MemStore

```
1  public class MemInsertOperation extends MemOperation{
2    [...]
3    public void writeExternal(ObjectOutput out) throws IOException {
4      CompressedNumber.writeLong(out,id.getContainerId());
5      CompressedNumber.writeInt(out,slot.getSlotId());
6      CompressedNumber.writeInt(out,row.length);
7      for (int i=0;i<row.length;i++){
8        CompressedNumber.writeInt(out,row[i].getTypeFormatId());
9        row[i].writeExternal(out);
10     }
11   }
12   [...]
13 }
```

Liste 4.5: Skriving av MemInsertOperation i loggen

tilhører gruppen Loggable.MEMSTORE i `FileLogger.logAndDo(..)` og lagre disse i en egen loggstrøm før `Loggable.doMe()` utføres. På denne måten vil alle loggpostene som trengs for transaksjonsrecovery bli holdt i den nye loggstrømmen, mens loggposter som ikke tilhører MemStore blir skrevet til disk. Dette vil kreve endel endringer i loggeren i Derby.

En annen måte å bruke MemStore uten durability er å ikke lage loggposter for MemStore i det hele tatt. For at transaksjonsrecovery likevel skal fungere kan referanser til operasjonsobjektene legges i transaksjonsobjetet RawTransaction. Disse objektene trenger man bare dersom transaksjonen aborteres, da må operasjonsobjektene generere kompenserende operasjonsobjekter og utføre disse. Når transaksjonen commiteres fjernes referansen til RawTransaction, og *garbage collection* vil fjerne objektet og tilhørende operasjonsobjekter. Fordelen her er at vi ikke trenger å lage, serialisere og lagre loggpostene. Det krever heller ikke veldig store endringer i loggeren.

Et problem med begge framgangsmåter er at Derby lagrer informasjon om nye tabeller i sin Dictionary som bruker RawStore for å lagre data. Dette medfører at hvis vi lager en ny ikke-persistent minnebasert tabell, lagres dette i den persistente Dictionary i tillegg. Vi får da en feilmelding neste gang vi prøve å opprette den samme ikke-persistente tabellen. En løsning her er å la oppretting av tabeller være persistent også i MemStore, mens dataene som settes inn er ikke-persistent.

Ønsker man å logge til nabonoder over nettverk må man lage og sende loggpostene istedenfor å forkaste de. Nabonodene kjører sin egen LogFactory som må kunne benyttes ved gjenoppretting av databasen.

4.5 Recovery i MemStore

Recovery vil si å gjenopprette dataene i databasen til en konsistent tilstand. I hovedsak er det to tilfeller hvor dette er nødvendig, når en transaksjon aborteres og når databasesystemet, operativsystemet eller datamaskinen krasjer. Det første tilfellet som blir kalt transaksjonsrecovery innebærer å angre alle endringer gjort av den aktuelle transaksjonen. I det andre tilfellet må hele databasen gjenopprettes til en konsistent

ved oppstart, dette kalles krasj-recovery. Vi vil her gå igjennom hvordan vi løste dette i MemStore.

4.5.1 Transaksjonsrecovery i MemStore

Transaksjonsrecovery behøves når en transaksjon har gjort endringer i databasen og aborterer ved at det utføres en *rollback*. Dette er det transaksjonsmanageren i Derby som tar seg av, den holder styr på hvilke transaksjoner som kjører og hvilke operasjoner representert av Loggable-objekter den har utført. Når en transaksjon aborterer vil transaksjonsmanageren hente Loggable-objektene fra loggen og kaller Loggable.generateUndo() på hvert av de. Denne metoden returnerer et kompanserende operasjonsobjekt for hver av de opprinnelige operasjonene som vil gjenopprette dataene når de utføres. Utførelsen og loggingen skjer på samme måte som for andre operasjoner, eneste forskjell er at disse kompenserende operasjonene ikke kan kompenseres.

I MemStore bruker vi disse operasjonensobjektene for å manipulere data: `MemInsertOperation`, `MemUpdateOperation` og `MemDeleteOperation`. For hvert av disse har vi også laget en kompanserende operasjon, kalt `MemInsertUndoOperation`, `MemUpdateUndoOperation` og `MemDeleteUndoOperation`. På bakgrunn av den unike SlotID vil disse kompenserende operasjonene slå opp i MemStore på den gitte SlotID og endre tilbake dataene. For manipuleringer av indekser brukes samme fremgangsmåte, operasjonene `MemSkiplistInsertOperation` og `MemSkiplistDeleteOperation` har sine kompanserende operasjoner som på bakgrunn av den indekserte nøkkelen endrer tilbake dataene. Oppdateringer av indekserte kolonner har vi ikke implementert støtte for.

4.5.2 Sjekkpunkt

For å gjøre gjenoppretting av en databasen raskere kjører Derby som de fleste andre databaser regelmessige sjekkpunktalgoritmer. Dette skjer i FileLogger.checkpoint() og suksessfullt sjekkpunkt markeres i loggen ved å utføre operasjonsobjektet kalt `CheckpointOperation`. Sjekkpunktsalgorimen består i hovedsak av å flushe datasider som ligger i bufferet og lagre nok informasjon til at det senere er mulig å bringe databasen til en konsistent tilstand. På grunn av at MemStore har et eget minnebasert lager og ikke er avhengig av et slikt buffer må dette behandles spesielt. Vi måtte derfor lage en egen sjekkpunktalgoritme for MemStore.

Et annet viktig mål med sjekkpunktalgoritmen er at den skal påvirke ytelsen til databasen minst mulig mens den kjører. Vi laget derfor en algoritme som kopierte data fra minnet og lagret til disk helt asynkront med det som ellers skjedde i databasen. Utgangspunktet for sjekkpunktalgoritmen er "fuzzy checkpoint"-algoritmen for MMDB introdusert av Hagmann [Hag86].

Sjekkpunktalgoritmen er implementert i MemStoreImpl.checkpoint(), og består i all enkelhet ut på å serialisere MemStoreImpl som igjen serialiserer all data i databasen ved hjelp av Externalize-interfacet. I metoden MemStoreImpl.writeExternal(..) kan vi spesifisere hvilke deler av objektet som skal serialiseres. Det eneste vi serialiserer er et array med kontainere representert ved MemContainer-objekter. På samme måte som i

Derby samsvarer en kontainer med et conglomerate-objekt og eventuelle tilhørende data. Tilhører kontaineren en MemHeap-conglomerate serialiseres selve MemHeap-objektet og alle tilhørende Slot-objektene. Om den tilhører en MemSkiplist-conglomerate serialiseres bare MemSkiplist-objektet, selve skiplist-strukturen lagres ikke.

Hvis et sjekkpunkt blir avbrutt av et systemkrasj står vi igjen med et ufullstendig sjekkpunkt på disk og må bruke forrige sjekkpunkt for gjenoppretting av databasen. Det er derfor viktig at et sjekkpunkt ikke overskriver det forrige sjekkpunktet, vi bruker derfor to ulike sjekkpunkt som vi veksler på å bruke på samme måte som K. Salem bruker i sin "ping-pong"-metode [SGM87]. Til forskjell fra K. Salem lagrer vi ikke objektene i faste blokkstørrelser i minnet, det er derfor vanskeligere å bare skrive ut poster som er endret siden sist sjekkpunkt. Vår sjekkpunktalgoritme skriver isteden hele innholdet ut på disk, siden dette skjer helt asynkront vil dette bare i begrenset grad påvirke ytelsen.

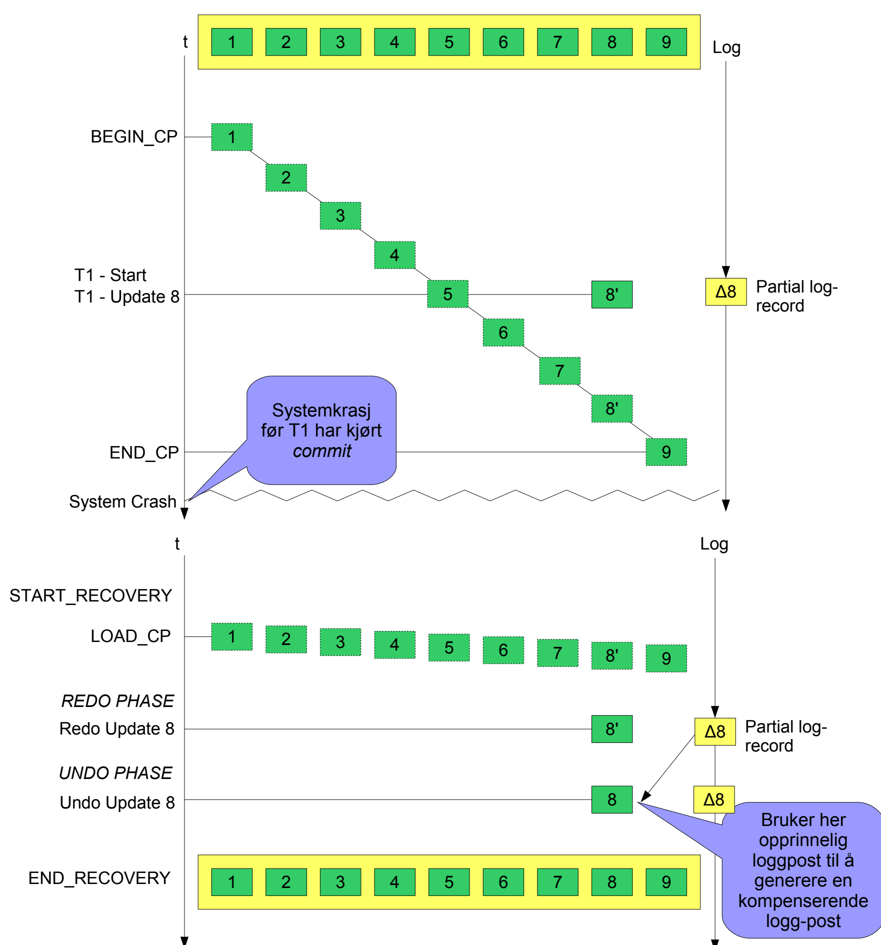
I tillegg for å sørge for rask gjenoppretting bør sjekkpunktet også bruke lite ressurser slik at resten av databasen ikke blir skadelidende. Vi lar derfor sjekkpunktalgoritmen vente i ett millisekund for hvert 10. slot-objekt som serialiserer, dette medfører at sjekkpunktingen tar ca 20% lengre tid men det vil påvirke resten av systemet mindre.

4.5.3 Gjenoppretting ved krasj

MemStore bruker et minne-basert lager, gjenoppretting av databasen må derfor skje hver gang databasen startes. Om databasen blir avbrutt uten å gjennomgå en avslutningsprosedyre med en tilhørende sjekkpunkt-operasjon, medfører dette merarbeid ved oppstart av databasen neste gang fordi større deler av loggen må inspiseres og operasjoner gjentas. Det er derfor ekstra viktig å avslutte databasen på riktig måte.

Skrivinga av sjekkpunktet skjer helt asynkront med aktiviteten i databasen, vi har derfor ingen måte å sikre oss konsistente data. Strategien i "fuzzy checkpoint"-algoritmer er å overlate dette problemet til redo- og undo-fasen av gjenopprettinga. Vi har illustrert et eksempel i figur 4.3, der post 8 blir endret etter at sjekkpunktet har begynt, men før posten er skrevet til sjekkpunktet. Bare før- og etterkopi av endringen blir skrevet til loggen, f.eks en kolonne i en rad. Systemet krasjer før transaksjonen har kommitert, det er her avgjørende at sjekkpunktsalgoritmen selv flusher alle logg-poster fordi en transaksjon ikke nødvendigvis gjør dette før den er ferdig.

Ved neste oppstart av databasen begynner gjenopprettingen umiddelbart. Her blir først siste komplette sjekkpunkt lastet inn, så kjøres det en REDO-fase på alle loggposter etter starten av sjekkpunktet. Her blir endringen av post 8 gjentatt, selv om endringen allerede finnes i minnet. Dette går bra siden operasjonsobjektene våre er idempotent. Etter REDO-fasen kjører Derby en UNDO-fase der alle endringer gjort av kjørende transaksjoner ved systemkrasjet blir angret. Her må endringene gjort av T1 angres, det gjøres ved at loggposten lastes inn og operasjonsobjektet hentes ut. På dette objektet kalles metoden `generateUndo(..)`. Det genereres da et nytt "Undo"-operasjonsobjekt som loggføres på vanlig måte før man kaller `doMe(..)`-metoden. Siden loggposten bare inneholder før-kopi av den ene kolonnen må det antas at den fullstendige posten allerede finnes i minnet. Dette er alltid tilfelle om vi har et fullstendig sjekkpunkt. Selv om post 8 slettes før sjekkpunktet rekker å lagre den, har operasjonsobjektet for slettingen lagret en før-kopi(BFIM i figur 4.4) av posten i loggen. Dette er illustrert i figur 4.4.



Figur 4.3: Gjennoppretting av MemStore etter systemkrasj

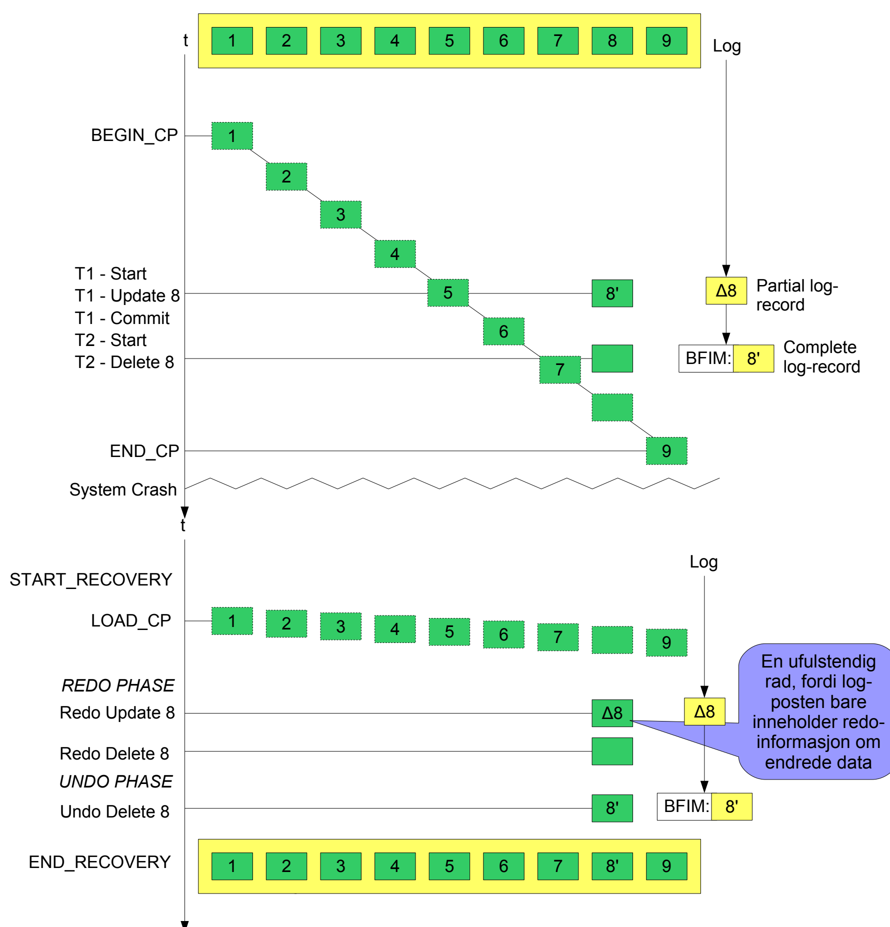
Ved utvikling av FCP[5]-algoritmer er det som tidligere nevnt i kapittel 2.3.2 viktig å ikke bryte med WAL-prinsippet. Vi løser dette ved å bruke to sjekkpunkter som i "ping-pong"-metoden til K. Salem [SGM87]. Når et sjekkpunkt er skrevet kan dette sjekkpunktet inneholde data som enda ikke er loggført. Vi venter derfor med å markere sjekkpunktet for gyldig før alle loggpostene laget før sjekkpunktet er ferdig blir flushet til disken. Dette gjøres ved å flushe alle loggposter med LSN[6] lavere enn sist utførte operasjon etter at sjekkpunkt-operasjonen er loggført. Dersom sjekkpunktet blir avbrutt av et systemkrasj mens det inneholder data som ikke er loggført er ikke sjekkpunktet gyldig og det forrige sjekkpunktet brukes til gjenoppretting isteden.

4.5.4 Implementering i Derby

For å gjenbruke mest mulig av Derby-koden, kjører vi vår sjekkpunktsalgoritme av MemStore samtidig som Derby kjører sin sjekkpunktalgoritme. Derby utfører sitt sjekkpunkt i `org.apache.derby.impl.store.raw.log.LogToFile.checkpoint()`, der

[5] *fuzzy checkpoint*

[6] Log Sequence Number - Dette er kalt LogInstant i Derby og identifiserer en loggpost



Figur 4.4: Gjennoppretting av MemStore etter systemkrasj

blir det generert et CheckpointOperation-objekt som logger et suksessfullt sjekkpunkt. Vi trigger vår sjekkpunkting av MemStore ved å kalle `MemStore.checkpoint()` i `org.apache.derby.impl.store.raw.log.LogToFile.checkpoint()` metoden. Dette skjer hver gang logg-volumet øker med 10 MB, om ikke annet er spesifisert. I LogToFile.checkpoint() må vi også sørge for å flushe loggfiler mot disken før sjekkpunktet ferdigstilles for å unngå å bryte med WAL-prinsippet.

Gjenopprettingen skjer i `org.apache.derby.impl.store.raw.log.LogToFile.recover()`, her kaller vi `MemStore.recover()` før REDO- og UNDO-fasen starter. REDO-fasen i Derby består av å gjenta loggførte operasjoner som ble utført etter starten av sjekkpunktet. Siden MemStore bruker det samme loggesystemet som RawStore til å logge sine operasjoner, trenger vi ikke gjøre noen endringer her annet enn å implementere operasjons-objektet som sørger for at de både kan loggføres og hentes ut fra loggen igjen. Det samme gjelder UNDO-fasen, her er det nok å implementere undo-operasjoner som blir instansiert om `generateUndo()`-metoden blir utført på et av operasjons-objektene.

Derby logger og gjenoppretter indekser på samme måte som data, i MemStore har vi derimot valgt å ikke lagre indeksene i sjekkpunktet og heller generere de på nytt etter at databasen er startet. Dette krever mindre ressurser enn å serialisere og lagre alle nodene

i skiplist-strukturen og de-serialisere nodene igjen etterpå.

Etter REDO og UNDO-fasen er databasen i minnet transaksjonskonsistent, det tas da umiddelbart et nytt sjekkpunkt. Dette gjør at databasen slipper å utføre en tilsvarende REDO og UNDO-fase ved neste oppstart, og logg-filene kan trygt fjernes for å spare disk-plass.

Kapittel 5
Tester og målinger

Å teste databasesystemer har vist seg å være en betydelig utfordring. Grunnen til dette er at de er veldig komplekse og ytelsen ved ulik arbeidsbelastning kan vise seg å gi helt forskjellige resultater. Et system kan være optimalisert for batch-transaksjoner[1] andre for interaktive transaksjoner mens andre igjen bare er optimalisert for lesing. Det er derfor utarbeidet flere ulike benchmark-tester for å kunne sammenligne flere databaser for en gitt type belastning. Vi vil her benytte en benchmark som er løst basert på Wisconsin Benchmark [DeW93].

5.1 Oppsett

Derby ble bygget etter beskrivelsen i 10.2.2.0 med Java 1.4.6. Målingene er gjort mot Derby i embedded-modus, vi gjorde dette fordi overhead på nettverksutgaven vil være tilnærmet lik for både MemStore og RawStore. Datamaskinen vi brukte var en Pentium4 3 GHz med 512 MB RAM som kjørte Debian Linux 2.6.16-smp. Vi utførte målingene med isolasjonsnivået satt til serialisering (RR), ellers var Derby satt opp slik:

`derby.storage.logArchiveMode=true`
Dette hindrer sletting av loggfiler underveis. Vi ønsker å hindre mest mulig unødvendig IO-arbeid under utføring av målingene

`derby.storage.pageCacheSize=1000000`
Dette sørger for at alle sider beholdes i minnet slik at all data i RawStore blir minneresident.

[1] Transaksjoner som behandler data i store blokker

`derby.storage.pageSize=8192`
Dette angir sidestørrelse til 8 KB.

`derby.storage.logSwitchInterval=5000000`
Tillater Derby å begynne på ny loggfil først etter 5 GB, dette hindrer skifte av loggfil under testing noe som kan påvirke ytelsesmålingene av RawStore og MemStore ulikt.

`derby.storage.checkpointInterval=100000000`
Dette hindrer at sjekkpunktalgoritmen starter under testing, dette kan også påvirke ytelsesmålingene ulikt. Effekten av sjekkpunkting er målt for seg selv i kapittel 5.5.

Hardisken på testmaskinen har også aktivert skrive-cache, dette vil medføre at systemet ikke kan oppfylle kravet for *durability* i ACID-egenskapene [GR93]. Samtidig øker dette ytelsen betraktelig og tiden dataene bli behandlet i "Data Store"-laget blir enklere å måle. Vi benytter samme loggemekanisme i både MemStore og RawStore. Loggekostnadene ved de to vil derfor være tilnærmet like. Om vi slår av skrive-cachen på disken må hver transaksjon vente på at loggpostene som er generert og blir flushet til disken. Dette medfører ca 6-8 ms forsinkelse på moderne disker.

5.2 Testdata

```
CREATE TABLE TENKTUP1
(   unique1     integer NOT NULL PRIMARY KEY,
    unique2     integer NOT NULL,
    two         integer NOT NULL,
    four        integer NOT NULL,
    ten         integer NOT NULL,
    twenty      integer NOT NULL,
    hundred     integer NOT NULL,
    thousand    integer NOT NULL,
    twothous    integer NOT NULL,
    fivethous   integer NOT NULL,
    tenthous    integer NOT NULL,
    odd100      integer NOT NULL,
    even100     integer NOT NULL,
    stringu1    char(52) NOT NULL,
    stringu2    char(52) NOT NULL,
    string4     char(52) NOT NULL
)
```

Figur 5.1: Tabell i Wisconsin benchmark

For å generere testdata som er i tråd Wisconsin Benchmark-spesifikasjonene bruker vi klassen `org.apache.derbyTesting.functionTests.tests.lang.WISCInsert` som er å finne blant Derby sine funksjonelle tester. Den originale Wisconsin Benchmark bestod av tre tabeller, en med 1000 rader (kalt ONEKTUP) og to andre med 10000 rader (kalt TENKTUP1 og TENKTUP2). Hver relasjon består av 13 integer-verdier og tre tekststrenger på 52 byte, i figur5.1 er SQL-Create-setningen for tabellen gitt. Den totale størrelsen på en rad utgjorde da 208 bytes forutsatt ingen overhead. Vi benytter en tabell

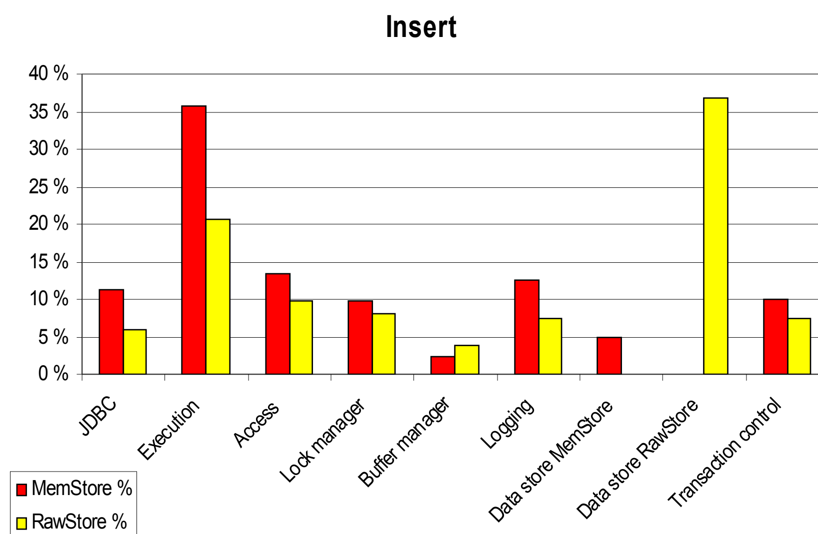
med 100000 rader kalt HUNDREDKTUP, en tabell med 10000 rader kalt TENKTUP og en med 1000 rader kalt ONEKTUP.

5.3 Fordeling av CPU-tid

Lag	Derby pakke
Network server	`org.apache.derby.drda` og `org.apache.derby.impl.drda`
JDBC	`org.apache.derby.jdbc` og `org.apache.derby.impl.jdbc`
Execution	`org.apache.derby.impl.sql`
Access	`org.apache.derby.impl.store.access`
Lock manager	`org.apache.derby.impl.services.locks`
Buffer manager	`org.apache.derby.impl.services.cache`
Logging	`org.apache.derby.impl.store.raw.log`
Data store	`org.apache.derby.impl.store.raw.data` og `org.apache.derby.impl.store.mem`
Transaction control	`org.apache.derby.impl.store.raw.xact`

Tabell 5.1: Lagoppdelingen i Derby [AD06]

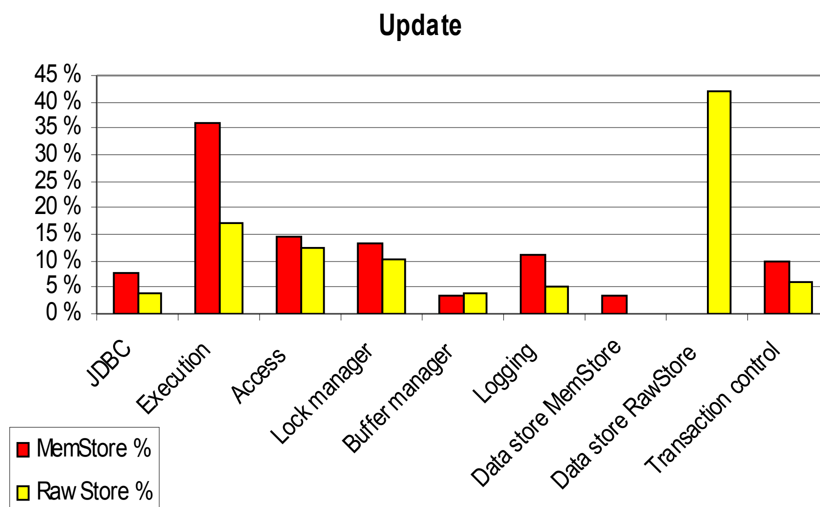
For å finne ut hvilke lag i Derby som bruker mest ressurser i MemStore kontra RawStore brukte vi NetBeans Profiler [Inc06] for å måle forbrukt CPU-tid i de ulike pakkene i Derby. Pakkestrukturen i Derby representerer lagdelingen som vist i tabell 5.1.



Figur 5.2: Graf som viser hvordan CPU-tiden er fordelt i de ulike lagene i Derby ved innsetting

Figur 5.2 viser prosentvis CPU-fordelingen når en klient kjører kontinuerlig innsetting av Wisconsin-rader. For Derby med RawStore ligger flaskehalsen i DataStore der lagring og organisering av dataene i side-objekter skjer. For Derby med MemStore ligger

flaskehalsen i Execution-laget ettersom Data Store-laget bruker her mye mindre CPU-tid. Execution-laget er lik for både MemStore og RawStore og innebærer *parsing*, kompilering og eksekvering [AD07] av SQL-spørringene. Vi ser også at Buffermanager bruker en del CPU-tid når vi bruker MemStore, dette skyldes at Derby mellomlagrer Derby *dictionary*-data og Conglomerate-objektene her.



Figur 5.3: Graf som viser hvordan CPU-tiden er fordelt i de ulike lagene i Derby ved oppdateringer

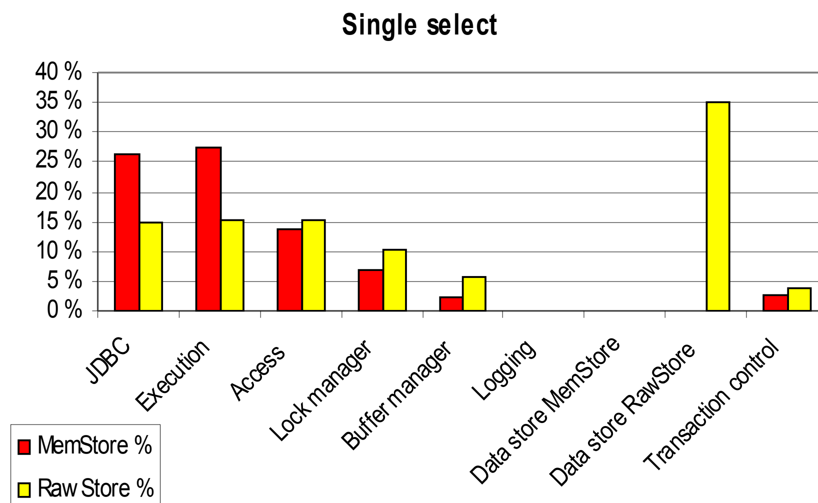
For *update* i figur 5.3 ser vi at Derby med RawStore bruker en enda større andel av CPU-tiden i DataStore enn ved *insert*. MemStore bruker veldig lite CPU-tid i DataStore, noe som også her fører til at flaskehalsen er flyttet til Execution-laget. Loggingen bruker her noe mindre CPU-tid fordi bare endringer av rader blir loggført.

Samme trenden kan vi se for *select* i figur 5.4, her utfører én klient kontinuerlige spørringer mot primærnøkkel mot Wisconsin testdata. MemStore eliminerer også her flaskehalsen for RawStore. Årsaken til at CPU-forbruket i MemStore er lik null er at vi her kjører oppslag mot primærnøkkelen som er indeksert med MemSkiplist. Hvert innslag i MemSkiplist inneholder en peker til hele raden som MemHeap bruker til å hente raden istedenfor å hente den fra MemStore.

5.4 Aksessmetoder

5.4.1 Skiplist

Som utgangspunkt for den nye aksessmetoden MemSkiplist brukte vi ConcurrentSkiplist-Map som er implementert i Java 1.6. Denne benytter den atomiske instruksjonen CompareAndSwap (CAS) som bare er å finne fra Java 1.5. Vi bør derfor ikke bruke denne i Derby. Vi modifiserte derfor algoritmen slik at den brukte et synkronisert metodekall mot en metode som gjorde det samme som CAS-instruksjonen. I figur 5.5



Figur 5.4: Graf som viser hvordan CPU-tiden er fordelt i de ulike lagene i Derby ved søk på primærnøkkel

har vi målt ytelsen på de ulike algoritmene mens vi kjørte de på forskjellige JVM'er. Soft-CAS vil si at algoritmen benytter en synkronisert metode for CAS, blokkerende vil si at vi ikke benytter CAS i det hele tatt men isteden synkroniserer metodene for skrivetilgang. Hw-CAS vil si at det benyttes en hardware-instruksjon for CAS representert ved AtomicReferenceFieldUpdater.

Målingene er gjort ved at ulike klienter kjører 60% lesing, 20% innsetting og 20% oppdateringer mot datastrukturen. Vi måler her ytelsen på vår implementasjon av som bare har små endringer i forhold til ConcurrentSkiplistMap. Den er kompilert med JDK 1.4, 1.5 og 1.6. Resultatene viser at bruk av CAS gir en gjennomgående bedre ytelse enn en blokkerende fremgangsmåte. Soft-CAS har en mye høyere ytelse i Java 1.5 og 1.6 i forhold til 1.4 med neste tre ganger så høy throughput. Bruk av hardware-CAS i Java 1.5 gir overaskende en noe dårligere ytelse enn soft-CAS, mens det i Java 1.6 gir hardware-cas en noe bedre ytelse igjen. Forklaringen her kan være at i Java 1.5 ble ikke hardwareinstruksjonen brukt av JVM-en vi kjørte, eller så er den ikke optimalisert for ytelse.

5.4.2 MemSkiplist og MemHeap

Vi vil her kjøre de forskjellige spørringene gitt i kapittel 5.4.2 mot Derby, og sammenligne ytelsen ved bruk av MemStore og RawStore. Klientene kjører kontinuerlige spørringer mot tabellene. Før målingene starter kjøres det ett minutt oppvarming for å stabilisere ytelsen, dette sørger også for at alle sidene i RawStore ligger i buffermanager.

Spørringene vi har valgt å måle ytelsen på er et utvalg av spørringene gitt i Winsconsin Benchmark [DeW93] og lister de opp her:

- Enkeltsøk: `SELECT * FROM hundredktup WHERE unique1 = <INTEGER>`

Figur 5.5: Ytelsestester for de ulike implementaringene av Skiplist kjørt i ulike versjoner av Java Virtual Machine

- Områdesøk: SELECT * FROM hundredktup WHERE unique1 BETWEEN <INTEGER> and <INTEGER>
- Join: SELECT * FROM hundredktup,tenktup WHERE hundredktup.unique1 = tenktup.unique1
- Update: UPDATE hundredktup SET unique2=<INTEGER> WHERE unique1=<INTEGER>
- Insert: INSERT INTO hundredktup values (<INTEGER>, <INTEGER>, <INTEGER>, <INTEGER>, <INTEGER>, <INTEGER>, <INTEGER>, <INTEGER>, <INTEGER>, <INTEGER>, <INTEGER>, <INTEGER>, <INTEGER>, <VARCHAR>, <VARCHAR>, <VARCHAR>)

Enkeltsøk

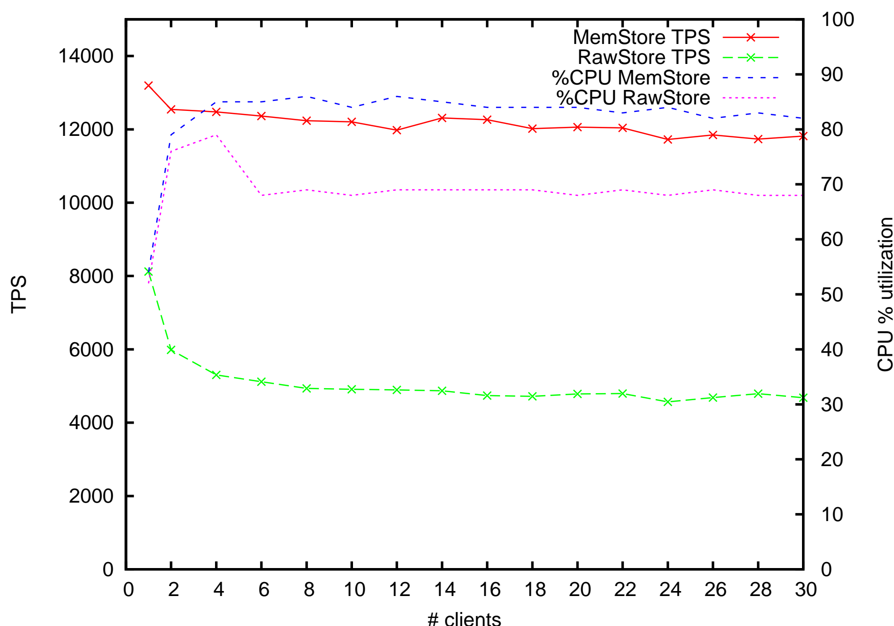
Her måler vi ytelsen på denne spørringen:

SELECT * FROM hundredktup WHERE unique1 = <INTEGER>

Den henter alle kolonner fra en indeksert rad. Vi illustrerte resultatet av målingen i grafen i figur 5.6. Y-aksen viser totalt antall fullførte transaksjoner per sekund. Grafen viser at MemStore har ca 70% høyere throughput enn RawStore med én klient kjørende, og over det dobbelte med flere enn én klient. Bidraget til ytelsesøkningen her ser vi av figur 5.4 er mindre tid brukt i DataStore, BufferManager og LockManager.

Årsaken er at dataene allerede er tilgjengelige som objekter, man trenger ikke sette latcher på sider i buffermanager og deserialisere dataene før de presenteres for klientene. Prosessorutnyttelsen er noe høyere for MemStore, dette skyldes sannsynligvis mindre ventetid i låsemanager fordi MemStore ikke bruker latcher.

Vi utførte også den samme spørringen mot den ikke-indekserte kolonnen `unique2`. Her blir ytelsen til Heap i RawStore og MemHeap i MemStore testet, i begge tilfellene vil hver transaksjon sette en S-lås på tabellen. Resultatet i figur 5.7 viser at MemStore har opp til tre ganger høyere ytelse med flere enn én klient og i underkant av dobbelt så høy ytelse med én klient.



Figur 5.6: Søking på primærnøkkel i Wisconsin-testdata mot Derby i embedded-modus.

Områdesøk

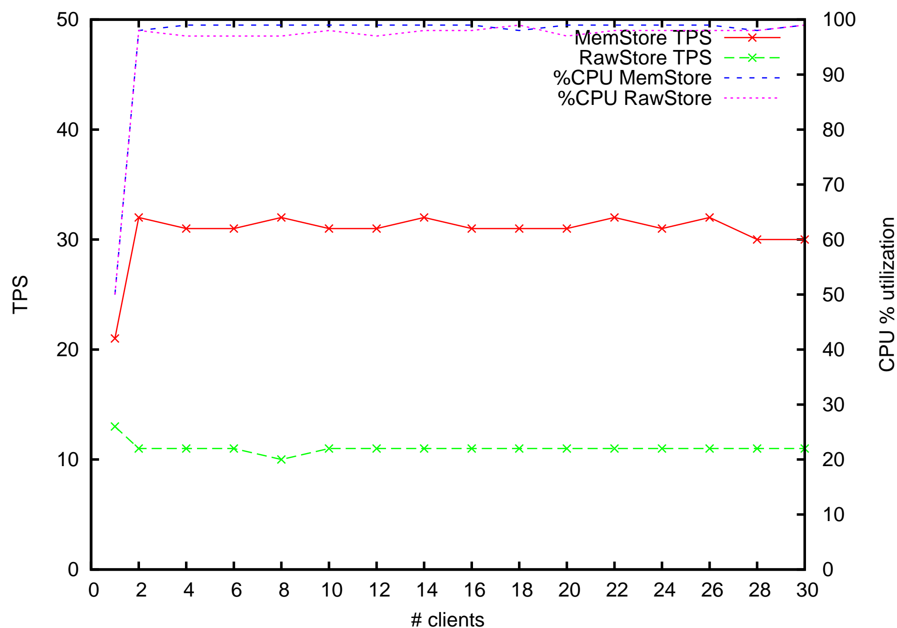
For å måle ytelsen på områdesøk bruker vi denne spørrigen:

```
SELECT * FROM hundredktup WHERE unique1 BETWEEN <INTEGER> and <INTEGER>
```

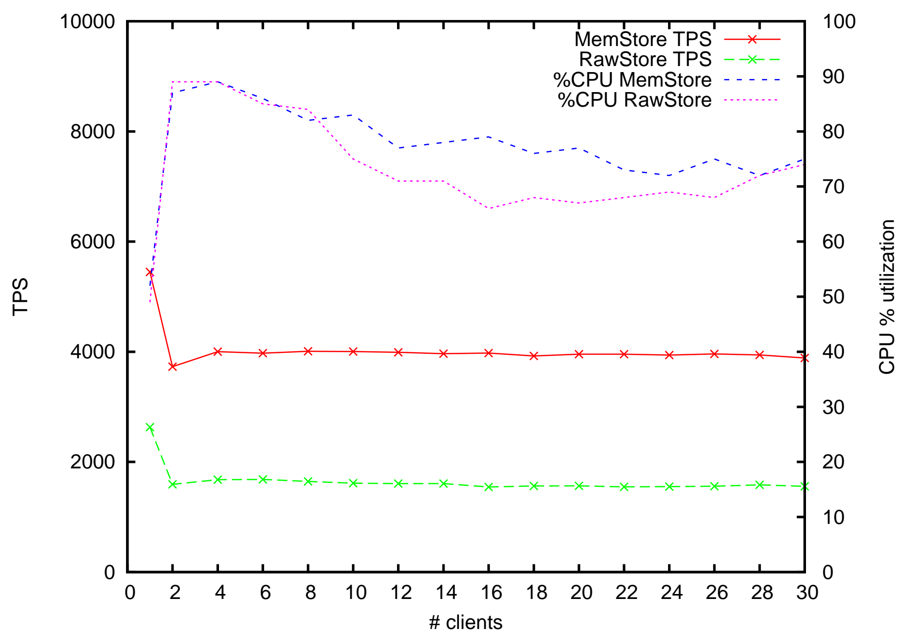
Den henter alle kolonner fra indekserte rader som ligger mellom to verdier. Hver spørring vil returnere ca 20 rader. Grafen i figur 5.8 viser ganske likt resultat som for enkeltsøk. MemStore utfører ca dobbelt så mange transaksjoner per sekund.

Join

I figur 5.9 viser vi ytelsesresultatet når vi utførte *join* av tabellene `onedktup` bestående av 1000 rader og `tenktup` bestående av 10000. Optimalisereren i Derby valgte her å utføre join ved hjelp av *hashjoin* istedenfor gjentatte gjennomløp fordi dette ga minst kostnad. Kostnaden beregnes ut fra hva hvert Conglomerate-objekt antar det koster å



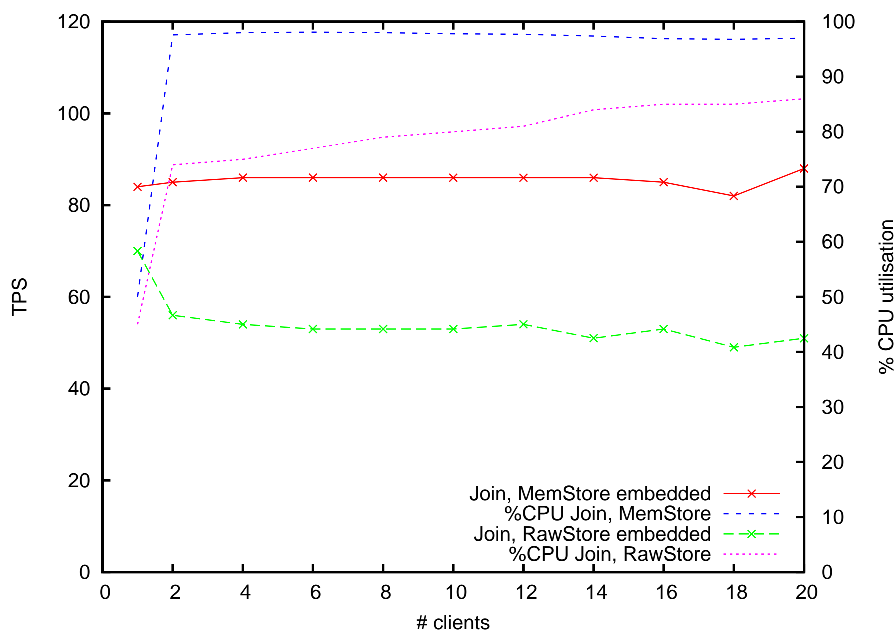
Figur 5.7: Søking på ikke-indeksert nøkkel i Wisconsin-testdata mot Derby i embedded-modus.



Figur 5.8: Områdesøk på ikke-indeksert nøkkel i Wisconsin-testdata mot Derby i embedded-modus.

hente relevante data. Dette er en relativ kostnad som i RawStore er basert på hvor mange Page-objekter som må hentes og hvor mange rader som hentes. I MemStore består kostnaden bare av antall rader og noe indekseringsoverhead. Det optimale her hadde vært en metode basert på *merge-join* fordi begge kolonnene er indeksert, men

dette støttes dessverre ikke i Derby. For MemStore er det også diskutabelt hva som er best av gjentatte gjennomløp eller hashjoin. Gjentatte gjennomløp ville krevd flere minneaksesser men mindre prosessering og minne, mens hashjoin hadde krevd færre minneaksesser men mer prosessering og minne. Resultatene viser at MemStore er ca 20% raskere på hashjoin med én klient, og ca 70% raskere med flere klienter.



Figur 5.9: Wisconsin sin join-operasjon utført ved hashjoin i MemStore og RawStore i embedded-modus.

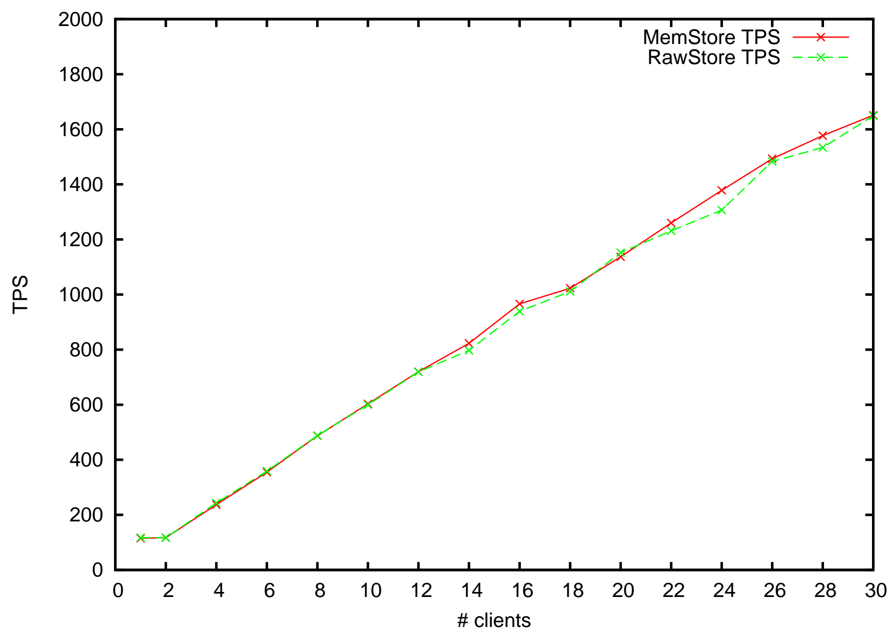
Update

For *update* kjørte vi spørringen:

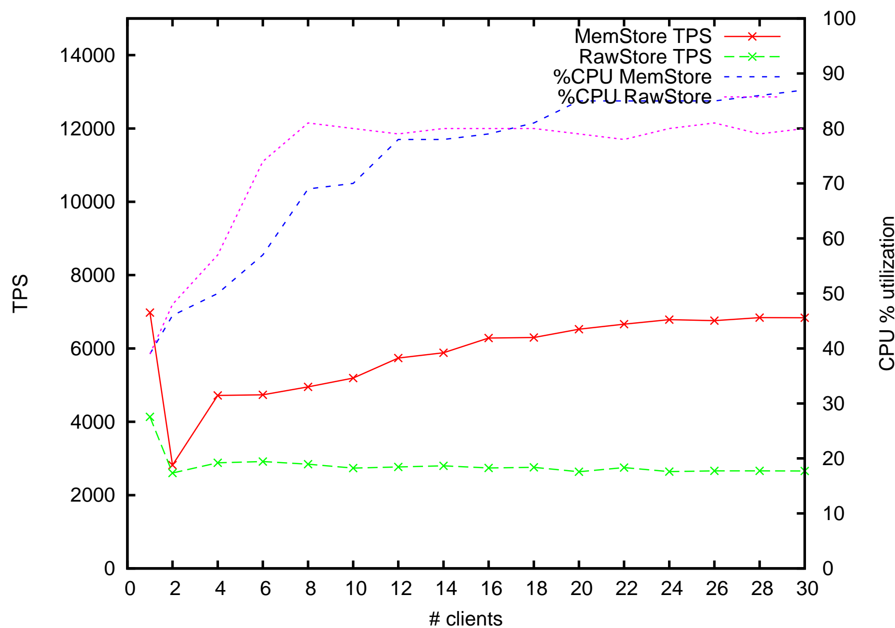
```
UPDATE hundredktup SET unique2=<INTEGER> WHERE unique1=<INTEGER>
```

Denne henter en rad ved hjelp av indeksen og endrer en ikke-indeksert kolonne. Det blir i både MemStore og RawStore satt IX-lås på tabell og X-lås på den aktuelle raden. På grunn av at begge bruker samme loggemekanisme forventer vi ikke mye forskjell i ytelse med skrive-cache deaktivert. Resultatene av målingen illustrert i figur 5.10 bekrefter dette. I snitt må hver transaksjon i *commit*-fasen vente i ca 8ms før loggpostene er skrevet, dette utgjør ca. 96% av tiden for hver transaksjonen. For å måle forskjell i ytelsen til selve DataStore-laget aktiverte vi skrive-cachen. Dette medfører at det durability ikke kan garanteres fordi loggpostene først blir skrevet til et flyktig buffer på harddisken før de blir lagret i stabilt lager. Resultatet som gitt i figur 5.11 viser at ved én klient er ytelsen ca. 70%, og mellom 70% og 100% med flere enn to klienter. Ved to samtidige klienter er ytelsen tilnærmet lik, det er uvisst hva dette skyldes men kan være på grunn av schedulering-problematikk i forbindelse med låsing eller logging.

Ytelsesøkningen som ble påvist ved bruk av MemStore vil bare gjøres gjeldende når skrive-cache er aktivert, når det logges til annet medium med mindre forsinkelse eller når det ikke logges i det hele tatt. Et annet medium kan for eksempel være ikke-flyktig minne eller flyktig minne på nabo-noder over nettverk på uavhengige krasjdomener på samme måte som i Clustra [HTBH95].



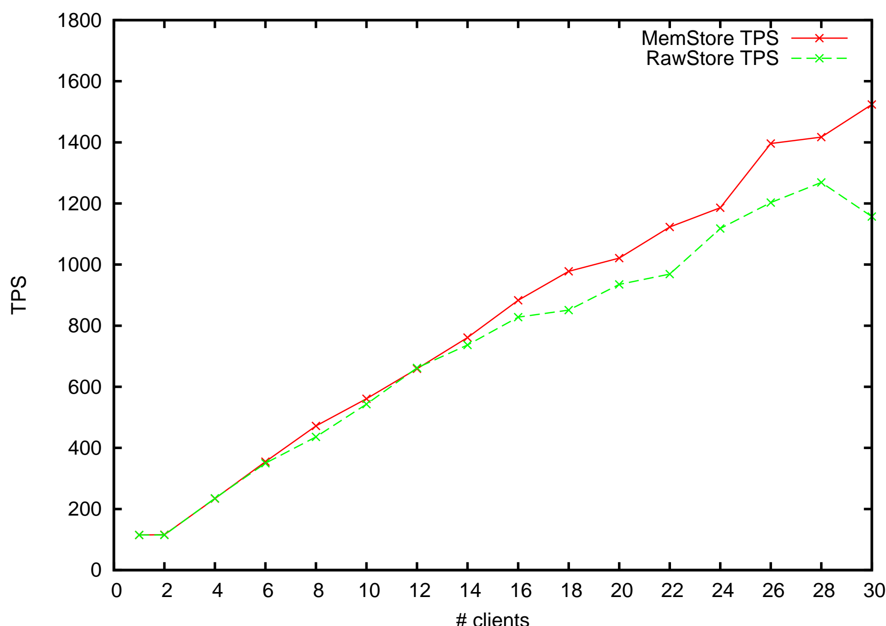
Figur 5.10: Update med skrive-cache deaktivert.



Figur 5.11: Update-operasjon mot Wisconsin-testdata i MemStore og RawStore i embedded-modus. Skrive-cache er aktivert.

Insert

For innsetninger fikk vi samme resultat som for *update*, nemlig at loggekostnadene dominerer såpass mye at ytelsen vil bli den samme for MemStore og RawStore når skrive-cache var deaktivert på harddisken (figur 5.12). Vi aktiverte derfor skrive-cache på disken og utførte *insert*-setningen gitt i i kapittel 5.4.2 mot en ikke-indeksert tabell med ulike antall klienter. Resultatene som vist i figur 5.13 viser at heller ikke her er forskjellene veldig store ved 1-2 klienter. MemStore har her rundt 20% høyere throughput. Det er først med flere enn 2 klienter at forskjellene blir større. Her har MemStore mellom 60% og 100% høyere throughput.



Figur 5.12: *Insert* med skrive-cache deaktivert.

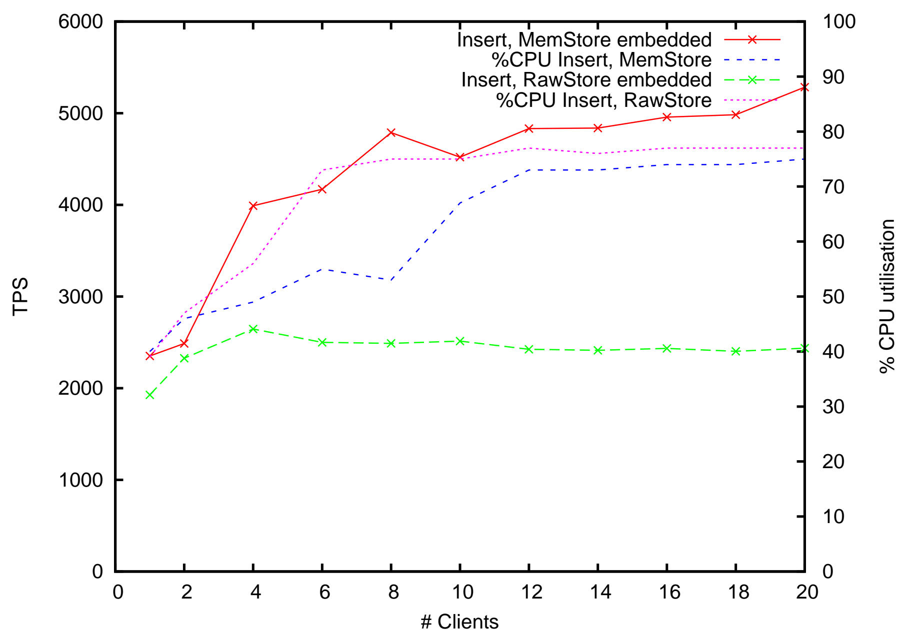
5.5 Sjekkpunkt-algoritmen

For å måle hvordan sjekkpunktsalgoritmen påvirker resten av systemet, målte vi throughput for en klient som kontinuerlig kjørte denne *select*-setningen samtidig som vi startet et sjekkpunkt:

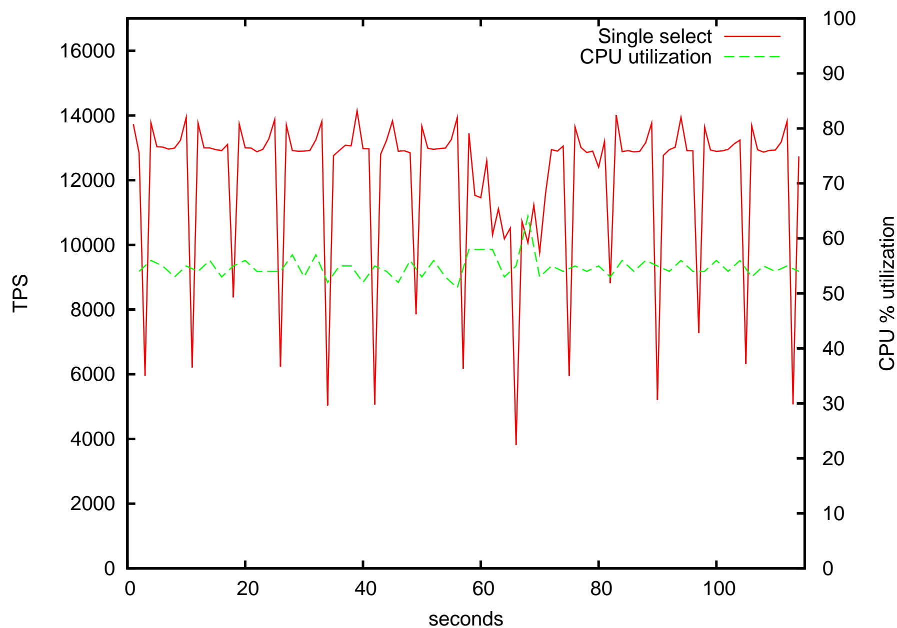
```
SELECT * FROM hundredktup WHERE unique1 = <INTEGER>
```

Målingene er plottet i figur 5.14. Sjekkpunktet av MemStore med 100 000 rader ble startet etter 55 sekunder og varte i ca 10 sekunder. Gjennomsnittet for throughput går ned med ca 17% mens sjekkpunktinga pågår. Denne degraderinga kan minskes ytterligere ved å gi sjekkpunktsalgoritmen enda lavere prioritet. CPU utnyttelsen øker bare marginalt.

Ut fra grafen ser vi også en tydelig dropp i throughput ca. hvert 15. sekund. Dette skyldes at én eller noen få transaksjon plutselig tar over 4000 microsekunder, noe som er 70-80



Figur 5.13: Innsetting av Wisconsin-testdata mot Derby i embedded-modus. med skrive-cache er aktivert.

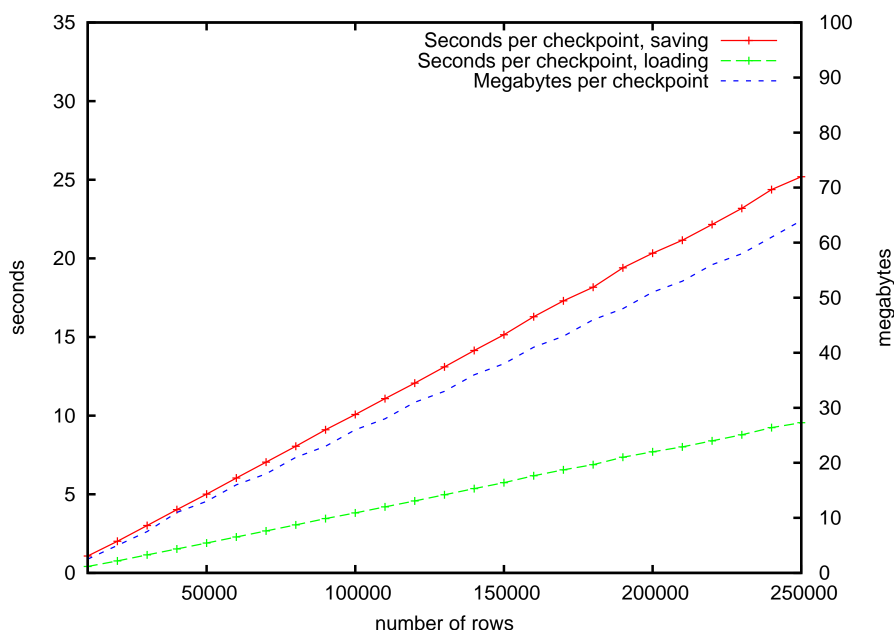


Figur 5.14: Grafen viser hvordan ytelsen til databasen påvirkes av et sjekkpunkt. Starter sjekkpunktet etter ca. 70 sekunder, og det varer i ca 20 sekunder.

ganger lengre enn medianen som er 56 microsekunder. Kjørte vi samme måling med GC deaktivert var droppene borte. Mye tyder derfor på at årsaken er at *garbage collection* (GC) i JVM-en kjører i dette tilfellet hvert 15. sekund og degraderer ytelsen i cirka ett sekund. Grunnen til at GC bruker så mye ressurser er at den fjerner ubrukte objekter

i heapen. Det er uvisst om det er implementasjonen av Derby eller vår implementasjon av MemStore som skylder dette. Dersom vi prøver å gjenbruke så mange objekter som mulig istedenfor å allokerere nye vil det medføre mindre arbeid for GC-en. Oppførselen til GC-en kan i tillegg styres med diverse opsjoner for å minimere påvirkningen på systemet.

I figur 5.15 har vi illustrert hvor stort sjekkpunktet blir i megabytes og hvor lang tid det tar å generere og laste inn. Vi ser at størrelsen og tiden det tar å behandle sjekkpunktet er helt lineært i forhold til hvor mange rader som ligger i minnet. Utgangspunktet for dataene er testdataene i Wisconsin-benchmark.



Figur 5.15: Grafen viser tiden det tar å utføre et sjekkpunkt og størrelsen det tar avhengig av antall rader.

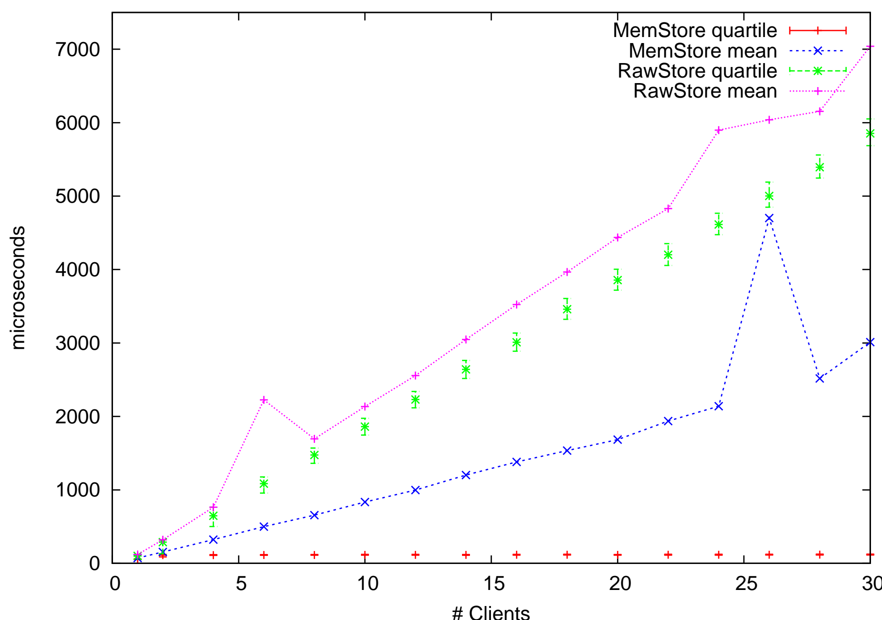
5.6 Responstid

I tillegg til throughput ønsket vi også å finne ut hvordan responstidene fordelte seg. Vi målte derfor responstiden for *select*-spørringen under ved å måle tiden hver transaksjon brukte på å utføre spørringen og hente ut dataene.

```
SELECT * FROM hundredktup WHERE unique1 = <INTEGER>
```

Spørringen ble utført 100 000 ganger mot RawStore og MemStore, målingene ble gjentatt for 1 til 30 klienter. Resultatet i figur 5.16 viser fordelingen av responstiden ved hjelp av kvartiler og gjennomsnittlig responstid. Under målingen av sjekkpunktalgoritmen fann vi ut at GC-en degraderte ytelsen på noen få transaksjoner veldig mye. Dette forklarer at gjennomsnittlig responstid er mye høyere enn median-verdien, fordi GC får mer og mer å gjøre og forsinker enkelte transaksjoner ytterligere ettersom antall klienter øker. Dette er spesielt merkbart for MemStore der medianen nesten er helt konstant, mens gjennomsnittet øker lineært.

Vi ser også noen topper i gjennomsnittlig responstid for 6 og 24 samtidige klienter for RawStore og 26 klienter for MemStore. Årsaken her er ukjent, men siden toppene opptrådte helt tilfeldig når vi målte flere ganger antar vi det for å være forsinkelse forårsaket av låsekonflikter.

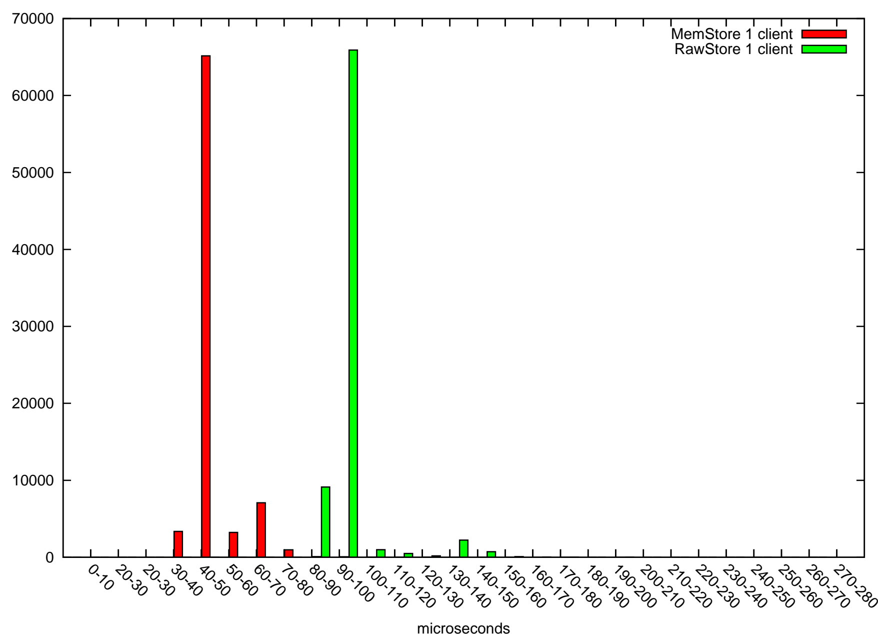


Figur 5.16: Responstid for *select*, viser fordeling av responstidene ved hjelp av kvartiler.

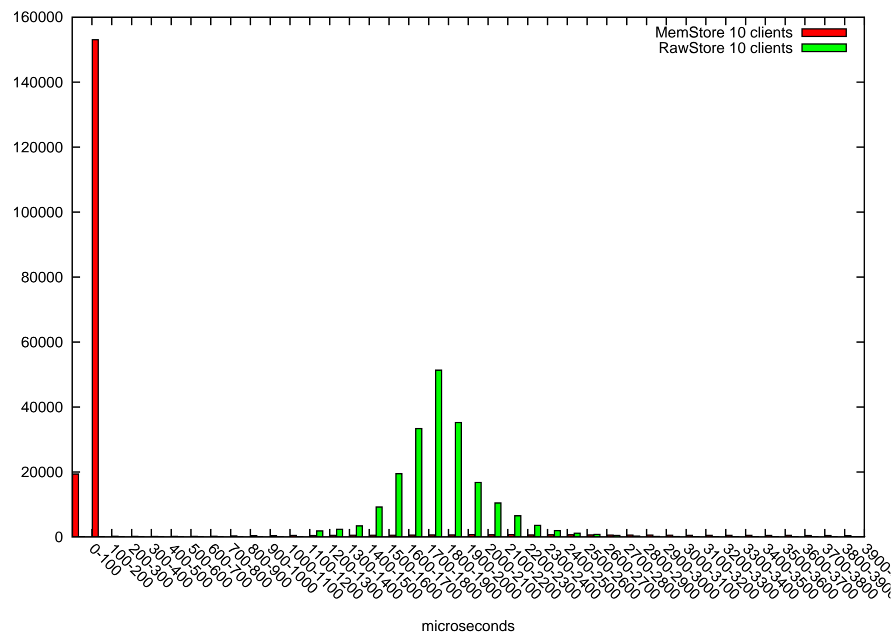
I figur 5.17 har vi illustrert fordelingen av responstidene for én klient. Her ser vi at de for både MemStore og RawStore er ganske jevnt fordelt, gjennomsnittet for MemStore er ca 56% av gjennomsnittet for RawStore. Dette stemmer godt overens med grafen for througput i figur 5.6. For 10 klienter i figur 5.18 er gjennomsnittet for MemStore 43% av gjennomsnittet for RawStore. Her ser vi tydelig at responstidene for RawStore er tilnærmet normalfordelt rundt 1800 microsekunder mens for MemStore har vi en stor andel på rundt 84% som er under 150 microsekunder mens resten er mer spredd utover. Vi antar at denne spredningen er grunnet påvirkning av GC og låsekonflikter på grunnlag av tidligere funn under måling av sjekkpunktsalgoritmen. Resultatene her kan tyde på at RawStore bruker mindre tid i GC enn MemStore.

5.7 Minneforbruk

Vi har tidligere antatt at MemStore vil bruke mer minne en RawStore. For å finne ut hvor mye mer minne MemStore bruker målte vi minneforbruket ved hjelp av NetBeans Profiler [Inc06] mens vi initialiserte databasen med Winsconsin testdata. Figur 5.19 og 5.20 viser hvordan minneforbruket utviklet seg mens vi satte inn 111 000 rader. Etter å ha fullført innsettinga av alle radene ser vi at MemStore har brukt ca 150 MB med minne, mens RawStore har brukt ca 100 MB. Trekker vi fra ca 40 MB som Derby og testapplikasjonen bruker før den laster inn databasen er minneforbruket til MemStore her

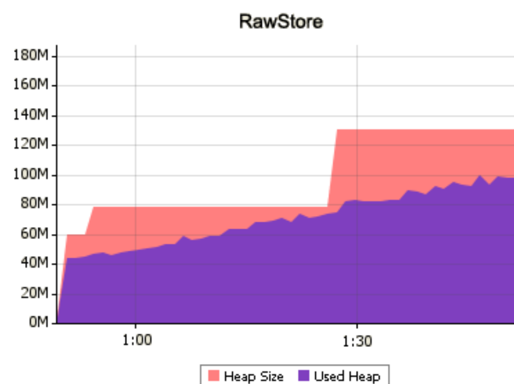


Figur 5.17: Histogram over responstid for en klient som kjører kontinuerlige *select*-setninger mot primærnøkkel.

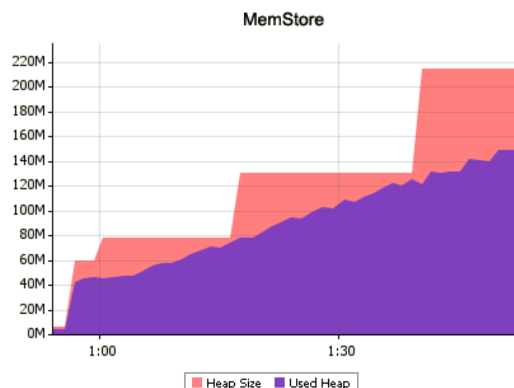


Figur 5.18: Histogram over responstid for 10 samtidige klienter som kjører kontinuerlige *select*-setninger mot primærnøkkel.

cirka 80% høyere enn RawStore. Det samme minneforbruket ble påvist etter at dataene ble lastet inn fra sjekkpunktet på 29 MB. En rad i MemStore inkludert indeks vil da legge beslag på ca 990 bytes mens den i RawStore vil bruke ca 540 bytes, i sjekkpunktet til MemStore vil den bruke ca 260 bytes men uten indeks.



Figur 5.19: Minneforbruk i Derby ved innsetting av Wisconsin-data i RawStore

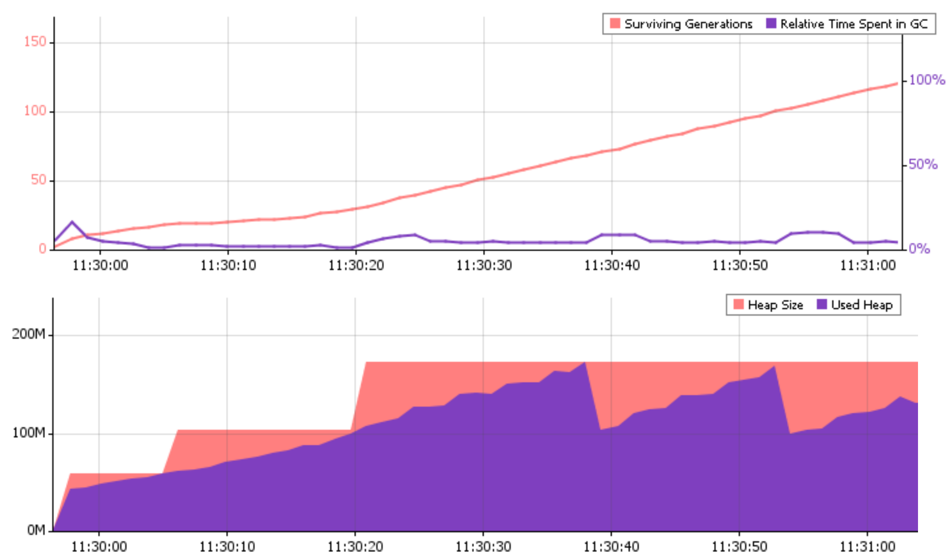


Figur 5.20: Minneforbruk i Derby ved innsetting av Wisconsin-data i MemStore

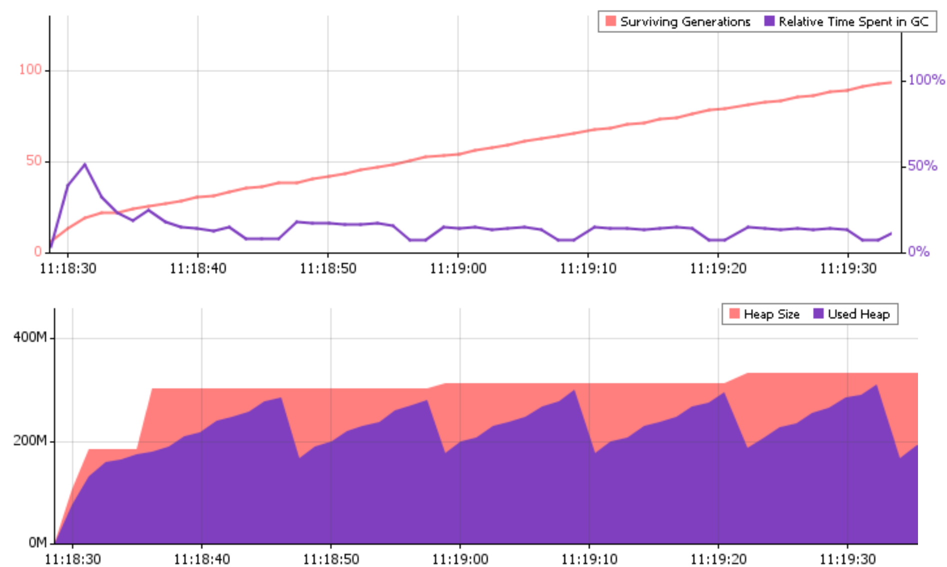
Det høye minneforbruket kan skyldes flere ting, i MemStore blir en rad beholdt i instansiert tilstand i minnet istedenfor å bli serialisert og delvis komprimert før de lagres som et byte-tabell i et Page-objekt slik som i RawStore. I MemStore lagres også hver rad i sitt eget Slot-objekt, dette medfører også mer minnebruk på grunn av overhead som klassen gir.

I tillegg til selve datastrukturen består også implementeringen vår av aksessmetodene MemHeap og MemSkiplist. For å få en indikasjon på minneforbruket her utførte vi målinger på *select*-setningen med 10 samtidige klienter i 60 sekunder. Resultatet i figur 5.22 og 5.21 viser at GC bruker ca 5% av all cpu-tiden i RawStore og ca 15% i MemStore. Det er tydelig at MemStore bruker mer minne på å utføre samme operasjon enn RawStore, MemStore bruker jevnt over 100 MB mer minne enn RawStore. GC kjører også ca dobbelt så ofte i MemStore, men tatt i betrakning av throughput i MemStore er dobbelt så høyt som i RawStore er ikke dette urimelig. Det som er verdt å merke seg er at MemStore bruker nesten tre ganger så mye tid i GC som RawStore. Dette kan tyde på at aksessmetodene i MemStore kan gjøres mer effektive ved at det initialiseres et mindre antall objekter og at de som initialiseres brukes om igjen. På en annen side er det færre overlevende generasjoner i MemStore enn i RawStore, dette betyr at til enhver tid er et færre antall objekter med forskjellig alder i RawStore. Med alder mener vi her

hvor mange ganger et objekt har overlevd en GC.



Figur 5.21: Minneforbruk og bruk av *garbage collection* ved søking i RawStore



Figur 5.22: Minneforbruk og bruk av *garbage collection* ved søking i MemStore

Kapittel 6
Konklusjon

Vi har gjennom dette prosjektet tatt utgangspunkt i MemStore som vi utviklet høsten 2006. Dette var en eksperimentell prototype av en minnebasert lagermodul for Derby. Denne implementeringen inneholdt kun et minimum av funksjonalitet for å få utført ytelsestester. Det meste av denne koden ble skrevet om og ny funksjonalitet lagt til. Den nye MemStore-utvidelsen består av en lagermodul med tilhørende aksessmetoder som tillater persistens ved hjelp av Derby sin loggemekanisme og en egen sjekkpunktsalgoritme. Utvidelsen bestod av å erstatte RawStore sin datalagermodul med en ny modul som vi har kalt MemStore. Denne forenkler lagringen av data en hel del ved å beholde tabelldataene som instansierte objekter i minnet. I tillegg til dette laget vi to conglomerate-objekter kalt MemHeap og MemSkiplist som fungerer som aksessmetoder mot MemStore. MemSkiplist er en indeksert aksessmetode basert på SkipList. Implementeringen bestod av ca 6000 kodelinjer som grovt sett gjorde samme jobben som ca 30000 kodelinjer i RawStore og tilhørende aksessmetoder.

Gjennom arbeidet har vi vist at arkitekturen til Derby gjør det enkelt å utvide systemet med nye moduler. Vi har også lykkes i å bruke eksisterende mekanismer for logging og låsing slik at MemStore kan tilby radbasert låsing og persistens av data. Ytelsestestene viste en betydelig ytelsesøkning på *select*-spørringen på fra ca 70 - 110% avhengig av antall samtidige klienter. For *update* og *insert* er ytelsen den samme om vi kjører uten skrive-cache på harddisken fordi vi bruker samme loggemekanisme for RawStore. Dersom vi aktiverer skrive-cachen får vi imidlertid økt ytelsen med 25-100% avhengig av antall klienter. Vi har også identifisert hvordan loggingen kan deaktiveres for MemStore mens RawStore samtidig har full persistens. Her kan det også lages en mekanisme som sender loggpostene til nettverksnoder istedenfor til disk.

Kapittel 7
Videre arbeid

MemStore er fortsatt en ganske uferdig utvidelse til Derby, det er derfor flere ting som bør utbedres eller implementeres. Videre utvikling bør gå i retning av å gi MemStore en mer komplett funksjonalitet. Det bør også fokuseres på å utvikle en effektiv måte å lagre data i minnet uten noen form persistens da det har vært etterspørsel etter dette i epostlistene til Derby-dev [Fou07] tidligere.

Vi har her listet forslag til hva det bør arbeides mer med:

En ny sjekkpunktsalgoritme som bare kopierer skitne data mot disken er ønskelig. Et problem her er at Slot-objektene som inneholder data for en rad har variabel lengde. Det er derfor vanskelig å oppdatere et Slot-objekt om oppdateringen medfører mer data. Dette kan f.eks løses ved å sette en terskelverdi for størrelsen for Slot-objektene, og lagre de sekvensielt i en fil med én terskelverdi mellom hvert serialisert objekt. Om terskelverdien overstiges må den økes og filen må gjenoppbygges, terskelverdien kan eventuelt settes til maks-størrelsen for Slot-objektet. For å begrense gjenoppbyggingen kan objektene lagres i flere filer. En annen måte er å lagre det nye Slot-objektet et annet sted på disken og invalidisere den gamle.

Distribuert log istedenfor logging mot disk er ønskelig for å bedre skrive-ytelsen i MemStore. Loggekostnadene ved *insert* og *update* utgjør ca 96% av transaksjonen når databasen kjører uten skrivebuffer.

Låseeskalering fungerer fortsatt ikke fordi objektet vi låser data på ikke implementerer ContainerKey. Vi foreslo at Slot også skulle implementere ContainerKey i [Sol06], men dette vil medføre enda mer overhead på data.

Mindre overhead på data i minnet er ønskelig da det ble påvist endel høyere minneforbruk enn i RawStore.

Konfigurerbar låsepolitikk bør implementeres. Nå er alt hardkodet til isoleringsnivå Serializable.

Flere indekser på samme tabell. Nå støtter MemStore bare en indeks som opprettes samtidig som tabellen opprettes, det er ønskelig å kunne opprette flere indekser og å kunne manipulere disse senere.

Forlengs og baklengssøk i indeksen støttes ikke i MemStore, dette gjør heller ikke Derby 10.2.2.0.

Gjøre MemStore klar for Derby 10.3 og Java 1.6 , i dette prosjektet ble MemStore utviklet for 10.2.2.0 og Java 1.4. Det er usikkert hvor mye arbeid det er å tilpasse MemStore for Derby 1.3, men MemSkiplist er veldig lett å optimalisere for Java 1.6.

Studere skalering og hvordan ressursene fordeles blant klientene. Dette er spesielt interessant for skiplist-algoritmen som bruker en låsefri mekanisme for samtidighetskontroll.

Undersøke utnyttelse av prosessor-cache ved Skiplist-algoritmen. Finne ut hvordan lokalitetsprinsippet utnyttes i Skiplist i forhold til andre aksessmetoder.

Opsjon for MemStore ved oppretting av nye tabeller .

Bibliografi

[AB06a] MySQL AB. *Chapter 15. MySQL Cluster*, 2006. MySQL 5.0 Reference Manual
http://dev.mysql.com/doc/refman/5.0/en/mysql-cluster.html.

[AB06b] MySQL AB. *MySQL Cluster FAQ*, 2006. MySQL 5.0 Reference Manual
http://dev.mysql.com/doc/refman/5.0/en/mysql-cluster-faq.html.

[AD06] Db-derby Wiki Apache Derby. Resource usage in derby. webpage, 2006. http://wiki.apache.org/db-derby/Derby1961ResourceUsage2.

[AD07] Db-derby Wiki Apache Derby. How derby works. webpage, 2007. http://wiki.apache.org/db-derby/HowItWorks.

[BGMS97] Bouge, Gabarro, Messeguer, and Schabanel. Concurrent rebalancing of AVL trees: A fine-grained approach. In *EUROPAR: Parallel Processing, 3rd International EURO-PAR Conference*. LNCS, 1997.

[BK95] P. Boncz and M. Kersten. Monet: an impressionist sketch of an advanced database system. In *Proceedings of Basque Int'l Workshop on Information Technology*, July 1995.

[Bon02] P. A. Boncz. *Monet: A Next-Generation DBMS Kernel For Query-Intensive Applications*. Ph.d. thesis, Universiteit van Amsterdam, Amsterdam, The Netherlands, May 2002.

[Cer06] Lavinio Cerquetti. Resource management in apache derby. https://twiki.informatik.tu-chemnitz.de/pub/DVS/SeminarDBMSImplementation/Ressourcenverwaltung_Derby.pdf, 2006.

[CLK+06] Seunglak CHOI, Jinwon LEE, Su Myeon KIM, Junehwa SONG, and Yoon-Joon LEE. Accelerating Database Processing at Database-Driven Web Sites. *IEICE Trans Inf Syst*, E89-D(11):2724–2738, 2006.

[Coo00] James W. Cooper. *Java Design Patterns: A Tutorial*. pub-AW, 2000.

[CT76] David Cheriton and Robert Endre Tarjan. Finding minimum spanning trees. In *Journal of Computing*, pages 724–742, December 1976.

[ddml06] derby-dev@db.apache.org mailing list. Releasing latches when waiting for locks. when and why? mailing list, 2006. http://thread.gmane.org/gmane.comp.apache.db.derby.devel/33135.

[Dera] Apache Derby. Derby engine architecture overview. http://db.apache.org/derby/papers/derby_arch.html.

[Derb] Apache Derby. Derby sources, 10.1 branch. https://svn.apache.org/repos/asf/db/derby/code/branches/10.1/.

[Der06a] Apache Derby. Derby write ahead log format. website, 2006. http://db.apache.org/derby/papers/logformats.html.

[Der06b] Apache Derby. Derby's cost-based optimization, 2006. http://db.apache.org/derby/manuals/tuning/perf45.html.

[Der06c] Apache Derby. Isolation levels and concurrency, 2006. http://db.apache.org/derby/docs/dev/devguide/cdevconcepts15366.html.

[DeW93] David J. DeWitt. The wisconsin benchmark: Past, present, and future. In Jim Gray, editor, *The Benchmark Handbook for Database and Transaction Systems (2nd Edition)*. Morgan Kaufmann, 1993.

[Eic86] Margaret H. Eich. Main memory database recovery. In *ACM '86: Proceedings of 1986 ACM Fall joint computer conference*, pages 1226–1232, Los Alamitos, CA, USA, 1986. IEEE Computer Society Press.

[Eic87] Margaret H. Eich. Mars: The design of a main memory database machine. In *Proceedings of the International Workshop on Database Machine, Oct 1987*, pages 468–481, 1987.

[Ew97] Er Vi Ew. Datablitz architectural overview, 1997.

[Fom03] M. Fomitchev. Lock-free linked lists and skip lists. Master's thesis, York University, 2003.

[For05] The Japan OSS Promotion Forum. Evaluation of the database management system cluster. website, 2005. Chapter 7 Results of Evaluating MySQL Cluster, http://www.ipa.go.jp/software/open/forum/north_asia/download/3-051dbmsc-7_e.pdf.

[Fou07] Apache Software Foundation. Derby-646 in-memory backend storage support. Apache Issue tracker, 2007. http://issues.apache.org/jira/browse/DERBY-646.

[FR04] Mikhail Fomitchev and Eric Ruppert. Lock-free linked lists and skip lists. In *PODC '04: Proceedings of the twenty-third annual ACM symposium on Principles of distributed computing*, pages 50–59, New York, NY, USA, 2004. ACM Press.

[Fra04] K. A. Fraser. *Practical lock-freedom.* PhD thesis, University of Cambridge, 2004.

[GD96] Le Gruenwald and Margaret Dunham. Recovery in main memory databases, 1996.

[GL92] V. Gottemukkala and T. J. Lehman. Locking and latching in a memory-resident database system. In *Proceedings of the 18th Conference on Very Large Databases, Morgan Kaufman pubs. (Los Altos CA), Vancouver*, 1992.

[GLPT94] J. N. Gray, R. A. Lorie, G. R. Putzolu, and I. L. Traiger. Granularity of locks and degrees of consistency in a shared data base. pages 181–208, 1994.

[GMP90] H. Garcia-Molina and C.A. Polyzois. Issues in disaster recovery. In *Thirty-Fifth IEEE Computer Society International Conference.*, pages 573–577, 1990.

[GMS92] H. Garcia-Molina and K. Salem. Main memory database systems: An overview. *IEEE Transactions on Knowledge and Data Engineering*, 4(6):509–516, 1992.

[GR93] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.

[Hag86] R B Hagmann. A crash recovery scheme for a memory-resident database system. *IEEE Trans. Comput.*, 35(9):839–843, 1986.

[Har01] T. L. Harris. A pragmatic implementation of non-blocking linked lists. In *Proceedings of the 15th International Symposium of Distributed Computing*, 2001.

[HH01] Michael Hohmuth and Hermann Härtig. Pragmatic nonblocking synchronization for real-time systems. In *Proceedings of the USENIX Annual Technical Conference*, pages 217–230, 2001.

[Hor84] Ellis Horowitz. Fundamentals of data structures in pascal. In *Computer Science Press, Rockville, MD,*, 1984.

[HR83] Theo Haerder and Andreas Reuter. Principles of transaction-oriented database recovery. *ACM Comput. Surv.*, 15(4):287–317, 1983.

[HTBH95] Svein-Olaf Hvasshovd, Øystein Torbjørnsen, Svein Erik Bratsberg, and Per Holager. The clustra telecom database: High availability, high throughput, and real-time response. In Umeshwar Dayal, Peter M. D. Gray, and Shojiro Nishio, editors, *VLDB'95, Proceedings of 21th International Conference on Very Large Data Bases, September 11-15, 1995, Zurich, Switzerland*, pages 469–477. Morgan Kaufmann, 1995.

[Hva92a] Svein-Olaf Hvasshovd. *HypRa/TR: A Tuple Oriented Recovery Method for a Continuously Available Distributed DBMS on a Shared Nothing Multi-Computer*. PhD thesis, The Norwegian Institute of Technology, Trondheim, 1992.

[Hva92b] Svein-Olaf Hvasshovd. *HypRa/TR: A Tuple Oriented Recovery Method for a Continuously Available Distributed DBMS on a Shared Nothing Multi-Computer*. PhD thesis, University of Trondheim, 1992.

[Inc06] Sun Microsystems Inc. The netbeans profiler project. Application, 2006.

[Int04] Intel. *Ia-32 intel architecture software developers manual, volume 3: System programming guide*, 2004.

[JLR+94] H. V. Jagadish, Daniel Lieuwen, Rajeev Rastogi, Avi Silberschatz, and S. Sudarshan. Dalí: A High Performance Main Memory Storage Manager. In *Proceedings of the Twentieth International Conference on Very Large Databases*, pages 48–59, Santiago, Chile, 1994.

[Joh05] Bernt Marius Johansen. Apache derby javazone presentasjon, 2005. `http://www3.java.no/JavaZone/2005/presentasjoner/BerntJohnsen/Bernt_Johnsen-DerbyJavaZone.pdf`.

[JSS93] H. Jagadish, A. Silberschatz, and S. Sudarshan. Recovering from main-memory lapses. In *Proceedings of the 19th Conference on Very Large Databases, Morgan Kaufman pubs. (Los Altos CA), Dublin*, 1993.

[KL80] H. T. Kung and Philip L. Lehman. Concurrent manipulation of binary search trees. *ACM Trans. Database Syst.*, 5(3):354–382, 1980.

[KWL99] Tei-Wei Kuo, Chih-Hung Wei, and Kam-Yiu Lam. Real-time data access control on b-tree index structures. In *ICDE '99: Proceedings of the 15th International Conference on Data Engineering*, page 458, Washington, DC, USA, 1999. IEEE Computer Society.

[LC86] Tobin J. Lehman and Michael J. Carey. A study of index structures for main memory database management systems. In *VLDB '86: Proceedings of the 12th International Conference on Very Large Data Bases*, pages 294–303, San Francisco, CA, USA, 1986. Morgan Kaufmann Publishers Inc.

[LC87] Tobin J. Lehman and Michael J. Carey. A recovery algorithm for a high-performance memory-resident database system. In *SIGMOD '87: Proceedings of the 1987 ACM SIGMOD international conference on Management of data*, pages 104–117, New York, NY, USA, 1987. ACM Press.

[LD96] Jun-Lin Lin and Margaret H. Dunham. Segmented fuzzy checkpointing for main memory databases. In *Selected Areas in Cryptography*, pages 158–165, 1996.

[LE92] X. Li and M. H. Eich. A new logging protocol: Law. Technical report, Southern Methodist University, Dept. of Computer Science and Engineering, Dallas, 1992.

[Lin79] et.al. Lindsay, B. Notes on distributed databases. Technical report, IBM Research Report RJ2571, San Jose, California, 1979.

[LN88] K. Li and J. F. Naughton. Multiprocessor main memory transaction processing. In *DPDS '88: Proceedings of the first international symposium on Databases in parallel and distributed systems*, pages 177–187, Los Alamitos, CA, USA, 1988. IEEE Computer Society Press.

[LPR93] Lawrence L. Larmore, Teresa Przytycka, and Wojciech Rytter. Parallel construction of optimal alphabetic trees. In *Journal of the Association for Computing Machinery*, pages 214–223, 1993.

[LS92] E. Levy and A. Silberschatz. Incremental recovery in main memory database systems. *IEEE Transactions on Knowledge and Data Engineering*, 4(6):529–540, 1992.

[LSC92] T. J. Lehman, E. J. Shekita, and L.-F. Cabrera. An evaluation of starburst's memory resident storage component. *IEEE Transactions on Knowledge and Data Engineering*, 4(6):555–566, 1992.

[Lyo90] Jim Lyon. Tandem's remote data facility. In *In Proceedings of IEEE Compcon, San Francisco, CA*, pages 562–567, 1990.

[MHL+92] C. Mohan, Don Haderle, Bruce Lindsay, Hamid Pirahesh, and Peter Schwarz. Aries: a transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Trans. Database Syst.*, 17(1):94–162, March 1992.

[MS01] M. Moir and N. Shavit. *Concurrent Data Structures*. Sun Microsystems, 2001. Book Chapter, available at `www.cs.tau.ac.il/~afek/ConcurrentDS-MS04.pdf`.

[Nie04] Arne Eirik Nielsen. Effekten av samtidighetskontroll i aksessmetoder for main memory databaser. Master's thesis, NTNU, 2004.

[OE06] Øyvind Reinsberg Olav Engelsåstrø. Håndtering av page cache i derby. Master's thesis, NTNU, 2006.

[Ora06] Oracle. Oracle timesten in-memory database architectural overview. Technical report, Release 6.0 `http://download-east.oracle.com/otn_hosted_doc/timesten/603/TimesTen-Documentation/arch.pdf`, 2006.

[Pug90a] William Pugh. Concurrent maintenance of skip lists. Technical Report CS-TR-2222, 1990.

[Pug90b] William Pugh. Skip lists: a probabilistic alternative to balanced trees. *Commun. ACM*, 33(6):668–676, 1990.

[SAB+05] Michael Stonebraker, Daniel J. Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Samuel Madden, Elizabeth J. O'Neil, Patrick E. O'Neil, Alex Rasin, Nga Tran, and Stanley B. Zdonik. C-store: A column-oriented dbms. In *VLDB*, pages 553–564, 2005.

[SGM87] K. Salem and H. Garcia-Molina. Crash recovery for memory-resident databases. Technical report, Princeton University, Department of Computer Science, 1987.

[SGM89] Kenneth Salem and Hector Garcia-Molina. Checkpointing memory-resident databases. In *Proceedings of the Fifth International Conference on Data Engineering*, pages 452–462, Washington, DC, USA, 1989. IEEE Computer Society.

[SM07] Inc Sun Microsystems. Java platform, standard edition 6 api specification. website, 2007. http://java.sun.com/javase/6/docs/api/.

[Sol06] Knut Magne Solem. Apache derby som mmdb. Technical report, The Norwegian University of Science and Technology (NTNU), 2006.

[Sou06] Gokul Soundararajan. Implementing a better cache replacement algorithm in apache derby. Progress Report http://www.eecg.toronto.edu/~gokul/derby/derby-report-aug-19-2006.pdf, 2006.

[Sun04] Håkan Sundell. *Efficient and Practical Non-Blocking Data Structures.* PhD thesis, Chalmers University of Technology, 2004.

[Val95] J. D. Valois. *Lock-Free Data Structures.* PhD thesis, Rensselaer Polytechnic Institute, Troy, New York, 1995.

[XL95] M. H. Eich X. Li. Checkpointing and recovery in partitioned main memory databases. In *Proceedings of the Intelligent Information Management Systems Conference*, 1995.