Forord

Denne prosjektrapporten er resultatet av den avsluttende diplomoppgaven for den 5-årige sivilingeniørutdanningen i datateknikk ved Norges Teknisk-Naturvitenskapelige Universitet (NTNU) våren 2005. Rapporten besvarer oppgaveteksten:

> *Resultatene fra forskningen på aksessmetoder for main memory databaser med samtidighetskontroll gir tildels motstridende resultater om hvilken aksessmetode som gir høyest ytelse. Nyere resultater har vist at synkroniseringsmekanismene knyttet til samtidighetskontroll kan ha lavere tidskostnad enn tidligere hevdet. Kandidaten skal undersøke ytelsen til ulike samtidighetsalgoritmer for aksessmetoder i main memory databaser under varierte realistiske omgivelser. Det er ønskelig at kandidaten også undersøker algoritmenes skalerbarhet med tanke på antall prosessorer. Da det er usikkerhet knyttet til tidskostnaden ved bruk av synkroniseringsmekanismer, skal kandidaten også undersøke hvilken innvirkning varierende tidskostnad har på algoritmenes ytelse. Basert på resultatene er det ønskelig at kandidaten også presenterer forslag til nye samtidighetsalgoritmer samt, om mulig, utfører ytelsesmålinger av disse.*

Vedlagt rapporten er CD med kildekode.

Jeg vil takke veilederne mine, professor Svein-Olaf Hvasshovd ved NTNU og Senior Research Scientist Øystein Torbjørnsen ved Fast Search & Transfer ASA, for god veiledning, motiverende engasjement og verdifulle diskusjoner som har bidratt til faglig utvikling gjennom hele prosjektet.

Trondheim, 16. juni 2005

Arne Eirik Nielsen

Sammendrag

Den publiserte forskningen om samtidige aksessmetoder for main memory databaser gir ingen entydige svar på hvilke aksessmetoder som yter best. Hovedårsaken til dette er usikkerhet knyttet til tidsforbruket til de nødvendige synkroniseringsmekanismene for samtidighetskontroll. Nyere ytelsesmålinger viser at tidsforbruket knyttet til synkroniseringsmekanismene kan være lavere enn først antatt.

Vi har i denne rapporten simulert flere samtidighetsalgoritmer for aksessmetodene B-trær og T-trær. Simuleringene er foretatt under ulike realistiske omgivelser med varierende prosessorantall. Resultatene viser at algoritmene for T-trær, som setter èn lås i treet, yter bedre enn eller like godt som algoritmene for B-trær under alle omgivelser. Det eneste unntaket er når det anvendes mange prosessorer og en stor andel av operasjonene gjør innsettinger. I slike omgivelser yter B-trær best. Det er også utført simuleringer hvor tidsforbruket knyttet til synkroniseringsprimitivene er variert. Resultatene viser at ved lavt tidsforbruk er det effektivitet med tanke på prosessorbruk som har har mest å si for ytelsen, mens ved høyt tidsforbruk er det antall låser som avgjør ytelsen.

Ved tilstrekkelig mange samtidige operasjoner i aksessmetodene oppstår det, i alle de simulerte algoritmene, en flaskehals som skyldes en delt ressurs ved inngangen til aksessmetoden. Denne flaskehalsen fører til at det ikke oppnås høyere gjennomstrømning av operasjoner ved bruk av flere prosessorer. Det presenteres løsninger på dette problemet som er anvendbare på alle algoritmene. Simuleringer av algoritmer som anvender løsningene viser at man kan oppnå tilnærmet lineær skalering av ytelsen opp til minst 32 prosessorer for alle algoritmene.

Innhold

1 Innledning **1**

2 Teori **5**

2.1 Samtidighet i aksessmetoder 5

2.2 T-tre 7

2.2.1 T-tre modifisert for samtidighet 10

2.2.2 Optimistisk samtidighetskontroll 11

2.2.3 Delvis låsing 14

2.2.4 Pessimistisk samtidighetskontroll 14

2.3 B-tre 17

2.3.1 B-tre modifisert for samtidighetskontroll 17

2.3.2 ARIES/IM 19

2.3.3 B-trær med forbikjøring 21

2.4 Modifisert T-tre 22

2.4.1 Algoritmer 23

3 Relatert forskning **27**

3.1 Aksessmetoder med samtidighet 27
3.2 Aksessmetoder uten samtidighet 28
3.3 Hurtigminneoppmerksomme aksessmetoder 33

4 Simulering 37

4.1 Diskret hendelsessimulering 37
4.2 Diskret hendelsessimulering med J-Sim 39
4.3 Simuleringsmodell 39
4.3.1 Forenklinger 42
4.3.2 Algoritmene 43
4.3.3 Parametre i simuleringsmodellen 45
4.3.4 Valg av multiprogrammeringsgrense 46

5 Resultater 49

5.1 Varierende prosessorerantall 49
5.1.1 Testoppsett 50
5.1.2 Omgivelse 1: OLAP 50
5.1.3 Omgivelse 2: Oppdatering av datavarehus 53
5.1.4 Omgivelse 3: Miks 1 55
5.1.5 Omgivelse 4: Miks 2 56
5.2 Varierende mutexkostnad 59
5.2.1 Testoppsett 59
5.2.2 Omgivelse 1: 4 prosessorer 59

5.2.3 Omgivelse 1: 16 prosessorer 61

5.3 Diskusjon 62

6 Algoritmer uten flaskehals **65**

6.1 B-trær: Flere låser 65

6.2 B-trær: Statisk rotnode 66

6.3 Optimistisk: Flere låser 67

6.4 Simuleringsresultater 68

6.5 Diskusjon 70

7 Konklusjon **71**

8 Videre forskning **73**

Bibliografi **77**

Kapittel 1

Innledning

Etterhvert som DRAM-brikker får bedre lagringskapasitet og prisen synker blir det mer og mer aktuelt å erstatte tradisjonelle diskbaserte databaser (DRDB[1]) med databaser som er lagret i sin helhet i primærminnet (MMDB[2]). I Asilomar-rapporten fra 1998 [1] ble det spådd at innen ti år ville det være vanlig at databaser har en terrabyte med RAM tilgjengelig, og at alle utenom de aller største databasene kjørte i primærminnet. Selv om det er et stykke igjen før dette slår til, er det ingen tvil om bruken av MMDB har blitt utstrakt og med fordel kan anvendes i en rekke domener.

Med hele databasen lagret i primærminnet kan man oppnå responstider som er flere størrelsesordener lavere enn det som har vært vanlig for DRDB. Denne høye forbedringen av ytelsen oppnår man ikke nødvendigvis ved å plassere en diskbasert database i primærminnet uten å gjøre optimaliseringer for databasens nye omgivelsener. Den mest åpenbare forandringen av omgivelsene er at man ikke lenger har disk-I/O som dominerende kostnadsfaktor[3]. Da disk-I/O var den helt klart dominerende kostnadsfaktoren for DRDB var det viktigste aspektet i designet å minimalisere behovet for I/O. For MMDB er det derfor nødvendig å revurdere store deler av designet av databasen. Garcia-Molina og Salem gir i [2] en god innføring til MMDB og diskuterer en rekke viktige designspørsmål.

En viktig del av optimaliseringen av databasesystemer er bruk av aksessmetoder, eller indekser. For å unngå å bruke sekvensielle søk for uordnede data, eller binærsøk for ordnede data for å finne riktig datatuppel i en tabell, anvender man en hjelpestruktur som gir mer effektiv aksess til datatuplene. Det er vanlig å skille mellom to typer aksessmetoder, indekssekvensielle aksessmetoder og aksessmetoder som ikke er indekssekvensielle.

[1]DRDB: Disk Resident Database

[2]MMDB: Main Memory Database

[3]Ordet *kostnad* refererer til tidsforbruk i hele rapporten.

Indekssekvensielle aksessmetoder støtter områdesøk. Det vil si at man gir aksess til datatupler i sortert rekkefølge. Aksessmetoder som ikke er indekssekvensielle gir kun direkte aksess til enkelttupler basert på nøkkelverdi. For MMDB har B-trær og T-trær tradisjonelt vært de mest utbredte indekssekvensielle aksessmetodene og hashbaserte metoder vært de mest utbredte for aksessmetodene som ikke er indekssekvensielle. De hashbaserte metodene har vist seg å gi bedre ytelse enn B-trær og T-trær for operasjoner som ikke krever indekssekvensiell aksess [3]. Denne rapporten fokuserer kun på indekssekvensielle aksessmetoder.

I de tidlige artiklene om aksessmetoder for MMDB er det lite fokus på samtidighetskontroll i indeksene. Faktisk nevner man oftest ikke samtidighetskontroll i det hele tatt. En stor andel av databasene som er i faktisk bruk i dag har aksess fra flere transaksjoner samtidig. Følgelig bør det være naturlig å ha et større fokus på samtidighetskontroll når man vurderer aksessmetoder.

T-trær ble, etter ytelsesmålinger gjort i [3] av Lehman og Carey, akseptert som den beste indekssekvensielle aksessmetoden for MMDB og har blitt tatt i bruk i flere kommersielle systemer [2, 4, 5, 6]. I disse ytelsesmålingene ble det ikke tatt hensyn til samtidighetskontroll. Lu, Ng og Tian presenterer i [7] resultater som viser at dersom man tar hensyn til samtidighetskontroll yter B-trær likevel bedre enn T-trær, selv når algoritmen for T-trær modifiseres for å sette så få låser som mulig i treet. Resultatene ble funnet ved hjelp av simulering. Flere viktige parametre for simuleringen er mangelfullt dokumentert i artikkelen, og tidsforbruket for hver operasjon i simuleringen ble funnet ved hjelp av en udokumentert microbenchmarking. Algoritmene simuleres på en datamaskin med èn prosessor. Dette fører til at man ikke får avdekket hvor effektive algoritmene er med tanke på ressurskonflikter og venting på opptatte ressurser, spesielt slik resultatene blir presentert i artikkelen. Artikkelen blir grundigere gjennomgått i delkapittel 3.2.

I [8] ble det ved hjelp av en ny microbenchmarking vist at kostnadene knyttet til synkroniseringsmekanismene for aksessmetoder i MMDB ikke nødvendigvis er like høye som hevdet i [7]. I tillegg ble det vist analytisk for søkeoperasjoner med de nye kostnadsmålene at T-trær yter bedre enn B-trær med samtidighetskontroll. Dette arbeidet kan ikke konkludere med at T-trær er en bedre aksessmetode enn B-trær for MMDB da det kun ble beregnet kostnader for søketransaksjoner kjørt i isolasjon. Likevel motiverer de lavere låsekostnadene og resultatene for søketransaksjoner kjørt i isolasjon til grundigere undersøkelser om hvorvidt B-tre eller T-tre yter best med samtidighetskontroll. For å kunne se de virkelige innvirkningene av de nye låsekostnadene er det ikke nok med analytiske beregninger for transaksjoner kjørt i isolasjon; det er nødvendig å teste ut hvordan de vil virke inn på virkelige indekser hvor flere transaksjoner utfører operasjoner samtidig. I denne rapporten presenteres derfor resultater fra simulering av de ulike aksessmetodene under omgivelser med samtidige transaksjoner som bygger på de nye microbenchmarkene.

Resultatene til Lu, Ng og Tian i [7] bygger, som sagt, på en noe mangelfullt dokumentert simulering av T-trær og B-trær. På grunn av den manglende dokumentasjonen er det vanskelig å etterprøve resultatene ved å reprodusere simuleringene. Simuleringsresultatene som presenteres i denne rapporten er derfor ikke ment å være noen *direkte* etterprøving av resultatene i [7]. Hovedfokuset er heller på å finne ut hvilken aksessmetode som yter best av T-tre og B-tre med samtidighetskontroll under forskjellige realistiske omgivelser.

Da et stort antall av dagens databaser kjører på datamaskiner med flere prosessorer, simuleres de ulike algoritmene med varierende prosessorantall. Med dette kan man oppnå å observere hvor godt hver algoritme skalerer med tanke på antall prosessorer. Dette er en aktuell problemstilling da effektiv bruk av låser og mutexer kan føre til at man kan utnytte flere prosessorer og dermed gi god responstid til flere transaksjoner samtidig. Ved bruk av indekser som er lagret i sin helhet i primærminnet trenger hver operasjon trolig ganske få prosessorsykler, det vil si at en operasjon ikke blir avbrutt av prosessor mange ganger. Dette fører til at man med bare èn prosessor trolig ikke får simulert effekten av å ha mange samtidige operasjoner da operasjonene blir så fort ferdige at man ikke "rekker" å få inn nye samtidige operasjoner. Ved bruk av flere prosessorer har man derfor også bedre mulighet til å ha flere samtige operasjoner.

I forskningen om MMDB har man operert med forskjellige kostnader på synkroniseringsmekanismer for akessmetodene. For å ta høyde for denne usikkerheten presenteres også simuleringsresultater hvor kostnaden på synkroniseringsmekanismene varieres. Dette gjøres for å avdekke hvor sensitive algoritmene er for kostnaden, og om den relative ytelsen til algoritmene endres ved varierende kostnad.

Lu, Ng og Tian målte ytelsen til B-trær med forbikjøring[4], T-trær med optimistisk samtidighetskontroll og T-trær med pessimistisk samtidighetskontroll. I [8] ble det introdusert tre nye algoritmer; T-trær med delvis låsing, modifisert T-tre og T-trær med pessismistisk samtidighetskontroll med større låsegranularitet. I dette prosjektet måles ytelsen på alle disse algoritmene utenom den sistnevnte. Årsaken til at man ikke måler ytelsen til denne algoritmen er som beskrevet i [8] at denne algoritmen i praksis er nesten helt lik algoritmen for B-trær.

Det viser seg at samtlige av de simulerte algoritmene har en svakhet som gjør at responstidene stiger kraftig når man har et system med tilstrekkelig mange prosessorer. Dette skyldes at samtlige av algoritmene har en delt ressurs som alle transaksjoner må gjennom. Når det blir mange samtidige transaksjoner blir belastningen på denne ressursen så stor at videre skalering ikke er mulig. Det presenteres løsninger på dette problemet og resultater fra simuleringer av algoritmer som implementerer disse løsningene.

I kapittel 2 gjennomgås nødvendig teori. Man ser på nødvendigheten av samtidighetskon-

[4] *B-trær med forbikjøring* er en norsk oversettelse av *B-trær med overtaking*. Det norske navnet anvendes i hele rapporten.

troll i aksessmetoder og de ulike aksessmetodene med tilhørende samtidighetsalgoritmer. I kapittel 3 presenteres resultater fra tidligere forskning om aksessmetoder. I kapittel 4 gis det bakgrunnsinformasjon om simuleringsmetodikken som anvendes i ytelsesmålingene og simuleringsmodellen spesifiseres. I kapittel 5 presenteres resulatene av ytelsesmålingene. I kapittel 6 presenteres algoritmer som skalerer bedre med tanke på antall prosessorer enn de opprinnelige algoritmene. Simuleringsresultater for disse algoritmene presenteres også. Rapporten avsluttes med konklusjon i kapittel 7 og anbefaling av videre forskning i kapittel 8.

Kapittel 2

Teori

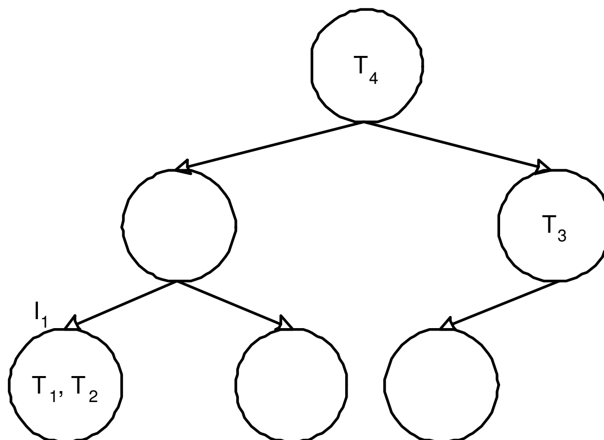
I databaser er det vanlig å anvende en form for aksessmetode for å gi rask aksess til datatupler. For main memory databaser (MMDB) er de mest utbredte indekssekvensielle aksessmetodene T-trær og B-trær. I omgivelser hvor man har flere samtidige transaksjoner er det nødvendig med en form for samtidighetskontroll for å sikre konsistens i aksessmetodene. I delkapittel 2.1 gis en kort introduksjon til samtidighet i aksessmetoder. I delkapittel 2.2 og 2.3 gjennomgås henholdsvis T-trær og B-trær. I delkapittel 2.4 gjennomgås den alternative aksessmetoden, modifisert T-tre, som ble introdusert i [8].

I [3] argumenteres det for å ikke lagre nøkkelverdier eller noen andre data fra datatuplene direkte i indeksene. I stedet lagres kun pekere til datatuplene. Bakgrunnen for dette er at man sparer plass i indeksene ved å ikke lagre noe annet enn datapekere, og at kostnaden knyttet til å følge pekere i primærminnet er meget lav. I [9] anbefaler man å lagre nøkler eller tilsvarende direkte i indeksene for å bedre utnytte hurtigminnet i dagens prosessorarkitektur. Ved å lagre nøkler direkte i indeksene slipper man å følge pekere hver gang man skal finne en nøkkelverdi, og man kan benytte nøkkelverdier som allerede ligger i hurtigminnet. I denne rapporten antas det at man alltid i tillegg til peker til datatupler også har lagret datatuplets nøkkel i indeksen.

2.1 Samtidighet i aksessmetoder

I DRDB er det ønskelig med høy grad av samtidighet i aksessmetodene. Årsaken til dette er at det ved søking i indekser benyttes I/O. Da I/O fører til at en transaksjon ikke bruker prosessoren over en lang tidsperiode, kan andre transaksjoner med fordel bruke samme indeksen mens transaksjonen venter på I/O. Da MMDB ikke bruker i I/O ved søking i indekser er det ikke like åpenbart at det er ønskelig med høy grad av samtidighet i

indeksene. Spesielt dersom databasen kjører på en maskin med èn prosessor kan det være nærliggende å argumentere for ikke å ha samtidighet i indeksen [10]. Årsaken til dette er at en transaksjon kan klare å søke gjennom hele indeksen uten å få tidsavbrudd fra prosessoren. For en multiprosessormaskin kan det være mer naturlig å anta at det kan være nyttig med samtidighet i indeksene. Man kan da slippe at flere transaksjoner som kjører på forskjellige prosessorer må vente på å bruke samme indeks. Figur 2.1 viser en indeks med fire samtidige transaksjoner. Man ser at det er flere transaksjoner tilstede i indeksen, og at flere transaksjoner kan bruke samme node samtidig.

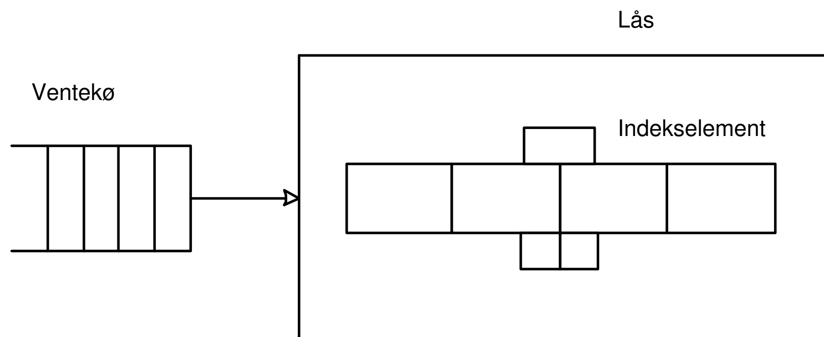


Figur 2.1: Indeks med samtidighet. T_1 - T_4 er transaksjoner som bruker indeksen samtidig.

Transaksjoner som skal gjøre endringer i databasen, for eksempel sette inn eller slette dataelementer, må ofte også gjøre endringer i indeksene knyttet til de aktuelle databasetabellene. I en omgivelse hvor man har samtidige transaksjoner kan ukontrollerte endringer i indekser føre til inkonsistens. Et eksempel på en situasjon som kan føre til inkonsistens er dersom to transaksjoner, T_1 og T_2 som illustrert i figur 2.1, ønsker å bruke samme indekselement, I_1, samtidig. Transaksjon T_1 ønsker å lese indekselement I_1. Før den er ferdig med å lese I_1 blir den avbrutt. Før T_1 får fortsette endrer T_2 på I_1. Når T_1 fortsetter kan indekselementet være forandret på en måte som gjør oppførselen til T_1 uforutsigbar.

For å unngå slike situasjoner er det ønskelig å begrense tilgangen til indekselementer som er under endring. Det betyr at mens en transaksjon endrer på et indekselement skal ingen andre transaksjoner få tilgang til det samme indekselementet. For å oppnå dette er det vanlig å bruke en form for låsemekanisme som ikke er ulik den som brukes for datatupler i en database. Med en slik låsemekanisme kan en transaksjon få eksklusive rettigheter til de indekselementene den skal endre på. Låser som anvendes i indekser omtales i litteraturen både som låser og "latcher". I denne rapporten omtales de kun som *låser*. Figur 2.2 illustrerer et indekselement beskyttet av en lås. Dersom en transaksjon ønsker å låse et indekselement med en låsetype som er inkompatibel med låsetyper som

andre transaksjoner allerede holder, plasseres transaksjonen i en ventekø inntil låsen blir ledig.



Figur 2.2: Lås som beskytter indekselement. Låsen har en kø hvor ventende transaksjoner plasseres dersom noden er låst.

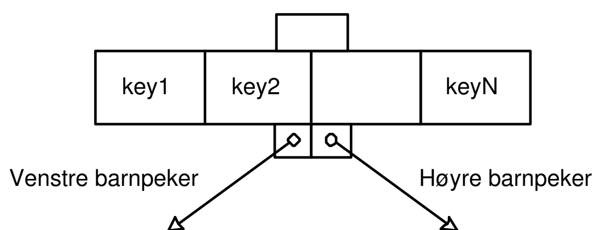
I algoritmene for samtidighetskontroll i T-trær og B-trær som beskrives i delkapittel 2.2, 2.3 og 2.4 brukes tre forskjellige låsetyper; *S-lås*, *SIX-lås* og *X-lås* [11].

2.2 T-tre

Lehman og Carey introduserte aksessmetoden T-tre for å dra nytte av de spesielle karakteristikkene til MMDB [3]. Motivasjon bak T-trær var behovet for en aksessmetode som var effektiv med tanke på CPU-bruk og som tok opp liten plass i minnet. T-trær stammer fra AVL-trær og B-trær.

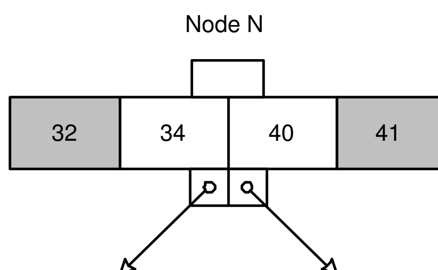
Figur 2.3 viser en node i et T-tre. T-treet er et binært tre, det vil si at hver node kan ha maksimalt to barnpekere. Et T-tre kan ha tre forskjellige nodetyper; noder som har pekere til to barnnoder kalles *interne noder*, noder som bare har peker til en barnnode kalles *halv-løvnoder*, mens noder som ikke har pekere til noen barnnoder kalles *løvnoder*. Hver node har mange nøkler og nøklene ligger i sortert rekkefølge slik at nøkkelen helt til venstre ($key1$) er den minste i noden, mens nøkkelen til høyre ($keyN$) er den største i noden. Hver node kan lagre et maksimalt og et minimalt antall nøkler. Minimums- og maksimumsverdier velges slik at behovet for trerotasjon blir minimert.

Subtreet venstre barnpeker peker til har kun lavere nøkler enn nøkkelen $key1$ i figur 2.3, mens høyre subtre har kun høyere nøkler enn $keyN$. Den minste nøkkelen i en node kalles også $minK$ og den største nøkkelen kalles også $maxK$. Dersom man har en nøkkelverdi K og $minK$ og $maxK$ for node N hvor $minK \leq K \leq maxK$, sier man at node N *avgrenser* K. I figur 2.4 har node N $minK$ og $maxK$ på henholdsvis 32 og 41. Node N



Figur 2.3: Node i T-tre

avgrenser alle nøkler fra og med 32 til og med 41.

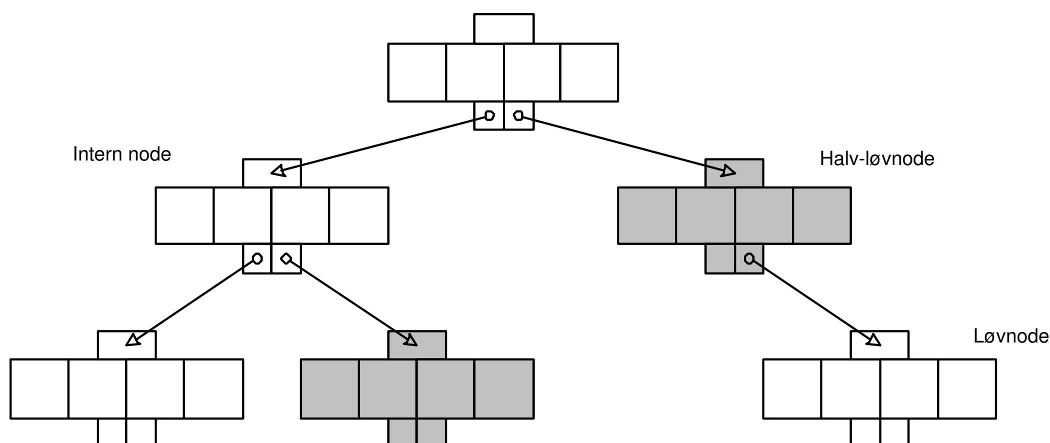


Figur 2.4: Node i T-tre som avgrenser alle nøkler fra og med 32 til og med 41

Figur 2.5 viser et T-tre med tre nivåer. Figuren viser også eksempler på intern node, halv-løvnode og løvnode. T-treet er et balansert tre, det vil si at for hver node i treet er forskjellen i høyden på subtrærne maksimalt 1. Dersom treet blir ubalansert, det vil si at forskjellen i høyde på subtrærne i en node blir høyere enn 1, vil det være nødvendig å balansere treet. Balansering gjøres ved hjelp av rotasjoner som likner på rotasjoner for AVL-trær [12].

For alle interne noder i et T-tre er det èn *foregående node* og èn *etterfølgende node*. Den foregående noden til node N er den noden som har den største nøkkelen som er mindre $minK$ for node N. Den etterfølgende noden til node N er den noden som har den minste nøkkelen som er større enn $maxK$ for node N. Foregående og etterfølgende noder er alltid halv-løv eller løvnoder. De skraverte nodene i T-treet i figur 2.5 viser foregående node og etterfølgende node for rotnoden. Den skraverte noden i det venstre subtreet til rotnoden er den foregående noden, mens den skraverte noden i det høyre subtreet er den etterfølgende noden.

Denne rapporten fokuserer hovedsakelig på samtidighetskontroll i aksessmetoder. Algoritmene for søking, innsetting og sletting for T-tre *uten* samtidighetskontroll gjennomgås derfor kun kortfattet her. For mer detaljerte beskrivelser henvises leseren til Lehman og Careys orginalartikkel om T-trær, se [3]. For søking starter man i rotnoden og søker



Figur 2.5: T-tre. Den skraverte noden til venstre er den foregående noden til rotnoden, mens den skraverte noden til høyre er den etterfølgende noden til rotnoden.

nedover treet. Dersom nøkkelen man søker etter er mindre enn $minK$ søker man videre i venstre subtre. Dersom nøkkelen er større enn $maxK$ søker man videre i høyre subtre, ellers er noden man står i den avgrensende noden. Etter at man har funnet den avgrensende noden finnes nøkkelen ved hjelp av binærsøk.

Ved innsettting brukes vanlig søk for å finne den avgrensende noden. Dersom det er plass til nøkkelen i den avgrensende noden setter man den inn og avslutter. Hvis det ikke er plass til den, fjerner man den minste nøkkelen fra noden og setter inn den opprinnelige nøkkelen. Nøkkelen som ble fjernet fra noden settes inn i den avgrensende nodens foregående node. Dersom den foregående noden er full, oppretter man en ny løvnode under den avgrensende noden og setter inn nøkkelen. Dersom man oppretter en ny løvnode sjekkes treets balanse og eventuelle rotasjoner utføres.

Ved sletting finner man den avgrensende noden ved hjelp av et vanlig søk. Dersom noden ikke inneholder nøkkelen avsluttes algoritmen. Dersom noden inneholder nøkkelen og sletting ikke vil føre til underflyt, slettes nøkkelen og algoritmen avsluttes. Dersom den avgrensende noden er en intern node og slettingen fører til underflyt, henter man den største nøkkelen i den foregående noden og plasserer den i den avgrensende noden. Man sletter så nøkkelen man ønsket å slette. Dersom den avgrensende noden er en halv-løvnode eller løvnode sletter man nøkkelen[1]. Dersom den avgrensende noden er en halv-løvnode og den kan slås sammen med barnnoden, flyttes alle nøklene fra barnnoden opp til den avgrensende noden og barnnoden slettes[2]. Dersom den avgrensende noden er en løvnode og slettingen fører til at den blir tom, slettes noden. Dersom slettingen av nøkkel har ført til at man har slettet noder sjekker man balansen i treet og utfører

[1] Det er tillatt at løvnoder har underflyt
[2] Barnnoden er alltid en løvnode.

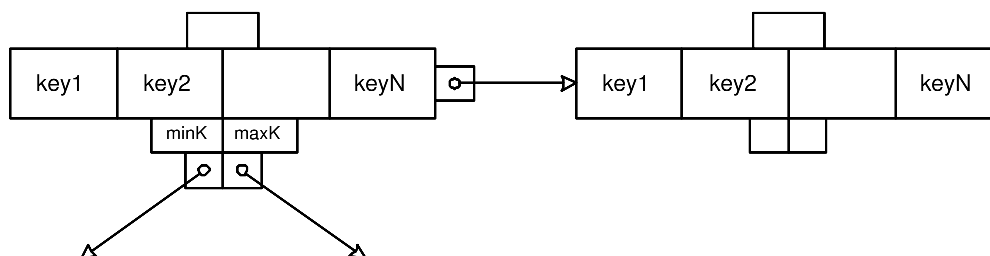
eventuelle rotasjoner.

2.2.1 T-tre modifisert for samtidighet

T-trær har blitt anerkjent som en god aksessmetode for MMDB på grunn av den gode ytelsen presentert i orginalartikkelen [3]. Lu, Ng og Tian påpeker at til tross for at T-trær er tatt i bruk i flere virkelige systemer er det publisert få ytelsesmål for T-trær med samtidighet [7]. De presenterer to algoritmer for samtidighetskontroll i T-trær: pessimistisk og optimistisk samtidighetskontroll. De modifiserer også den orginale strukturen noe for å effektivisere algoritmene for samtidighetskontroll.

Hovedforskjellen fra den orginale datastrukturen er at man for hver node i det samtidige T-treet kan ha èn *halenode*. Halenoder er vanlige T-tre-noder som kun bruker nøklene i nodestrukturen. Dersom man forsøker å sette inn en ny nøkkel i en full node i T-treet, lenker man en halenode til noden. Man kan da lagre nøklene som er avgrenset av den opprinnelige noden i både den vanlige noden og i halenoden. Nøklene i de to nodene ligger fremdeles sortert i stigende rekkefølge. Alle nøklene i halenoden er større enn nøklene i den opprinnelige. Halenoder kan ikke ha lenket flere halenoder til seg. Dersom både den opprinnelige noden og den tilhørende halenoden er fulle, sier man at noden er *helt full*. Halenoder kan på et senere tidspunkt bli satt inn som løvnode under den etterfølgende noden.

Figur 2.6 viser en T-tre-node med halenode. Figuren viser også at den opprinnelige noden har to ekstra felter, nemlig $minK$ og $maxK$. Feltene har samme betydning som beskrevet i kapittel 2.2, forskjellen er at man eksplisitt lagrer verdiene separat i noden. Feltet $minK$ viser altså den laveste nøkkelen i den opprinnelige noden og halenoden, mens $maxK$ viser den høyeste nøkkelen i den opprinnelige noden og halenoden.

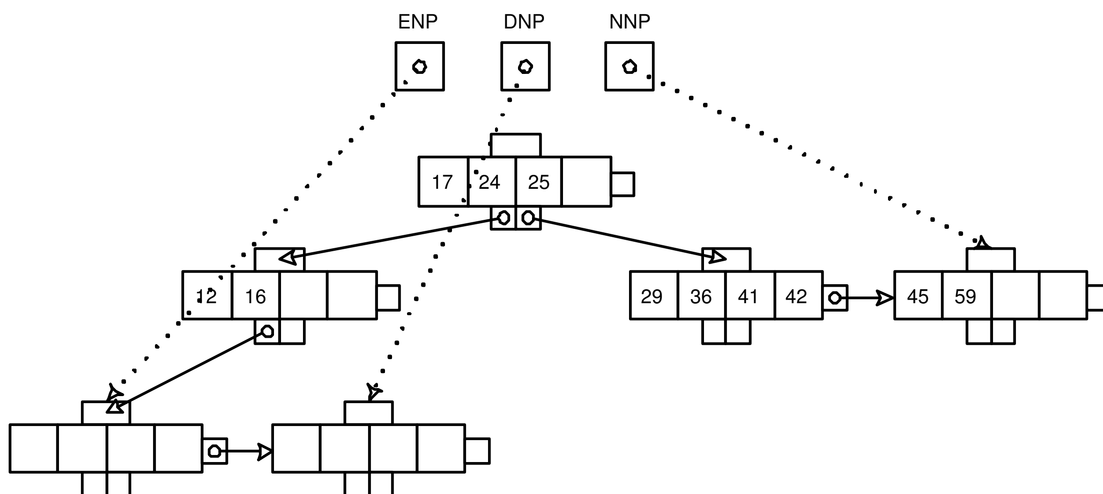


Figur 2.6: Node i T-tre med $minK$ og $maxK$ samt tilhørende halenode

2.2.2 Optimistisk samtidighetskontroll

Lu, Ng og Tian beskriver i [7] optimistisk samtidighetskontroll. Hver transaksjon som bruker algoritmen setter maksimalt èn nodelås. I tillegg må hver transaksjon aksessere to hjelpevariable, *count* og $fixing$, beskyttet av èn mutex. Variabelen *count* er et heltall som viser hvor mange transaksjoner som bruker treet for øyeblikket. Den boolske variabelen $fixing$ viser hvorvidt hele treet er låst og er under omorganisering (fiksing). Transaksjoner som må omorganisere treet setter altså $fixing$ til *true*.

Man bruker tre samlinger med nodepekere; New Node Pool (NNP) som har pekere til alle halenoder opprettet etter forrige omorganisering av treet, Deallocated Node Pool (DNP) som har pekere til halenoder som har blitt tomme etter forrige omorganisering av treet og Empty Node Pool (ENP) som har pekere til tomme vanlige noder. Figur 2.7 illustrerer disse pekersamlinene. Pekeren i ENP peker på en tom vanlig node, pekeren i DNP peker på en tom halenode, mens pekeren i NNP peker på en nyopprettet halenode.



Figur 2.7: T-tre med pekersamlingene NNP, DNP og ENP.

Det ble vist i [8] at optimistisk samtidighetskontroll slik den er beskrevet i [7] ikke nødvendigvis alltid gir forutsigbar oppførsel. Årsaken til dette er at søketransaksjoner ikke setter lås på den avgrensende noden. Dette kan føre til at en transaksjon kan gjøre oppdateringer på en node samtidig som en søketransaksjon bruker den. Søketransaksjonen kan derfor få uforutsigbar oppførsel. I delkapittel 2.2.3 gjennomgås algoritmen *delvis låsing* som ble presentert i [8] for å unngå disse problemene.

Søking

Før man starter søket i selve treet tar man mutexen og sjekker om $fixing$ er $false$. Hvis den ikke er det, slipper transaksjonen mutexen og venter. Ellers inkrementerer man *count* og slipper mutexen.

Man søker så nedover i treet til man finner den avgrensende noden. For hver node man er i sjekker man nøkkelen man søker etter mot $minK$ og $maxK$. Dersom nøkkelen er mindre enn $minK$ søker man i venstre subtre, dersom den er større enn $maxK$ søker man i høyre subtre, ellers er noden man er i den avgrensende noden. Når man har kommet til den avgrensende noden søker man seg fram til den riktige nøkkelen ved hjelp av binærsøk. Til slutt tar man mutexen og dekrementerer *count*.

Oppdatering

Oppdatering foregår på nesten samme måte som søking dersom man antar at oppdatering ikke fører til flytting av data. Man finner den avgrensende noden som for søking, låser noden med X-lås og finner den rette nøkkelen ved hjelp av binærsøk. Til slutt slipper man låsen og dekrementerer *count*.

Innsetting

For å sette inn et datatuppel finner man den avgrensende noden som for vanlig søking. Når man har funnet den avgrensende noden tar man X-lås på den. Dersom noden ikke er helt full, setter man inn nøkkelen på riktig sted i noden eller i en eventuell halenode. Posisjonen til nøkkelen finnes ved hjelp av binærsøk. Dersom innsettingen fører til at det blir opprettet en ny halenode lagres det peker til den i NNP. Til slutt tar man mutexen, dekrementerer *count* og slipper mutexen.

Dersom den avgrensende noden er helt full slipper man nodelåsen. Man tar så mutexen, dekrementerer *count* og dersom $fixing$ er $false$ setter man den til *true*. Man venter så til alle andre transaksjoner har forlatt treet og starter omorganisering av treet. For detaljer rundt omorganisering, se avsnittet *Omorganisering*. Etter omorganiseringen tar man mutexen, setter $fixing$ til $false$ og slipper mutexen.

Sletting

Ved sletting av datatuppel finner man den avgrensende noden ved hjelp av vanlig søking. Man setter X-lås på den avgrensende noden, finner nøkkelen som skal slettes ved hjelp av binærsøk i node og eventuell halenode og sletter nøkkelen. Dersom slettingen fører til at den vanlige noden blir tom lagres det peker til noden i ENP, dersom slettingen fører til at halenoden blir tom lagres det peker til halenoden i DNP. Til slutt tar man mutexen, dekrementerer *count* og slipper mutexen.

Dersom man ikke finner den avgrensende noden i søket tar man mutexen, dekrementerer *count* og slipper mutexen.

Omorganisering

Under omorganiseringen går man gjennom alle halenodene det er pekere til i NNP og forsøker å slå de sammen med den tilhørende vanlige noden. Hvis alle nøklene kan settes inn i den vanlige noden slettes halenoden. Hvis ikke settes halenoden inn som en vanlig løvnode under den etterfølgende noden. Etter flytting av halenode sjekkes balansen på treet og man utfører en eventuell rotasjon som beskrevet i orginalartikkelen om T-trær [3].

Alle tomme noder i ENP erstattes i treet med sine foregående noder. Den foregående noden slettes fra sin opprinnelige posisjon. Man sjekker om treet er balansert og utfører eventuelle rotasjoner. Alle tomme halenoder i DNP slettes.

Nødvendige tillegg til orginalalgoritmen

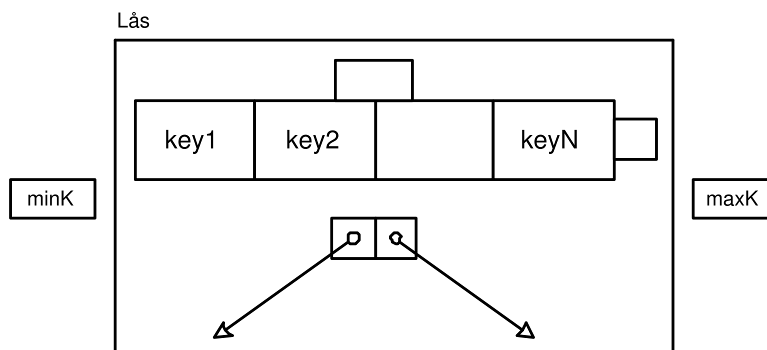
For at implementasjon av algoritmene skal fungere er det nødvendig å spesifisere hvordan man skal håndtere to spesialtilfeller. Lu, Ng og Tian beskriver i [7] ikke hvordan man setter inn selve nøkkelen dersom den avgrensende noden er helt full ved innsetting og man må gjøre omorganiseringer. I denne rapporten antas det at man etter omorganiseringen starter hele innsettingsalgoritmen på nytt, det vil si at man søker seg fram til den avgrende noden og X-låser den før man setter inn nøkkelen.

Orginalartikkelen for optimistisk samtidighetskontroll spesifiserer heller ikke hva som bør skje dersom dersom en transaksjon skal omorganisere et tre som allerede er under omorganisering. En slik situasjon oppstår dersom en transaksjon T_1 må omorganisere treet når en annen transaksjon, T_2, allerede har satt $fixing$ til $true$. I denne rapporten antas det derfor at T_1 restarter innsettingsalgoritmen uten å sette inn peker eller å gjøre

omorganisering. Årsaken til at dette trolig er en god løsning på dette problemet er at omorganiseringen gjort av T_2 vil føre til at den avgrensende noden ikke lenger er helt full. Det er derfor lite sannsynlig at T_1 trenger å gjøre omorganisering igjen etter restarten.

2.2.3 Delvis låsing

Det ble vist i [8] at optimistisk samtidighetskontroll ikke vil gi forutsigbar oppførsel under alle omstendigheter. For å dra nytte av den lovende ytelsen til optimistisk samtidighetskontroll ble det derfor også foreslått endringer av orginalalgoritmen. Hovedforskjellen i den nye algoritmen, *optimistisk samtidighetskontroll med delvis låsing* er at hjelpevariablene $minK$ og $maxK$ ikke er beskyttet av nodelåsene. I tillegg setter *alle* transaksjoner lås på den avgrensende noden. Søketransaksjoner som i den opprinnelige algoritmen ikke setter noen låser må sette S-lås på den avgrensende noden. Transaksjoner som gjør endringer på noder må fremdeles sette X-lås på den avgrensende noden. Med denne låsestrategien kan alle transaksjoner søke seg fram til sin avgrensende node uten å måtte vente på å få låser på vei nedover i treet. Hjelpevariablene $minK$ og $maxK$ i en node vil kun endres når treet er under omorganisering, og da er det ingen transaksjoner som benytter dem til å navigere i treet. Figur 2.8 illustrerer hvordan man låser noder.



Figur 2.8: Node i T-tre som benytter optimistisk samtidighetskontroll med delvis låsing. Transaksjoner kan aksessere $minK$ og $maxK$ for en node uten å låse noden.

2.2.4 Pessimistisk samtidighetskontroll

Lu, Ng og Tian presenterer også en samtidighetsalgoritme som låser noder på hvert nivå i treet ved hjelp av *lock coupling* [13]. Algoritmen tar altså med andre ord flere låser enn optimistisk samtidighetskontroll som kun låser den avgrensende noden ved oppdateringstransaksjoner. Algoritmen tillater rotasjoner i treet samtidig som andre transaksjoner gjør operasjoner i andre deler av treet.

Søking

Man starter søket med å ta S-lås på rotnoden. Man søker nedover i treet og bruker lock coupling med S-låser. Man velger venstre subtre dersom nøkkelen man søker etter er mindre enn $minK$, høyre subtre dersom nøkkelen er større enn $maxK$, ellers er noden man er i den avgrensende noden. Etter at søketransaksjoner har funnet den avgrensende noden bruker de binærsøk til å finne den riktige nøkkelen. Til slutt låser man opp den avgrensende noden.

Oppdatering

Det antas at oppdatering av datatupler aldri fører til flytting av nøkler, altså endres nøkkeldelen aldri. Man finner den avgrensende noden som for søking. Det vil si at man kun bruker S-låser helt ned til den avgrensende noden. Når man har funnet den avgrensende noden låser man den opp og låser den med X-lås før man bruker binærsøk for å finne den riktige nøkkelen. Til slutt slipper man X-låsen og algoritmen terminerer.

Innsetting

Ved innsetting starter man søket ved å sette SIX-lås på rotnoden. Rotnoden settes til å være potensiell *kritisk node*. Kritisk node er den forfedrenoden som er nærmest den avgrensende noden som har forskjell i høyde på barntrærne på 1. Man søker nedover treet og låser nodene med SIX-låser uten å slippe forelderlåsene. Dersom man kommer til en ny node som har forskjell i høyde på barntrærne på 1 setter man denne noden til ny potensiell kritisk node. Man slipper så SIX-låsene på alle noder utenom den nye potensielle kritisk noden og dens forelder.

Dersom man finner den avgrensende noden og den ikke er helt full, oppgraderer man SIX-låsen til en X-lås og man setter inn nøkkelen i noden. Man finner posisjon for nøkkelen ved hjelp av binærsøk. Til slutt slippes alle låsene man holder.

Dersom den avgrensende noden er helt full, søker man etter den etterfølgende noden. Man søker på samme måte som man søkte etter den avgrensende noden. Det vil si at man kun holder SIX-låser fra og med foreldernoden til den etterfølgende nodens kritiske node og ned til den etterfølgende noden. I tillegg holder man låsen på den avgrensende noden. Når man har funnet den etterfølgende noden oppgraderer man låsen på den avgrensende noden til X-lås før man oppgraderer låsen på den etterfølgende noden. For å unngå muligheter for vranglås er det viktig at oppgraderingene skjer i denne rekkefølgen. Halen til den avgrensende noden flyttes ned som løvnode under den etterfølgende noden. Til

slutt setter man inn nøkkelen enten i den avgrensende noden eller i den nyinnsatte løvnoden.

Dersom man satte inn en ny løvnode sjekker man balansen ved å følge stien fra den nye løvnoden opp til den kritiske noden. For hver node på stien sjekker man om forskjellen i høyden på de to subtrærne er større enn 1. Dersom den er det oppgraderer man SIX-låsene fra foreldernoden til den kritiske noden og ned til noden man står i til X-låser før man roterer treet som beskrevet i [3]. Etter at man har utført èn rotasjon er treet balansert. Til slutt slippes alle låser og algoritmen terminerer.

Sletting

Ved sletting finner man den avgrensende noden som for innsetting. Dersom slettingen ikke vil føre til underflyt oppgraderes låsen til X-lås i den avgrensende noden. Man finner den riktige nøkkelen i den avgrensende noden ved hjelp av binærsøk og sletter den fra noden, til slutt slippes alle låsene og algoritmen terminerer.

Dersom den avgrensende noden er en intern node og slettingen fører til underflyt, søker man etter den foregående noden på samme måte som man søkte etter den etterfølgende noden for innsetting. Låsen på den avgrensende noden holdes fremdeles. Alle SIX-låsene fra den avgrensende noden eller fra foreldernoden til den foregående nodens kritiske node oppgraderes til X-låser. Den største nøkkelen i den foregående noden flyttes opp til den avgrensende noden. Dersom den etterfølgende noden er en løvnode og flyttingen av en nøkkel ut av noden gjør at den blir tom, slettes noden. Dersom den foregående noden er en halv-løvnode og blir tom som følge av flyttingen av nøkkel, slettes den og erstattes i treet av sin venstre barnnode. Til slutt balanseres treet som beskrevet i [3] og alle låser slippes.

Dersom den avgrensende noden er en halv-løvnode, og slettingen fører til at den blir tom, erstattes den i treet av sin eneste barnnode. Dersom den avgrensende noden er en løvnode, og innsettingen fører til at den blir tom, slettes den fra treet. Tilslutt sjekkes balansen i treet for begge tilfellene som beskrevet i [3] og alle låser slippes.

Oppgradering av låser

I orginalartikkelen beskrives det ikke i detalj hvordan man oppgraderer SIX-låser til X-låser. For å unngå vranglås er det viktig, dersom man ønsker å oppgradere flere SIX-låser til X-låser samtidig, å alltid oppgradere låsene låsene *ovenfra og ned* i treet. Dette kan illustreres med følgende eksempel. Dersom en transaksjon skal oppgradere to låser, L_1 og L_2, hvor L_2 er lenger nede i treet enn L_1, må transaksjonen oppgradere T_1 før T_2.

2.3 B-tre

B-trær [14] er en aksessmetode spesielt godt egnet for DRDB [3]. Årsaken til dette er at de har få nivåer og har høy forgreiningsfaktor. Den lave høyden fører til at man har behov for å lese få diskblokker ved traversering av treet da hver node ofte er lagret som ei diskblokk i DRDB. For et B-tre med høyde 3 som er lagret i sin helhet på disk, trenger man kun å gjøre 3 diskaksesser for å finne et datatuppel. For MMDB er ikke I/O den dominerende kostnadsfaktoren, følgelig er ikke nødvendigvis B-trær en like god aksessmetode for MMDB. For grunnleggende gjennomgang av datastrukturen henvises leseren til [14].

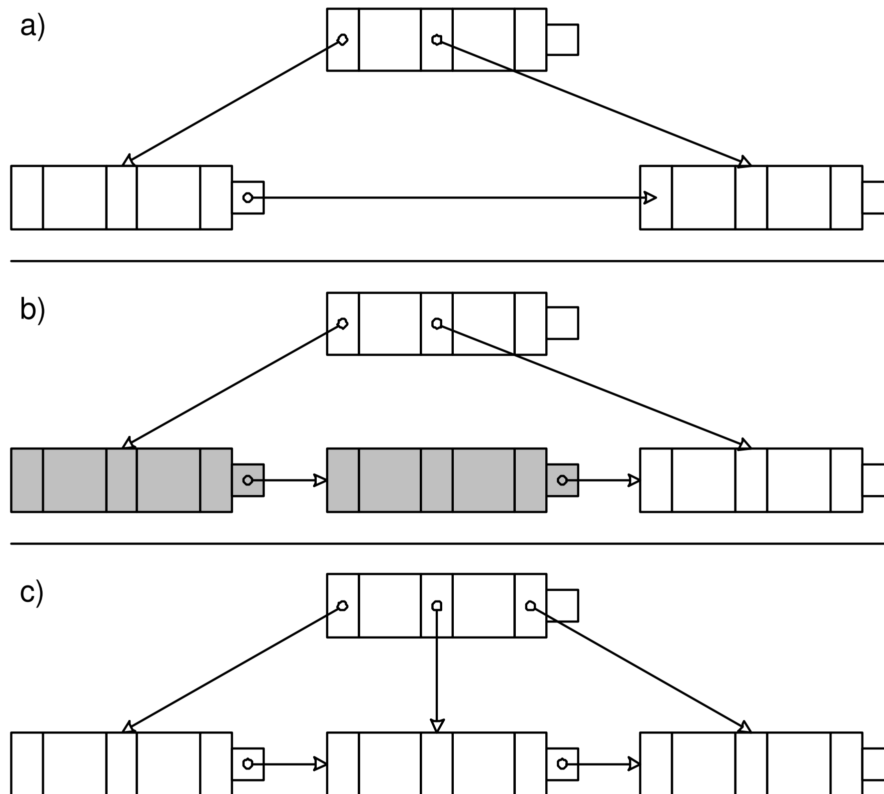
I dette prosjektet anvendes to samtidighetsalgoritmer for B-trær; ARIES/IM og B-trær med forbikjøring. Disse algoritmene gjennomgås i henholdsvis delkapittel 2.3.2 og 2.3.3. ARIES/IM tas med i denne rapporten på grunn av dens støtte for gjenopprettingsmekanismene i ARIES [15]. B-tre med forbikjøring tas med fordi det er algoritmen som ble benyttet i simuleringene til Lu, Ng og Tian i [7]. Den tas også med fordi den kan virke som en mer effektiv metode enn ARIES/IM, da den holder låsene kortere ved traversering av treet. Den største forskjellen mellom de to algoritmene er at ARIES/IM ved traversering bruker lock coupling, mens B-trær med forbikjøring kun holder èn lås om gangen ved traversering. Dette fører til at de setter det samme antall låser, men at ARIES/IM holder låsene lenger enn B-trær med forbikjøring.

2.3.1 B-tre modifisert for samtidighetskontroll

Lehman og Yao modifiserer i [16] B*-trær[3] til B-link-trær for å gjøre omorganiseringer og nodesplitt mer effektivt i omgivelser hvor man bruker samtidighetskontroll. B-link-trær anvendes i algoritmene som gjennomgås i delkapittel 2.3.2 og 2.3.3. Hovedforskjellen fra B*-trær er at man i hver node har en peker til den høyre nabonoden på samme nivå i treet. På hvert nivå i treet danner altså de nye pekerne en lenket liste av noder som starter i noden helt til venstre og slutter i noden helt til høyre som har en null-peker. Den nye strukturen illustreres i figur 2.9a.

Hensikten med de nye pekerne er at man skal ha en ekstra metode for å nå noder i tillegg til å kunne følge pekere nedover i treet og treffe noden direkte. Når man splitter en node på grunn av overflyt, erstattes noden av to nye noder. Pekeren i den nye noden med de minste nøkkelverdiene peker på den nye noden med de største nøkkelverdiene. Linken i noden med de største nøkkelverdiene peker på den høyre nabonoden til den gamle noden. Noden med de minste nøkkelverdiene plasseres på samme fysiske adresse som den gamle noden.

[3] B*-trær har lagret alle datatupler i løvnodene. I de interne nodene er det kun pekere til barnnoder.



Figur 2.9: a) B-link-tre med to nivåer. Rotnoden har to barnnoder. b) Venstre barnnode har fått overflyt som følge av innsetting. Den har blitt splittet og de to nyopprettede nodene er skravert. Det er enda ikke peker fra foreldernoden til den høyre noden. c) Det er opprettet peker fra foreldernoden også til den høyre noden.

Et B-link-tre er gyldig selv om det har enkelte noder som kun kan nås gjennom høyrepekeren til nabonoden, det vil si at nodene ikke har noen foreldernode. For å oppnå bedre responstid for transaksjoner er det mulig å utsette å lage peker i foreldernoden slik at man samler opp flere slike operasjoner og utfører dem samtidig. Dersom man kommer til en node hvor nøkkelverdien man søker etter er høyere enn den høyeste nøkkelverdien i noden må man følge høyrepekeren og sjekke hvorvidt noden er splittet. Figur 2.9b viser to nyopprettede noder som følge av splitt av den venstre barnnoden i figur 2.9a. Figuren viser at det enda ikke er opprettet peker til den ene noden i foreldernoden. I figur 2.9c har peker fra foreldernoden til den siste nyopprettede noden også blitt opprettet.

En annen fordel med B-link-trær er at man effektivt kan benytte dem til sekvensielle søk ved å følge høyrepekerne i løvnodene.

2.3.2 ARIES/IM

C. Mohan introduserer i [17] ARIES/IM. Ved bruk av ARIES/IM holder hver transaksjon maksimalt to nodelåser samtidig på veg nedover i treet. Søking internt i hver node etter barnnode eller datatuppel i løvnoder gjøres ved hjelp av binærsøk. I spesifikasjonen av algoritmen i [17] låser man også selve datatuplene, dette tas det ikke hensyn til i denne rapporten da man kun fokuserer på samtidighetsalgoritmene for selve aksessmetodene.

Algoritmen bruker en trelås som transaksjoner som ønsker å splitte eller slette noder må låse med X-lås. Søketransaksjoner trenger ikke å ta denne låsen. I tillegg har hver node to ekstra bits, *Delete_Bit* og *SM_Bit*. *SM_Bit* settes til 1 i hver node som har blitt påvirket av splitting eller sletting av noder. Hensikten med *SM_Bit* er å advare andre transaksjoner som ønsker å gjøre innsettinger eller slettinger slik at de ikke for eksempel setter inn en nøkkel i feil node.

Søking

Ved søking anvender ARIES/IM *lock cupling* med S-låser på veg nedover i treet, det vil si at man beholder låsen på foreldernoden inntil man har fått låsen på barnnoden. Dersom man har kommet til en løvnode og den ikke har noen datatupler med nøkkel større enn eller lik nøkkelen man søker etter følger man pekeren til den høyre nabonoden og låser den ned S-lås uten å slippe låsen i den første løvnoden man kom til. Dersom den høyre pekeren er en null-peker terminerer algoritmen. Man søker etter nøkkelen i den riktige noden. Til slutt slippes låsene man holder på løvnoder.

Innsetting

Ved innsetting finner man løvnoden man skal sette inn datatuplet i. Man anvender lock coupling med S-låser på veg nedover treet og setter X-lås på løvnoden. Dersom løvnoden ikke er full sjekker man om SM_Bit eller $Delete_Bit$ er 1. Dersom de er det setter man S-lås på trelåsen og setter SM_Bit og $Delete_Bit$ til 0. Trelåsen slippes og dersom nøkkelen man ønsker å sette inn er mindre enn eller lik den største nøkkelen setter man inn nøkkelen, låser opp noden og terminerer.

Dersom nøkkelen man skal sette inn er større enn den største nøkkelen i løvnoden følger man pekeren til den høyre nabonoden og låser den med en X-lås. Dersom det er plass i den nye noden settes den inn i noden, man slipper låsene og man terminerer.

Dersom det ikke er plass til å sette inn nøkkelen i løvnoden må man splitte noden. Man låser trelåsen med X-lås splitter noden som beskrevet i delkapittel 2.3.1 og setter SM_Bit til 1 i noden. Man slipper nodelåser og gjør nødvendige endringer høyere opp i treet og setter SM_Bit i nodene man endrer til 1. Etter at man har gjort nødvendige endringer oppe i treet setter man SM_Bit i de berørte nodene til 0. Man setter så inn nøkkelen i riktig node og slipper nodelåsen og tre-låsen.

Sletting

Ved sletting finner man løvnoden hvor man skal slette nøkkelen fra ved hjelp av lock coupling med S-låser og låser løvnoden med X-lås. Dersom løvnoden har SM_Bit satt til 1, låser man trelåsen med S-lås. Man setter så SM_Bit til 0 i noden og setter $Delete_Bit$ til 1 i noden, man låser så opp foreldernoden. Dersom sletting av nøkkel ikke fører til underflyt og nøkkelen er den største eller den minste i noden låser man treet med S-lås. Man setter så $Delete_Bit$ til 0. Til slutt sletter man uansett nøkkelen og låser opp noden og trelåsen.

Dersom slettingen av nøkkel fører til at en løvnode blir tom og må slettes låser man trelåsen med X-lås og låser opp forelderlåsen. Man sletter så nøkkelen og løvnoden, man justerer høyrepekere slik at de passer den nye strukturen. Man slipper så alle nodelåser. Man propagerer så endringer oppover i treet og setter SM_Bit til 1 i alle berørte noder. Til slutt setter man SM_Bit til 0 i alle noder og låser opp trelåsen.

2.3.3 B-trær med forbikjøring

B-trær med forbikjøring ble introdusert av Yehoshua Sagiv i [18]. Algoritmen er utviklet for bruk i DRDB og gjør noen antakelser som neppe er gyldige i MMDB. Ved bruk av indeksen må transaksjoner gjøre lokale kopier i eget minneområde av nodene de besøker. Det vil si at når en søketransaksjon besøker en node, må den først *lese* noden inn i eget minneområde før den kan søke etter riktig barnpeker. Dersom en transaksjon skal endre på en node, må den først lese inn noden før den gjør endringene i eget minneområde. Til slutt *skrives* noden tilbake til disk. Algoritmen forutsetter at operasjonene *lesing* og *skriving* av noder er atomiske [16]. På grunn av denne antakelsen trenger ikke lesetransaksjoner å sette noen låser i treet, og transaksjoner som fører til at noder forandres trenger bare å sette låser på noder som de faktisk endrer på.

Ved bruk av MMDB vil man ikke kunne få noen garanti for at operasjonene lesing og skriving er atomiske uten å anvende en form for synkroniseringsprimitiv. Dersom man ikke låser andre noder enn de man endrer på, kan man få inkonsistens i indeksen. For eksempel kan transaksjoner når som helst bli avbrutt mens de leser inn en node til eget minneområde, i mellomtiden kan andre transaksjoner ha endret på noden som ble lest. I tillegg kan bruk av maskiner med flere prosessorer føre til at en lesetransaksjon leser en node *samtidig* som en annen transaksjon skriver noden. Med andre ord må det gjøres endringer på algoritmen for at den alltid skal gi konsistens i indeksene. Alle transaksjoner som skal lese en node må derfor låse den med S-lås. I denne rapporten antas det også at transaksjonene gjør endringer direkte på noden og ikke har kopi i eget minneområde.

Søking

Ved søking starter algoritmen i rotnoden. Man låser noden med S-lås og bruker binærsøk for å finne riktig barnnode før man slipper låsen. Slik fortsetter algoritmen til man har kommet til en løvnode. Dersom nøkkelen man søker etter er større enn den største nøkkelen i løvnoden følges høyrepekeren til nabonoden. Til slutt finner man den riktige nøkkelen ved hjelp av binærsøk i den riktige løvnoden.

Innsetting

Ved innsetting søker man seg til den riktige løvnoden nesten som for søking. Den eneste forskjellen er at man lagrer pekere til alle noder man har besøkt på veg nedover i treet. Når man har kommet til løvnoden låses denne med X-lås. Dersom det er plass til nøkkelen i løvnoden setter man den inn og låser opp noden. Viss det ikke er plass nøkkelen i den opprinnelige noden, A, opprettes det en ny løvnode, B. Høyrepekeren i B peker til

samme node som A opprinnelig pekte til, og høyrepekeren i A settes til å peke til B. De opprinnelige nøklene i A fordeles mellom A og B og nøkkelen settes inn i riktig node. Låsen på noden slippes. Dersom innsettingen førte til nodesplitt følger man de lagrede pekerne oppover i treet og setter inn pekere til eventelle nyopprettede barnnoder.

Sletting

Ved sletting finner man den riktige noden som for søking. Man X-låser løvnoden og finner den riktige nøkkelen ved binærsøk. Nøkkelen slettes og låsen på løvnoden slippes og algoritmen avsluttes.

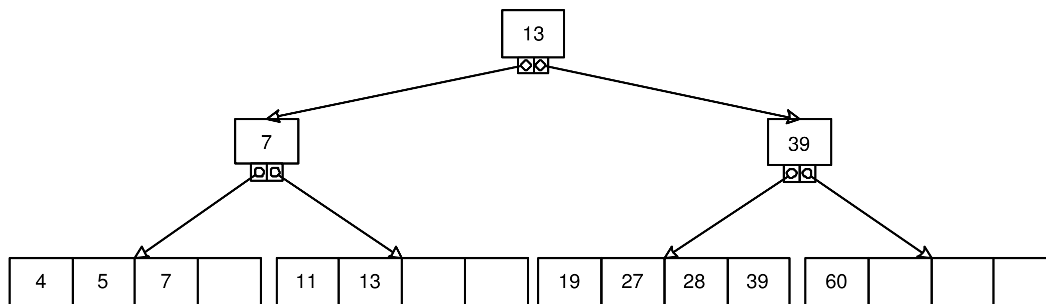
Denne algoritmen for sletting kan føre til at enkelte noder har underflyt. I [18] spesifiseres derfor også en komprimeringsprosess som kan kjøres samtidig som vanlige at vanlige transaksjoner beynytter treet. Hensikten med prosessen er å omorganisere indeksen slik at ingen noder lenger har underflyt. Komprimeringsprosessen spesifiseres ikke nærmere her, og leseren henvises derfor til [18].

2.4 Modifisert T-tre

T-treets struktur gjør at størsteparten av alle transaksjoner har sin avgrensende node på et av de aller nederste nivåene. Det nederste nivået har plass til like mange nøkler som alle de andre nivåene til sammen. For hvert nivå man besøker i et T-tre må man gjøre èn eller to nøkkelsammenlikninger. Da T-trær oftest er veldig høye fører dette til at man må gjøre mange nøkkelsammenlikninger på vei nedover i treet. Motivert av dette ble det i [8] introdusert en ny indekstype; modifisert T-tre.

Modifiserte T-trær har en noe annen struktur enn ordinære T-trær. For det første lagrer ikke de interne nodene vanlige nøkler, de lagrer kun èn nøkkelverdi som brukes i traverseringen av treet. Dette fører til at man for hvert nivå i treet må gjøre èn sammenlikning mindre og at treet blir ett nivå høyere. Kostnaden ved at treet blir ett nivå høyere ved traversering spares inn gjennom at man kun gjør èn sammenlikning for hver node. Figur 2.10 viser et modifisert T-tre. Man ser at internnodene kun inneholder èn nøkkelverdi og pekere til to barnnoder. Løvnoder lagrer nøkler på samme måte som noder i T-trær, det vil si med den minste nøkkelverdien lengst til venstre og størst nøkkelverdi lengst til høyre.

Modifisert T-tre er som vanlige T-trær balanserte, men det har kun løvnoder og interne noder, altså ingen halv-løvnoder. Et modifisert T-tre er balansert dersom man i alle interne noder har forskjell i høyde på subtrærne på maksimalt 1. Dersom innsetting



Figur 2.10: Modifisert T-tre.

fører til at treet blir ubalansert må man utføre rotasjoner. Nøkkelen i de interne nodene brukes ved traversering av treet. Dersom nøkkelen man søker etter har mindre eller lik verdi enn nøkkelen i den interne noden følger man den venstre pekeren, ellers følger man den høyre pekeren.

For å gjøre algoritmen mindre sensitiv for hyppige omorganiseringer hvor hele treet er låst foreslås her en endring fra orginalstrukturen. Hver løvnode kan, som noder i T-trær med samtidighetskontroll, ha en *halenode*. Dersom en løvnode blir full som følge av innsetting, opprettes det en halenode som lenkes til orginalnoden. Nøklene i halenoden er større enn alle nøklene i den orginale noden. Dersom både den orginale noden og den tilhørende halenoden er fulle sier man at noden er *helt full*. Man vedlikeholder også pekersamlingene New Node Pool (NNP), Deallocated Node Pool (DNP) og Empty Node Pool (ENP) som har samme betydning som for T-tre med optimistisk samtidighetskontroll beskrevet i delkapittel 2.2.2.

2.4.1 Algoritmer

Algoritmene for bruk av treet likner på algoritmene for optimistisk samtidighetskontroll med delvis låsing. Det vil si at man har variablene *count* og $fixing$ som er beskyttet av en semafor eller mutex. Hver transaksjon setter kun èn nodelås, og låsing av hele treet som følge av omorganisering skjer relativt sjelden på grunn av halenodene. Alle transaksjonene må ha tilgang til variablene *count* og $fixing$ som er beskyttet av en semafor eller mutex både ved algoritmens start og slutt. Alle søkene i et modifisert T-tre ender i en løvnode.

Søking

Ved søking i modifisert T-tre tar man først semaforen sjekker om $fixing$ er satt til $true$. Dersom den er det slipper transaksjonen semaforen og venter. Hvis ikke inkrementerer man *count* og slipper mutexen.

Man søker så nedover i treet i de interne nodene. Dersom nøkkelen man søker etter er mindre enn eller lik nøkkelen i den interne noden man står i, søker man videre i det venstre subtreet, hvis ikke søker man i det høyre subtreet. Når man når en løvnode låser man den med S-lås og finner den riktige nøkkelen ved hjelp av binærsøk. Til slutt tar man mutexen, dekrementerer *count* og slipper mutexen.

Oppdatering

Oppdatering forgår på nesten helt lik som måte som søking så lenge oppdateringen ikke fører til flytting av data. Man finner den riktige løvnoden som for søking, men låser den med X-lås og finner nøkkelen med binærsøk. Til slutt dekrementeres *count*.

Innsetting

Ved innsetting finner man den riktige løvnoden som for søking, men man låser noden med X-lås. Dersom noden ikke er helt full, bruker man binærsøk og finner riktig sted for den nye nøkkelen og setter den inn. Dersom innsettingen fører til at en ny halenode opprettes lagres peker til den i NNP. Til slutt dekrementerer man *count*.

Dersom noden er helt full slipper man nodelåsen. Man tar så mutexen, dekremeneterer *count* og dersom $fixing$ er $false$ setter man den til *true* og venter til *count* blir 0. Når den blir det er det ingen andre som utfører operasjoner på treet og man starter omorganiseringsalgoritmen. Til slutt restartes innsettingsalgoritmen for å sett inn nøkkelen i riktig node.

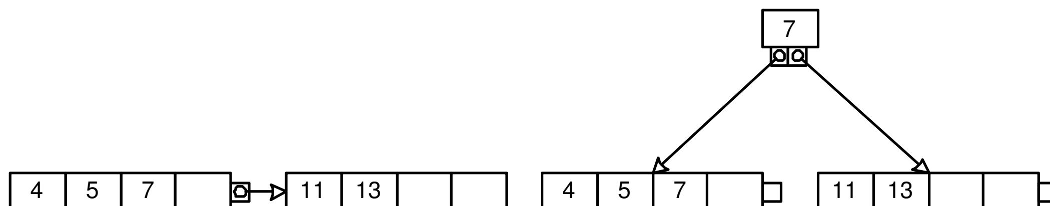
Sletting

Ved sletting finner man den riktige løvnoden som for søking, men man låser noden med X-lås. Man finner nøkkelen som skal slettes ved hjelp av binærsøk i noden og i en eventuell halenode og sletter den. Dersom slettingen fører til at halenoden blir tom, lagres det peker til den i DNP. Dersom slettingen fører til at den vanlige noden blir

tom lagres det peker til den i ENP. Til slutt tar man mutexen, dekrementerer *count* og slipper mutexen.

Omorganisering

Ved omorganisering går man gjennom alle halenodene i NNP og forsøker å slå dem sammen med deres tilhørende opprinnelige noder. Dersom det er mulig, slettes halenoden, hvis ikke erstattes nodene av en intern node i treet med nøkkelverdi lik den største nøkkelverdien i den opprinnelige noden. Den venstre barnnoden til den nye interne noden er den opprinnelige noden, mens den høyre barnnoden er halenoden. Dette illustreres i figur 2.11. Etter å ha gjort dette sjekkes treet for balanse og utfører eventuelle rotasjoner som beskrevet i [3].



Figur 2.11: Til venstre ser man en løvnode med tilhørende halenode. Under omorganisering erstattes de i treet av nodene til høyre.

Alle tomme løvnoder i ENP slettes fra treet og foreldernoden erstattes med nabonoden. Man sjekker så treet for balanse og utfører eventuelle rotasjoner. Alle tomme halenoder i DNP slettes. Til slutt slettes alle pekerne i NNP, DNP og ENP.

Kapittel 3

Relatert forskning

Forskningen som er publisert om ytelsen til ulike aksessmetoder for main memory databaser (MMDB) gir ingen entydige svar på hvilke aksessmetoder som er best. Faktorer som virker inn på resultatene fra de ulike ytelsesmålingene er blant annet hvorvidt man skal ta hensyn til samtidighetskontroll, om man tar hensyn til hurtigminnebom og hvorvidt aksessmetodene er indekssekvensielle.

I dette kapitlet presenteres noen av forskningsresultatene. I delkapittel 3.1 presenteres resultater fra forskningen som ikke tar hensyn til samtidighetskontroll, mens i delkapittel 3.2 presenteres og vurderes resultater fra forskningen som tar hensyn til samtidighetskontroll. Resultatene til Lu, Ng og Tian i [7] gjennomgås spesielt nøye og det vises hvorfor resultatene fra denne artiklene ikke kan etterprøves direkte. I delkapittel 3.3 diskuteres resultater fra forskning som omhandler effekten av ytelsesforskjellen mellom hurtigminne og primærminne på aksessmetoder for MMDB.

Dette prosjektet fokuserer på indekssekvensielle aksessmetoder, det vil si aksessmetoder som støtter områdesøk. Hashbaserte aksessmetoder kan gi meget god ytelse dersom støtte for områdesøk ikke er nødvendig. Eksempler på hashbaserte aksessmetoder for MMDB er *fast search multi-directory hashing* (FSMH) og og *controlled search multi-directory hashing* (CSMH) [19]. I tillegg har extendible hashing blitt utvidet for å støtte samtidighetskontroll i MMDB [20].

3.1 Aksessmetoder med samtidighet

Lehman og Carey introduserte og sammenliknet i [3] T-trær med en rekke andre eksisterende aksessmetoder for bruk i MMDB. De sammenliknet både med hashbaserte

metoder og indekssekvensielle metoder. De hashbaserte metodene kom klart best ut for alle operasjonstyper som ikke var avhengige av områdesøk. Da dette prosjektet tar for seg indekssekvensielle aksessmetoder, det vil si indekser som støtter områdesøk, vil resultatene for de hashbaserte metodene ikke bli videre kommentert.

Ytelsen til B-tre, T-tre og AVL-tre ble sammenliknet. Testene som ble utført på aksessmetodene som er relevante i dette prosjetet er søking, innsetting, sletting og en miks av operasjonene. For søking yter AVL-trær best, noe bedre enn T-trær. Årsaken til dette er at man ved hjelp av AVL-treet søker seg helt fram til nøkkelen ved hjelp av pekere, mens med T-treet anvender et kostbart binærsøk i den avgrensende noden. B-trær er den dårligste metoden på grunn av at den bruker binærsøk i alle noder man besøker.

For innsetting yter T-trær best av metodene. AVL-trær og B-trær yter relativt likt og er ca 50% dårligere enn T-trær. Årsaken til at T-treet yter klart best for innsetting er at metoden har god søkeytelse og innsettingen fører sjelden til rotasjon. Ved overflyt av noder slipper man å splitte noder[1], noe som fører til sjeldnere rotasjoner. AVL-trær har god søkeytelse, men er ineffektive ved innsetting på grunn av at de utfører mange minneallokasjoner samt at de utfører hyppige rotasjoner i treet. B-treet tar igjen den dårlige søkeytelsen ved at den utfører få minneallokasjoner og sjeldne nodesplitter ved innsetting. For sletting yter T-treet best av metodene foran B-tær og AVL-trær.

Ved en miks av de ulike operasjonstypene med 60% søking, 20% innsetting og 20% sletting yter T-trær best på grunn av at de har både gode søke- og oppdateringsegenskaper. AVL-trær yter noe bedre enn B-trær på grunn av den klart bedre søkeytelsen. Lehman og Carey anbefaler å bruke T-trær som aksessmetode for MMDB når man trenger støtte for områdesøk.

3.2 Aksessmetoder uten samtidighet

Gottemukkala og Lehman [10] undersøker hvordan man kan forbedre ytelsen i aksessmetodene for main memory-versjonen av Starburst[2]. Starburst bruker T-trær som aksessmetode. Man fant ut at låsing ved bruk av indekser stod for opp til 40% av utførelsestiden, og som en følge av dette vurderes det hvorvidt det kan være lurt å bare bruke èn lås for indekser; en indekslås som låser hele tabellen. En årsak til at dette skal være attraktivt er at sjansen for at man får tidsavbrudd i prosessoren mens man er i indeksen er liten siden man ikke bruker I/O. Man beregnet analytisk at dersom man bruker indekslåser blir ikke samtidigheten i databasesystemet som helhet vesentlig redusert ved maskiner

[1] Innsettingsalgoritmen for T-trær uten samtidighetskontroll splitter ikke noder, den flytter bare nøkler nedover i treet.

[2] Starburst var en databaseprototyp for forskning på nye databasekonsepter utviklet ved IBM Almaden Research Center. Starburst kunne kjøre som en ren main memory database

med flere prosessorer, samtidig som ytelsen for hver operasjon blir 35% bedre.

Lu, Ng og Tian presenterer i [7] resultater fra simulering av B-tre og T-tre med samtidighetskontroll. De sammenliknet T-tre med optimistisk samtidighetskontroll, T-tre med pessimistisk samtidighetskontroll og B-tre med forbikjøring. Resultatene viser at den relative ytelsen til B-tre og T-tre med samtidighetskontroll ikke er den samme som uten samtidighetskontroll. T-tre med pessimistisk samtidighetskontroll hadde klart dårligst ytelse. Faktisk var den mange ganger så tidkrevende som de to andre algoritmene. B-tre med forbikjøring hadde noe bedre ytelse enn T-tre med optimistisk samtidighetskontroll. Forutsetningene for resultatene i [7] virker noe mangelfullt dokumentert og resultatene presenteres på en måte som ikke avdekker hvorvidt en algoritme tillater høy grad av samtidighet.

Simuleringene som ble utført i [7] bygget på microbenchmarking av operasjonene prosessoren utfører i sammenheng med algoritmene. Hensikten til microbenchmarkingen var å avdekke tidsforbruket til hver operasjon. Resultatene fra microbenchmarkingen viste at bruk av synkroniseringsmekanismer, altså semaforer, var den klart dominerende kostnadsfaktoren. Faktisk var disse operasjonene nesten tre størrelsesordener mer tidkrevende enn de andre operasjonene, for eksempel sammenlikning eller aritmetiske operasjoner. Selve utførelsen av microbenchmarkingen er tilnærmet udokumentert og det blir ikke sagt noe om hvilke typer semaforer som anvendes.

Simuleringsmodellen som blir brukt i [7] har en rekke terminaler som sender forespørsler til systemet. Systemet har èn prosessor som er schedulert etter round robin-prinsippet. Det vil si at hver prosess får bruke prosessoren i et forhåndsbestemt tidsintervall, når tidsintervallet er brukt opp blir prosessen plassert i kø for å få tilgang til prosessoren igjen. Enkelte av parametrene i simuleringsmodellen er tildels mangelfullt dokumentert. Det blir ikke oppgitt hvor mange terminaler som sender forespørsler til systemet, eller hvor lang tenketid hver terminal har før den sender en ny forespørsel etter å ha fått svar på den forrige. Dette er viktige parametre da det er ønskelig å vite hvor mange transaksjoner som er i systemet samtidig. Dersom det er mange transaksjoner samtidig er sannsynligheten for ressurskonflikter større, noe som kan få innvirkning på responstidene til transaksjonene i simuleringen. Det blir heller ikke oppgitt hvor lange de forhåndsbestemte tidsintervallene prosesser får bruke prosessoren er. Dersom tidsintervallet er veldig langt kan operasjonene gjøre seg ferdig i indeksen på det første intervallet. Dette vil føre til at man i praksis ikke har noen samtidighet og at alle operasjonene blir serialisert. Dersom intervallet er kort kan flere operasjoner være tilstede i indeksen samtidig og sjansen for ressurskonflikter er tilstede.

I presentasjonen av resultatene fra ytelsesmålingene til Lu, Ng og Tian i [7] presenteres kun den *totale* tiden det tar å utføre et bestemt antall transaksjoner. Denne presentasjonsformen gir i praksis samme informasjon som systemets throughput, eller *transaksjoner per sekund* (TPS). Dersom man for eksempel ved en simulering kjører en halv

million transaksjoner på 10 sekunder, tilsvarer dette throughput på 50.000 TPS. Denne presentasjonsformen sier ingenting om *responstiden* til transaksjonene. To systemer kan ha identisk throughput, men samtidig ha svært forskjellig responstid. For eksempel kan en aksessmetode som kun tillater èn samtidig transaksjon gi throughput på 50.000 TPS og gjennomsnittlig responstid, R_1, på

$$R_1 = \frac{1}{50.000} = 0,00002s,$$

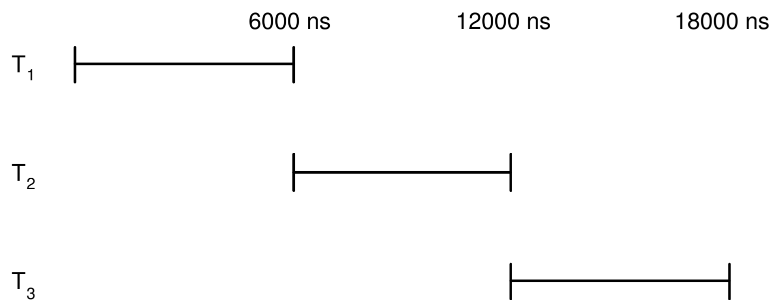
mens en annen aksessmetode som også har throughput på 50.000 TPS, men tillater 100 samtidige transaksjoner kan gi en gjennomsnittlig responstid, R_2 på

$$R_2 = 100 \cdot R_1 = 0,002s.$$

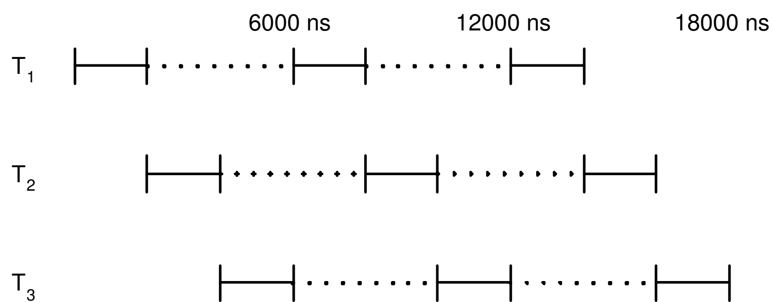
Med andre ord er denne fremstillingsformen noe mangelfull. En aksessmetode kan gi god througput, men samtidig gi transaksjoner dårlig responstid.

I simuleringsmodellen som ble brukt i [7] var det bare èn prosessor i systemet. Når det bare er èn prosessor i systemet er den totale tiden det tar å utføre et bestemt antall transaksjoner, altså throughputen, uavhengig av hvor mange transaksjoner man har samtidig i indeksen. At transaksjoner må vente lenge på låser i treet, eller av andre grunner blir avbrutt til fordel for andre transaksjoner har ingenting å si på throughputen, så lenge prosessoren hele tiden har arbeid å gjøre. Dette illustreres i figur 3.1 og 3.2 hvor tre transaksjoner utfører operasjoner i en indeks med og uten samtidighet. I figur 3.1 har man ikke samtidighet i indeksen, her har alle transaksjonene responstid på 6000 nanosekunder. I figur 3.2 har man samtidighet i indeksen og alle transaksjonene blir avbrutt etter å ha brukt prosessoren i 2000 nanosekunder. Her har indeksen samme throuhput som transaksjonene i figur 3.1, men transaksjonene har en responstid på 14000 nanosekunder. Med andre sier fremstillingen i [7] ikke noe om hvor godt hver enkelt transaksjon yter med tanke på responstid. Man får da heller ikke avdekket hvor stor grad av samtidighet de ulike metodene tillater.

For å måle ytelsen til de tre algoritmene i [7] utføres en halv million transaksjoner. Simuleringene utføres en rekke ganger med forskjellig oppdateringsandel. Det vil si at man varierer andelen innsetting og sletting fra ingen oppdatering (kun søking) til bare oppdatering (ingen søking). Den totale tiden for å utføre alle transaksjoner for hver simulering, altså througput, registreres. Den totale tiden for T-tre med pessimistisk samtidighetskontroll varierer fra 136.6 sekunder for ingen oppdatering til 140.4 sekunder for bare oppdatering. For T-tre med optimistisk samtidighetskontroll varierer den totale tiden fra 10.8 til 19.6 sekunder, mens for B-tre varierer den fra 2.2 til 11.3 sekunder. Til tross for at T-tre med pessimistisk samtidighetskontroll er klart underlegen ser man at den



Figur 3.1: Transaksjonene T_1, T_2 og T_3 i en indeks *uten* samtidighet. Alle transaksjonene har responstid på 6000 nanosekunder.



Figur 3.2: Transaksjonene T_1, T_2 og T_3 i en indeks *med* samtidighet. Alle transaksjonene har responstid på 14000 nanosekunder.

totale tiden varierer relativt lite. Simuleringen tar bare litt under 4 sekunder lenger tid med bare oppdatering enn med bare søking. Man kan forvente at dersom transaksjonenes responstid hadde blitt målt ville denne blitt langt høyere ved bare oppdatering enn ved bare søking. Årsaken til dette er at alle oppdateringsoperasjonene låser rotnoden med SIX-lås helt til man har funnet en kritisk node lenger nede i treet, i mellomtiden må alle andre transaksjoner som vil bruke rotnoden vente. Siden man bare har èn prosessor vil ikke dette gi utslag på througputen. T-tre med optimistisk samtidighetskontroll bruker nesten dobbelt så lang tid med bare oppdatering som den gjør med bare søking. Det argumenteres for at noe av årsaken til dette er at låsingen av hele treet ved overflyt fører til nedsatt samtidighet. Som vist vil ikke nedsatt samtidighet ha noe å si på throughputen med bare èn prosessor. Det blir ikke gitt noen forklaring på hvorfor B-tre med forbikjøring bruker nesten 5 ganger så lang tid med bare oppdatering enn med bare søking. En mulig årsak til dette kan være at man har implementert B-trær med forbikjøring som beskrevet i orginalartikkelen [18], det vil si uten å sette låser på noder man bare leser. Som vist i delkapittel 2.3.3 fører bruk av denne algoritmen ikke alltid til konsistente indekser i MMDB.

Videre utføres simuleringene med varierende totalt antall operasjoner fra 0.1 til 1 millioner operasjoner. Man har 80% søking, 10% innsetting og 10% sletting. Også her måler man den totale tiden som brukes for å utføre alle operasjonene, altså througput. Resultatene fra disse simuleringene viser at det totale tidsforbruket stiger lineært og ved 1 million operasjoner bruker alle algoritmene tilnærmet nøyaktig 10 ganger så lang tid som ved 0.1 million operasjoner. Dette resultatet er som ventet, da det er ganske naturlig at 10 ganger flere operasjoner bruker 10 ganger så lang tid. Så lenge man ikke varierer noen andre parametre vil ikke througputen endres ved at terminalene utfører flere operasjoner totalt. Å variere det toalte antallet operasjoner, kan virke noe unødvendig, da man ikke avdekker noe nytt aspekt ved algoritmene.

I [7] konkluderes det med at B-tre yter bedre enn T-tre med samtidighetskontroll. Det konkluderes også med at på grunn av den høye kostnaden knyttet til bruk av semaforer vil algoritmene som setter flest låser yte dårligst. Det tas altså ikke hensyn til hvor effektivt algoritmene bruker låsene, det vil si om låsestrategien fører til at man får flaskehalser i indeksene som gjør at transaksjoner må vente lenge for å få tilgang til ressurser. En mulig årsak til dette kan være at man i artikkelen kun har hatt fokus på throughput, og ikke responstid. Man har derfor ikke hatt mulighet til å avdekke hvorvidt nedsatt samtidighet fører til høyere responstid.

3.3 Hurtigminneoppmerksomme aksessmetoder

Da Lehman og Carey introduserte og målte ytelsen til T-trær i [3] førte ikke hurtigminnebom[3] til store tidstap. Siden artikkelen ble publisert i 1986 har prosessorytelsen økt med cirka 60% per år, mens minneytelsen har økt med cirka 10% per år. Dette har ført til at den relative kostnaden for hurtigminnebom har økt med to størrelsesordener siden 1986.

Man kan sammenlikne optimaliseringen av indekser for god utnyttelse av hurtigminne med optimaliseringen av indekser for disk. For disk er det vanlig å lagre deler av indeksen i minne for å unngå flere diskaksesser enn nødvendig. For hurtigminne er ikke tidstapet like stort ved bom som for indekser lagret på disk, men det fører til vesentlige ytelsestap. Ved bruk av disk kan man selv kontrollere hvilke diskblokker man skal lagre i minne for hurtig aksess. Dette er ikke mulig for hurtigminne; hurtigminnet administreres av hardware og databasesystemet har derfor ikke direkte kontroll over hvilke minneområder som er lagret i hurtigminnet.

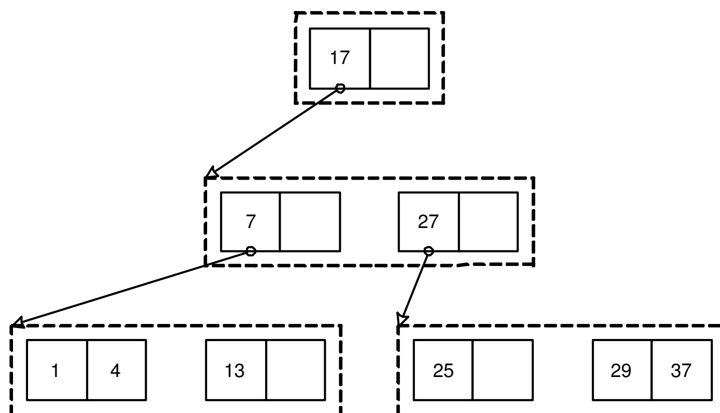
Aksessmetoder som nyttiggjør hurtigminne på en god måte, det vil si minimaliserer antall hurtigminnebom, kalles hurtigminneoppmerksomme. I hurtigminneoppmerksomme aksessmetoder bør man ha så stor tetthet som mulig av data man har bruk for i de innhentede hurtigminnelinjene. Dersom man man bare gjør nytte av en liten del av hurtigminnelinjene vil man måtte hente inn flere hurtigminnelinjer fra primærminnet, og man får følgelig flere kostbare hurtigminnebom. Nodene i T-trær inneholder mange pekere, men bare to av dem er til barnnoder og kan derfor anvendes under traverseringen av treet. De andre pekerne er datapekere som kun brukes i binærsøket i den avgrensende noden. Man bruker derfor bare en liten del av dataene som ligger lagret i hurtigminne. Nodene i B-trær derimot har pekere til mange barnnoder og man kan derfor gjøre nytte av de innhentede dataene i langt større grad enn for T-trær. Ytelsesmålingene til Rao og Ross i [9] bekrefter at B-trær er mer hurtigminneoppmerksom enn T-trær. De har testet trærne i en omgivelse med bare søketransaksjoner uten samtidighetskontroll. Ved store indekser bruker B-treet cirka 30% kortere tid enn T-treet.

Rao og Ross har også introdusert to nye aksessmetoder: Cache Sensitive Search Trees (CSS-trær) [9] og Cache Sensitive B^+-trær (CSB^+-trær) [21]. CSS-trær har karakteristikker som gjør det attraktivt for bruk ved datavarehusomgivelser, det vil si at det støtter hurtig leseaksess og yter godt under periodiske gjenoppbygninger av treet. Ved bruk av CSS-trær ligger alle dataene lagret i sortert rekkefølge i en tabell i minne. CSS-treet

[3]Hurtigminne eller cache er et lite og raskt SRAM-minne som inneholder de senest brukte data i primærminnet. Programmereren har ingen direkte kontroll over innholdet i hurtigminnet. Når man skal bruke data lagret i primærminne sjekker man først om man har lagret kopi av den i hurtigminne. I så fal har man hurtigminnetreff og man bruke denne kopien uten å måtte gå til primærminnet. Dersom man ikke har det må man hente dataene i primærminnet, dette kalles hurtigminnebom og fører til ytelsestap.

brukes for å søke i tabellen på en mer effektiv måte enn ved binærsøk[4]. Alle nodene i et CSS-trær inneholder et forhåndsbestemt antall nøkler, m. Man velger m slik at nodene akkurat passer i en hurtigminnelinje slik at søk i hver node fører til maksimalt èn hurtigminnebom. Man kan kalkulere adressen til barnnodene slik at man slipper å lagre pekere, på denne måten får man plass til flere nøkler i hver node.

Da CSS-trær kun er til bruk under omgivelser hvor databasen er statisk og kun oppdateres batchvis har Rao og Ross utviklet CSB$^+$-trær for å støtte databaser med behov for hyppige oppdateringer. CSB$^+$-treet er avledet fra B$^+$-treet og likner derfor mye på denne strukturen. Hovedforskjellen fra B$^+$-trær er at nodene ikke lagrer pekere til alle barnnodene eksplisitt, de lagrer istedet bare peker til den *første* barnnoden, altså barnnoden lengst til venstre. Alle barnnodene ligger lagret etter hverandre i en tabell og man kan nå noder ved hjelp av en offset i tabellen fra den første noden. Barnnodene til en node ligger alltid lagret etter hverandre, og kalles en *nodegruppe*. Figur 3.3 illustrerer dette. Hensikten med bare å lagre peker til den første barnnoden er at man sparer plass i noden og dermed får plass til flere nøkler. Dette vil føre til færre hurtigminnebommer. Ved innsetting av nøkler kan det bli nødvendig med splitting av noder. Nodesplitt gjøres slik at nodene i nodegrupper alltid er lagret etter hverandre.



Figur 3.3: CSB$^+$-tre. De stiplede boksene illustrerer nodegrupper.

Ytelsesmålinger utført i [9] viser at CSS-trær har bedre ytelse enn B$^+$-trær i omgivelser med bare søketransaksjoner. T-trær har klart dårligst ytelse på grunn av den dårlige utnyttelsen av hurtigminnelinjer. Det viser seg at B$^+$-trær har 50% flere hurtigminnebom enn CSS-trær. I [21] vises det at i omgivelser med oppdatering yter CSB$^+$-træt langt bedre enn B$^+$-trær, og det anbefales at i fremtidige implementasjoner av main memory databaser bør man vurdere å bruke CSB$^+$-trær som aksessmetode.

CSS-trær og CSB$^+$ ble studert uten hensyn til samtidighet. I [22] studeres hurtigminne-

[4] Binærsøk er ikke hurtigminneoppmerksomt.

oppmersomme aksessmetoder som støtter samtidighet i multiprosessoromgivelser. Et viktig aspekt i dette studiet var å minimere behovet for invalidering av hurtigminne i de ulike prosessorene. Man introduserer samtidighetsalgoritmen *Optimistic, Latch-Free Index Traversal* (OLFIT) som er en optimistisk metode som traverserer treet uten å låse noen noder. Oppdateringstransaksjoner låser kun løvnoder, og låser oppover i treet dersom oppdateringen fører til nodesplitt eller sletting av noder. For å gjøre traverseringen av treet uten låser sikker har hver node et versjonsnummer. Transaksjoner som låser noden inkrementerer versjonsnummeret mens den holder nodelåsen. Transaksjoner som leser noden under traversering av treet sjekker versjonsnummeret før og etter lesing av noden, dersom versjonsnummeret har endret seg leses noden på nytt. Dette gjentas inntil versjonsnummeret ikke endrer seg. OLFIT kan anvendes for både B^+-trær og CSB^+-trær. Ytelsesmålinger viser at for omgivelser med bare lesing skalerer algoritmen like bra som algoritmer hvor man ikke har samtidighetskontroll, og for omgivelser med oppdateringer skalerer algoritmen langt bedre enn andre alternative algoritmer for B^+-trær og CSB^+-trær som setter flere låser.

Det har nylig også større fokus på hurtigminne som kan kontrolleres av programmereren [23]. Dette har til nå hovedsakelig vært forbeholdt innebygde systemer. Med den nye CELL-arkitekturen[5] er det sannsynlig at programmerbart hurtigminne kan bli tilgjengelig også for standard datamaskiner. Med denne typen hurtigminne vil det kunne være enklere å utvikle aksessmetoder som utnytter hurtigminne på en god måte.

I denne rapporten tas det ikke hensyn til effekter av hurtigminne. Årsaken til dette er at det ville gjort arbeidet for omfattende i forhold til tid og ressurser tilgjengelig. De samme antakelsene med hensyn på hurtigminne ble forøvrig også implisitt gjort i [7, 8].

[5]CELL-arkitekturen er et samarbeidsprosjekt mellom IBM, Sony og Toshiba. Prosessorarkitekturen anvender en rekke prosessorer på samme brikke som har et eget programmerbart hurtigminne. Hensikten med dette er at man skal kunne bruke applikasjonsspesifikk administrasjon av hurtigminnet for å minimere antall hurtigminnebom. Da dette er en ny arkitektur er det publisert lite detaljert teknisk informasjon om arkitekturen. Sonys forskningsweb inneholder en foreløpig spesifikasjon på: http://www.research.scea.com/ .

Kapittel 4

Simulering

Det ble indikert i [8] at resultatene til Lu, Ng og Tian i [7] ikke nødvendigvis ga riktig bilde av ytelsesforholdet mellom de ulike samtidighetsalgoritmene. Spesifikt ble det ved hjelp av analytisk benchmarking vist at utførelsestiden for søking i T-trær med optimistisk samtidighetskontroll er lavere enn utførelsestiden i B-tre med tilsvarende størrelse. Siden resultatene ble funnet analytisk ble det ikke tatt hensyn til ressurskonflikter og eventuelle omorganiseringer av treet som følge av oppdateringstransaksjoner. For å etterprøve resultatene til Lu, Ng og Tian på en mer realistisk måte simuleres de ulike algoritmene. Ved bruk av simulering kan man lettere finne effekten av at det er flere samtidige transaksjoner i systemet som ønsker tilgang til de samme ressursene. I tillegg er det mulig å simulere virkningen av å ha et system med flere prosessorer.

Simuleringsmetodikken som benyttes, *diskret hendelsessimulering*, beskrives kort i delkapittel 4.1. I gjennomføringen av selve simuleringen brukes simuleringspakken *J-Sim*. J-Sim gjennomgås kort i delkapittel 4.2.

I delkapittel 4.3 gjennomgås simuleringsmodellen som benyttes i simuleringene. For å gi et bedre innblikk i hvordan de ulike simuleringsresultatene fremkommer gis også en nærmere beskrivelse av de implementerte algoritmene og de forenklingene som er gjort. Det gis også en beskrivelse av parametrene som kan varieres i simuleringsmodellen for å simulere ulike omgivelser.

4.1 Diskret hendelsessimulering

En simulering er en imitasjon av en prosess eller et system over tid fra den virkelige verden [24, 25, 26]. En *simuleringsmodell* er en representasjon av prosessen eller systemet

som skal simuleres som tar hensyn endringene som hender i løpet av tiden simuleringen utføres. Ved *diskret hendelsessimulering* endres modellen kun ved diskrete tidspunkt, og ikke kontinuerlig.

Tilstanden til en diskret modell beskrives ved hjelp av en rekke tilstandsvariabler. Tilstandsvariablene endres kun ved bestemte tidspunkt som forårsakes av diskrete hendelser. En *entitet* er et objekt i modellen. Dynamiske entiteter kan oppstå under simuleringen og representerer vanligvis et objekt fra den virkelige verden som beveger seg gjennom systemet. *Ressurser* er entiteter som tilbyr tjenester til dynamiske entiteter. Ressurser har vanligvis en begrenset kapasitet som kan representere en flaskehals i systemet. Entiteter som vil ha tilgang til ressurser kan plasseres i køer dersom ressursen er opptatt. Tilgang til ressursen styres gjennom at man henter ut entiteter fra køen etter en bestemt køalgoritme når ressursen blir ledig.

En *hendelse* er et tidspunkt hvor modellens tilstand endres. Simuleringsmodellens klokke står stille under endringen av tilstanden. En *aktivitet* er en periode med kjent tidsvarighet. Dette betyr at når aktiviteten starter er slutttidspunktet for aktiviteten kjent. En *forsinkelse* er en periode med ukjent varighet. Ved starten av forsinkelsen er altså ikke tidspunktet for slutten av forsinkelsen kjent.

Det er vanlig at diskrete simuleringsmodeller er stokastiske. Det vil si at deler av modellen er modellert som en statistisk fordeling. På denne måten får modellen en tilfeldig variasjon som gjør at man kan modellere systemer fra den virkelige verden som inneholder en variasjon som kan modelleres ved en sannsynlighetsfordeling. For eksempel kan tidspunktet entiteter ankommer systemet variere.

Hovedårsaken til at man anvender diskret hendelsessimulering er at man ønsker å få en realistisk innsikt i ytelsen til et nytt system, eller endringer til et eksisterende system, uten å måtte gjøre den faktiske implementasjonen. Man kan også få god innsikt i fenomener som det er vanskelig å få innsikt i i den virkelige verden på grunn av at tidsskalaen for fenomenene enten er for stor eller for liten til at det er mulig å observere dem. Man får også muligheten til å prøve ut forskjellige muligheter uten å faktisk implementere det virkelige systemet eller å gjøre endringer til et eksisterende system. Simuleringsmodeller er forenklinger av virkeligheten, det er derfor viktig at man nøye vurderer hvilke forenklinger man gjør slik at modellen er en realistisk representasjon som muliggjør belysning av de ønskede sidene av systemet.

4.2 Diskret hendelsessimulering med J-Sim

J-Sim [27] er en programvarepakke som gjør det mulig å utvikle diskrete hendelsessimulatorerer i Java[1]. J-Sim er inspirert av simuleringsspråket Simula [28].

I J-Sim kan en simuleringsmodell inneholde flere uavhengige *prosesser*. En prosess tilsvarer en dynamisk entitet som beskrevet i delkapittel 4.1. Alle prosessene i en modell har samme tidsoppfatning som kalles *simuleringstid*. Simuleringstiden er 0 ved oppstart og øker under utførelsen av simuleringen. Prosessene må implementere den forhåndsdefinerte metoden *life*. Oppførselen til hver prosess under simuleringen er definert i life. Koden i life kan endre modellens tilstand og kan kun utføres som hendelser, noe som fører til at simuleringstiden ikke endres under utførelsen.

En simulering har en *kalender* hvor de ulike hendelsene til prosessene er lagret. Simuleringen utføres ved at man utfører *steg*. For hvert steg utføres neste hendelse i kalenderen, det vil si at kode fra life-metoden til en prosess utføres. Når kalenderen ikke har flere hendelser slutter simuleringen. Prosesser i J-Sim kan utføre aktiviteter og ha forsinkelser. En aktivitet utføreres ved kall til prosessens *hold*-metode. Den gjør at prosessen for et bestemt antall tidsenheter ikke får eksekvere kode. En hendelse legges til kalenderen på det riktige tidspunktet for når prosessen skal aktiveres igjen. Når et steg igjen aktiverer prosessen fortsetter den å eksekvere kode fra life-metoden der den startet på aktiviteten. En forsinkelse gjør at en prosess passiveres på ubestemt tid. Det til vil si at prosessen slutter å utføre kode fra life-metoden, og at det ikke lagres nye hendelser i kalenderen. En prosess som er passivert kan aktiveres av andre prosesser. En ny hendelse legges da til kalenderen slik at prosessen aktiveres så snart den aktive prosessen enten utfører en aktivitet eller får en forsinkelse.

J-Sim har også andre nyttige funksjoner. En av disse er at *semaforer* [29] som muliggjør gjensidig utelukkelse mellom prosesser og simulering av begrensede ressurser. For eksempel kan en semafor simulere prosessorer i en datamaskin som operativsystemstråder ønsker tilgang til.

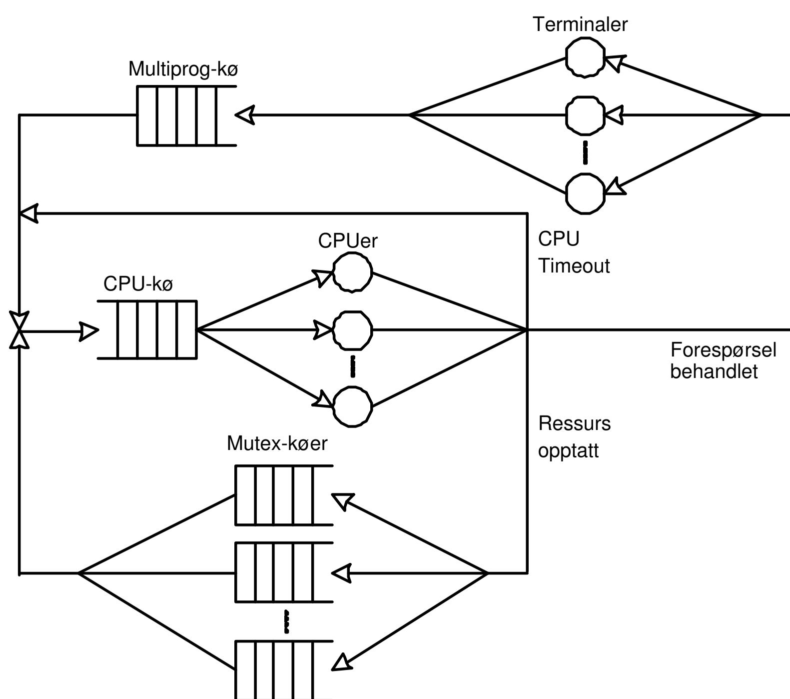
4.3 Simuleringsmodell

Figur 4.1 gir en overordnet oversikt over simuleringsmodellen. Modellen består av et bestemt antall terminaler som sender forespørsler i form av transaksjoner[2] til systemet. Systemet har en *multiprogrammeringsgrense* som definerer antall transaksjoner systemet

[1] J-Sim er fritt tilgjengelig for nedlastning fra prosjektets hjemmeside: http://www.j-sim.zcu.cz/

[2] Transaksjonsbegrepet er noe forenklet i denne sammenhengen. Hver transaksjon utfører kun èn indeksoperasjon, det vil si søking, oppdatering, innsettting eller sletting.

kan behandle samtidig. Når systemet allerede behandler maksimalt antall transaksjoner blir nye transaksjoner satt i kø før de kommer inn i systemet. Hensikten med dette er at det er ønskelig å måle effekten hvor mange transaksjoner man ønsker å slippe til i systemet samtidig. Systemet kan ha èn eller flere prosessorer. Etter at en transaksjonen har kommet inn i systemet må den få tilgang til en prosessor før den kan utføre arbeid. Tilgangen til prosessor skjer gjennom en FIFO-kø. Prosessorene har en forhåndsdefinert *timeslice* som angir hvor lenge en transaksjon kan bruke en prosessor før den får timeout. Når en transaksjon får timeout blir den plassert bakerst i prosessorkøen. En transaksjon kan også slutte å bruke en prosessor på grunn av ressurskonflikter. Det vil si at en transaksjon ønsker tilgang til en ressurs som er beskyttet av en mutex som er opptatt. Transaksjonen blir da plassert i ventekøen til den aktuelle mutexen. Når ressursen blir ledig for transaksjonen blir den igjen plassert i prosessorkøen. Når en transaksjon har fullført sitt arbeid, får terminalen melding om at forespørselen er ferdigbehandlet. Terminalen sender da en ny forespørsel til systemet.

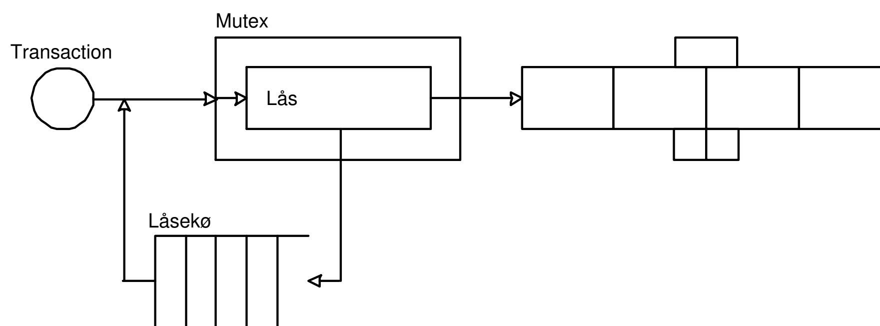


Figur 4.1: Simuleringsmodell

Terminaler og transaksjoner implementeres i J-Sim som *prosesser*. Terminalene starter nye transaksjoner som er de dynamiske entitene som flyter gjennom systemet. Både multiprogrammeringskøen og CPU-køen implementeres som *semaforer* i J-Sim. Dersom man har n prosessorer vil CPU-semaforen ha initiell teller på n. Multiprogrammeringsgrensen implementeres også som en semafor, dersom man for eksempel tillater dobbelt så mange transaksjoner som det er prosessorer i systemet har multiprogrammerings-

semaforen en initiell teller på $2n$.

Som vist i figur 4.2 har nodene i indekstærne i de ulike samtidighetsalgoritmene knyttet til seg en lås. Låsene i simuleringsmodellen støtter låsetypene S-lås, SIX-lås og X-lås. Aksess til å endre hver nodes lås er serialisert ved hjelp av èn mutex per node. Transaksjonene må endre på låsetabellen både når låsene settes og slippes, derfor trenger man for hver lås tilgang til låsens mutex *to ganger*. I J-Sim er mutexene implementert som semaforer med initiell teller på 1. Dersom en transaksjon ønsker å sette en lås på en node som allerede er låst med en låsetype som er inkompatibel med den ønskede låsen slipper transaksjonen mutexen før den passiveres og settes i kø. Dersom den ønskede låsetypen er kompatibel med låsene som allerede er satt, slipper transaksjonen mutexen og får tilgang til noden.



Figur 4.2: Transaksjoners flyt gjennom låsemekanismen for hver node.

Simuleringen av transaksjonenes tidsforbruk, altså aktivitetene de utfører, skjer gjennom kall til hold-metoden i J-Sim. Lengden på hver aktivitet, altså tidskostnaden for å utføre bestemte operasjoner i algoritene som simuleres, er hentet fra resultatene fra microbenchmarkingen i [8]. Tabell 4.1 viser disse resultatene. Tidsforbruket er oppgitt i nanosekunder og er fremkommet fra microbenchmarking på en Intel Celeron 566 med 128MB minne som kjører GNU/Linux.

Operasjon	10^{-9} **sek**
Nøkkelsammenlikning	75
Integersammenlikning	3
Pekertilordning	2
Addisjon	5
Divisjon	64
POSIX-mutex (ta/slipp)	357

Tabell 4.1: Tidsforbruk for de ulike operasjonene

I alle simuleringene anvendes nodestørrelse på 400 elementer. Det vil si at i hver node

i et T-tre er det plass til 400 nøkler, med halenode er det da plass til 800 nøkler. For B-link-trær er det plass til 400 barnpekere i hver node og 400 nøkler i løvnodene. Det antas at tabellen som indekseres har 50.000.000 tupler. Dette fører til at T-trær får, dersom man ser bort fra halenoder, $\lceil log_2(\frac{50.000.000}{400})\rceil = 17$ nivåer. For B-trær får man 3 nivåer. Prosessoren i simuleringene scheduleres med timelice på 10 millisekunder.

4.3.1 Forenklinger

For alle algoritmene er det gjort enkelte forenklinger. Hensikten med forenklingene er å gjøre implementasjonen av simulatoren enklere uten at det får nevneverdige konsekvenser for simuleringsresultatene.

I alle de implementerte algoritmene regnes det med at oppdatering aldri fører til flytting av nøkler mellom noder. Flytting av nøkler vil kunne skje dersom man for eksempel endrer på nøkkelverdien til datatuppelet slik at nøkkelen får en ny avgrensede node. Da dette er noe som sannsynligvis vil skje relativt sjelden i realistiske brukstilfeller, antas det at denne forenklingen ikke vil få stor innvirkning på simuleringsresultatene. Videre utfører man aldri sammenslåing eller sletting av noder som følge av sletting av nøkler. Dette er en forenkling som også har blitt gjort i implementasjon av kommersielle databasesystemer, for eksempel ClustRa[3] [30].

Indeksstrukturene implementeres statiske, det vil si at størrelsen på trærne er forhåndsdefinert, og at transaksjoner aldri endrer på selve trestrukturen. For å simulere effekten av nodesplitt med eventuelle påfølgende trerotasjoner eller omorganiseringer brukes forhåndsdefinerte sannsynligheter. Nodene inneholder ikke noen virkelige nøkler og det tas derfor ikke hensyn til at man søker etter faktiske nøkkelverdier. Når man søker nedover i treet blir neste barnnode valgt tilfeldig fra barnpekerne i noden man er i. Dette antas heller ikke å ha noen innvirkning på resultatene med mindre man ønsker å simulere virkningen av at enkelte deler av treet brukes spesielt hyppig. Belastningen vil altså med denne forenklingen overt tid bli jevnt fordelt utover hele treet.

Nodene antas alltid å inneholde maksimalt antall elementer. Da nøklene nodene ligger i sortert rekkefølge etter nøkkelverdi fører innsetting og sletting[4] av nøkler til at man i gjennomsnitt må flytte halvparten av nøklene internt i en node. Som en forenkling flytter man alltid halvparten av nøklene i en node som følge av innsetting og sletting. Dersom en node er full ved innsetting må man splitte noden, eller omorganisere treet. Da treet er statisk modelleres dette ved hjelp av sannsynligheter for at noder er fulle. Da T-trealgoritmene har halenoder er sannsynligheten mindre for at man har helt fulle noder i T-treet enn for at man har fulle noder i B-treet. I simuleringene antas det

[3] Denne informasjon er ikke publisert, men har fremkommet i samtaler med systemarkitektene. Dette gjengis her med deres tillatelse.

[4] Man kompakterer nøklene etter sletting

at sannsynligheten for full node i B-tre er cirka 3%, og for T-tre cirka 1.5%. Disse sannsynlighetene har framkommet ved tester på T-tre-implementasjonen i Perst[5].

4.3.2 Algoritmene

I dette delkapitlet gjennomgås algoritmene som simuleres. Forenklinger og eventuelle avvik fra de opprinnelige algoritmene i implementasjonen av simuleringsmodellen spesifiseres.

T-tre med optimistisk samtidighetskontroll

Innsettingsalgoritmen er implementert helt likt beskrivelsen i delkapittel 2.2.2 bortsett fra at man ser bort i fra kostnaden knyttet til å vedlikeholde NNP, ENP og DNP. Dette vil trolig ha liten innvirkning på resultatene da pekertilordninger er, som vist i tabell 4.1, en forsvinnende liten kostnad. I tillegg er kostnaden for omorganiseringer statisk satt til 10.000 nanosekunder. Årsaken til at den er satt statisk er at omorganisering er en relativt avansert operasjon som ville ført til en omfattende implementasjon, og at omorganisering trolig skje relativt skjeldent. Transaksjoner som ønsker å gjøre omorganisering på et tre som allerede er har en transaksjon som venter på å gjøre omorganisering restartes.

Optimistisk samtidighetskontroll med delvis låsing

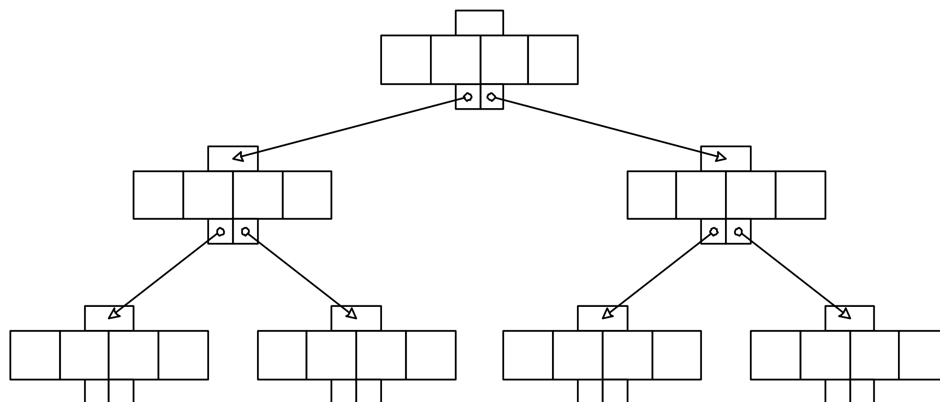
For å presentere resultater fra en fungerende optimistisk algoritme for T-trær simuleres også T-tre med delvis låsing. Søke- og oppdateringsalgoritmen er implementert som beskrevet i delkapittel 2.2.3. Også for denne algoritmen ser man bort ifra kostnaden knyttet til å vedlikeholde NNP, ENP og DNP. Man setter kostnaden for omorganiseringer statiskt til 10.000 nanosekunder.

T-tre med pessimistisk samtidighetskontroll

Søke- og oppdateringsalgoritmen for pessimistisk samtidighetskontroll er implementert som beskrevet i delkapittel 2.2.2. Algoritmen for sletting er noe forenklet da man i simulatoren har antatt at man ved underflyt i noder aldri sletter noder. Algoritmen for sletting blir da helt lik algoritmen for oppdatering, bortsett fra at man også må flytte halvparten av pekerne i noden for å kompaktere pekerne.

[5]Perst er en Java-basert objektorientert database som er fritt tilgjengelig for nedlastning til ikke-kommersielle formål på www.garret.ru/ knizhnik/perst.html.

Da selve treet er implementert statisk, det vil si at innsettinger og slettinger ikke gjør faktiske endringer på treet, må innsettingsalgoritmen også inneholde enkelte forenklinger. I algoritmen holder man alltid ved søk nedover i treet SIX-låser fra forelderen til den kritiske noden og nedover. Da treet er statisk, og alltid er helt fullt, vil det aldri være noen kritisk node. Figur 4.3 illustrerer et typisk T-tre lagret internt i simulatoren. For å simulere effekten av kritiske noder anvendes en bestemt sannsynlighet for at hver node på veg nedover i treet er kritisk. Det antas at på veg nedover er i gjennomsnitt *hver tredje* node kritisk. For denne sannsynligheten er det gjort konservativt anslag da det er vanskelig å beregne noen nøyaktig sannsynlighet for at en node er kritisk.



Figur 4.3: T-tre internt i simulatoren, alle noder har maksimalt antall barnnoder helt ned til løvnivået. På grunn av dette har ikke treet noen kritisk node.

Etter at man har splittet en node på grunn av at den avgrensende noden var helt full må man i enkelte tilfeller gjøre rotasjoner i treet. Dette skjer dersom den kritiske noden har fått forskjell i høyde på subtrærne som er større enn 1. Da treet er statisk må også dette simuleres ved hjelp av sannsynligheter. Sannsynligheten for dette er maksimalt 0.5 og minimalt $\frac{1}{2^n}$ hvor n er antallet nivåer under den kritiske noden i treet. Disse sannsynlighetene er gyldig dersom hver innsetting skjer på en tilfeldig node i treet. For å gjøre et konservativt anslag velges denne sannsynligheten til 0.5.

ARIES/IM

ARIES/IM simuleres som beskrevet i delkapittel 2.3.2. Man antar alltid at man ikke trenger å følge høyrepekeren når man kommer til løvnoden, det vil si at man kun simulerer selve sjekken av hvilken løvnode nøkkelen ligger i og at sjekken alltid viser at den ligger i den første løvnoden man kommer til. I tillegg fører sletting aldri til sletting av løvnoder. Ved nodesplitt som følge av innsetting propageres endringene kun ett nivå oppover i treet. Årsaken til denne forenklingen er sannsynligheten for at man må propagere høyere

opp er meget liten. Ved nodesplitt må man låse tre-låsen med X-lås, men man simulerer ikke bruk av *SM_Bit* og *Delete_Bit*, følgelig trenger man aldri å ta tre-låsen med S-lås i slette- og oppdateringsoperasjoner. Årsaken til at man ikke simulerer *SM_Bit* og *Delete_Bit* er at effekten av disse trolig vil være veldig liten i simuleringsresultatene.

B-tre med forbikjøring

B-trær med forbikjøring simuleres som beskrevet i delkapittel 2.3.3. Man antar også her at man ikke trenger å følge høyrepekeren når man kommer til løvnoden og at man kun simulerer selve sjekken av hvilken løvnode nøkkelen ligger i. Ved nodesplitt som følge av innsetting propageres endringene kun ett nivå oppover i treet. Komprimeringsprosessen for å fjerne noder med underflyt implementeres ikke.

4.3.3 Parametre i simuleringsmodellen

Simuleringsmodellen har flere parametre det er naturlig å variere under simuleringene. Hensikten med dette er å se hvordan de forskjellige algoritmene yter under forskjellige omgivelser. I dette delkapitlet beskrives de ulike parametrene som kan varieres.

Multiprogrammeringsgrense

For å se effekten av hvor mange transaksjoner som tillattes i systemet samtidig vil det være naturlig å observere effekten av å variere multiprogrammeringsgrensen. Multiprogrammeringsgrensen vil alltid være større enn antall prosessorer i systemet. Det vil kunne være nyttig å se om man kan oppnå noen ytelsesfordeler av å ha flere transaksjoner i systemet enn det er prosessorer. For å øke multiprogrammeringsgrensen økes den initielle telleren til multiprogrammeringssemaforen i simuleringsmodellen.

Andelen av ulike transaksjonstyper

I simuleringsmodellen kan man definere hvor stor andel av det totale transaksjonsantallet transaksjonstypene søking, oppdatering, innsetting og sletting skal ha. Hver gang en terminal starter en ny transaksjon bestemmes sannsynligheten for hvilken transaksjonstype det skal være utifra andelstallene. Hensikten med å variere dette parameteret er å observere hvor godt algoritmene yter under ulike realistiske brukstilfeller.

Antall prosessorer

Det er ønskelig å se effekten av å øke prosessorantallet for å se hvor godt hver algoritme skalerer med hensyn til antall prosessorer. For å øke antallet prosessorer økes den initielle telleren til semaforen som representeres prosessoren i simuleringsmodellen.

Kostnaden på bruk av mutexer

Microbenchmarkene i [8] viste at kostnaden på bruk av låser kan være langt lavere enn det som ble vist av Lu, Ng og Tian i [7]. Da resultatene fra disse studiene er så forskjellige, er det av interesse å se hvor sensitiv hver algoritme er for låsekostnad, og se om det relative ytelsesforholdet mellom dem endres ved varierende låsekostnad. I tillegg ble mutexkostnaden i [8] benchmarket på en maskin med èn prosessor. Denne kostnaden kan være misvisende for datamaskiner med flere prosessorer da bruk av mutexer fører til at man må invalidere hurtigminne i alle prosessorene. Det er derfor også interessant å se effekten av høyere mutexkostnad for datamaskiner med flere prosessorer.

4.3.4 Valg av multiprogrammeringsgrense

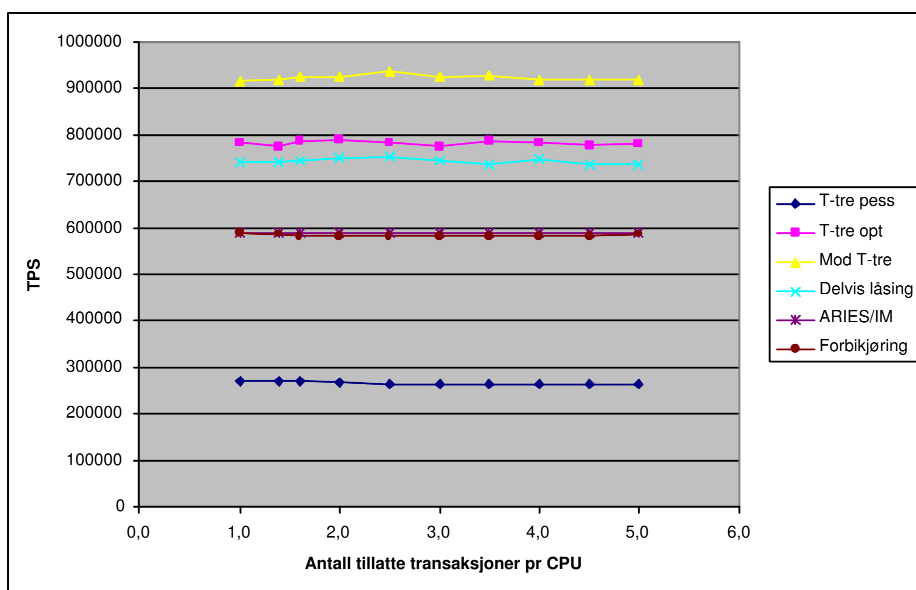
Ved bruk av flere prosessorer vil ressurskonflikter kunne føre til at enkelte transaksjoner må vente på ressurser i kortere eller lenger tid. Dersom man ikke tillater flere transaksjoner i systemet enn det er prosessorer, vil prosessorer kunne være passive mens transaksjoner venter på ressurser. For å unngå dette er det nærliggende å ha flere transaksjoner i systemet enn det er prosessorer. I dette delkapitlet undersøkes det hvor mange transaksjoner det lønner seg å ha per prosessor. Resultatene fra undersøkelsene anvendes i simuleringene som ligger til grunn for resultatene som presenteres kapittel 5.

Det blir utført to forsøk hvor man simulerer en datamaskiner med 4 og 8 prosessorer. Man utfører en miks av de ulike transaksjonstypene med 50% søking, 40% oppdatering, 5% innsetting og 5% sletting. Man varierer antallet tillatte transaksjoner fra like mange som det er prosessorer i systemet til 5 ganger så mange. Det er like mange terminaler som sender forespørsler til systemet som det er tillatte transaksjoner i systemet. Terminalene har ingen tenketid mellom forespørslene. For hvert forsøk registreres antall transaksjoner per sekund.

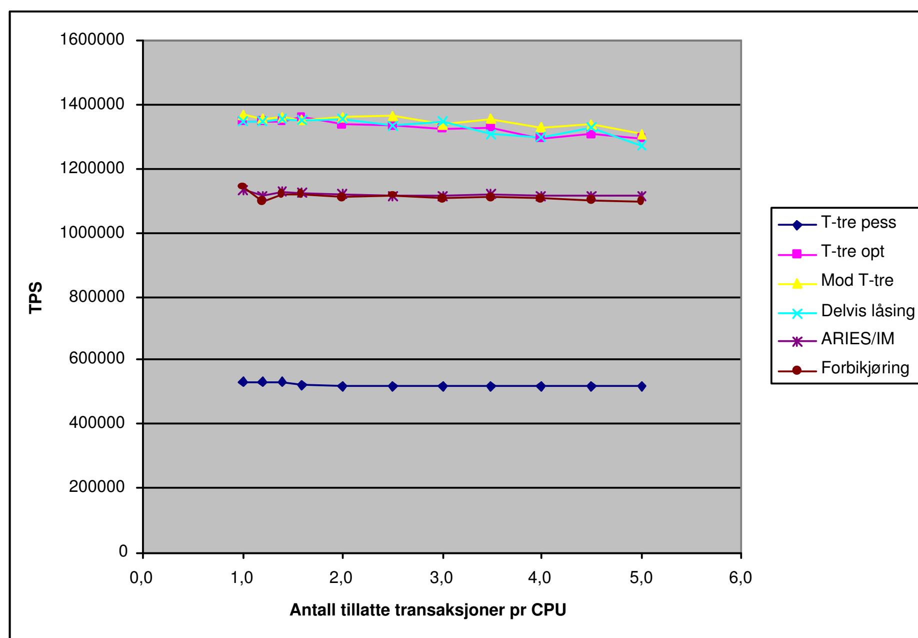
Tabell 4.4 og 4.5 viser antall transaksjoner per sekund for henholdsvis 4 og 8 prosessorer. Man ser at ingen av algoritmene får noen stor oppgang eller nedgang som følge av høyere antall transaksjoner i systemet. Man klarer med andre ord ikke å utnytte eventuelle passive perioder på prosessorene til å oppnå høyere throughput. Årsaken til

dette er trolig at transaksjonene bruker veldig kort tid i indeksen i forhold til størrelsen på prosessorintervallene. Man vil derfor veldig sjeldent ha noen transaksjoner som venter nede i indeksen siden flaskehalsene som transaksjonene venter på enten er rotnoder eller mutexer som beskytter hjelpevariablene. Rotnodene aksesseres bare i starten av en algoritme, og mutexen som beskytter hjelpevariablene aksesseres i starten og slutten av algoritmene. Dersom en ny transaksjon får slippe til på en ledig prosessor, er sannsynligheten stor for at den også må vente på den samme ressursen som de andre transaksjonene.

Som følge av disse resultatene vil man i simuleringene i kapittel 5 bruke èn transaksjon per prosessor.



Figur 4.4: Transaksjoner per sekund for varierende multiprogrammeringsgrad med 4 prosessorer.



Figur 4.5: Transaksjoner per sekund for varierende multiprogrammeringsgrad med 8 prosessorer.

Kapittel 5

Resultater

I dette kapitlet prenteres resultateter fra de ulike simuleringene. Simuleringsmodellen som brukes er dokumentert i kapittel 4. For hver simulering diskuteres først motivasjonen for simuleringen og testoppsettet spesifiseres. Lu, Ng og Tian fokuserte i [7] på totalt anvendt tid, eller *throughput* når simuleringsresultatene for algoritmene ble presentert. Som vist i delkapittel 3.2 er dette en noe mangelfull presentasjonsform. For å bedre avdekke algoritmenes ytelse fokuseres det defor her både på *gjennomsnittlig responstid* og *throughput* i presentasjonen av resultatene. På denne måten avdekker man både hvor responsivt terminalene oppfatter systemet og hvor stor gjennomstrømning man har av transaksjoner.

I delkapittel 5.1 måles effekten av å variere antall prosessorer i systemet. Simuleringene utføres under flere forskjellige omgivelser. I delkapittel 5.2 presenteres fra simulering av algoritmene med varierende mutexkostnad. I delkapittel 5.3 diskuteres resultatene.

5.1 Varierende prosessorerantall

Hovedmotivasjonen for å utføre simuleringene med varierende prosessorantall er at et stort antall av dagens databasesystemer kjører på maskiner med flere prosessorer. Derfor er det naturlig å teste ytelsen i slike omgivelser.

Som vist i delkapittel 3.2 vil throughput være uavhengig av ressurskonflikter dersom systemet man simulerer bare har èn prosessor. Dette er ikke tilfellet for systemer med flere prosessorer; dersom det blir ressurskonflikter kan èn eller flere prosessorer bli passive mens den aktuelle ressursen er opptatt. Dette vil føre til nedsatt samtidighet og derfor også lavere throughput. Med andre ord vil man ved å variere prosessorantallet bedre

finne ut hvilke algoritmer som tillater høy grad av samtidighet, og hvorvidt de anvender låser på en effektiv måte. Algoritmer som tillater høy grad av samtidighet vil tillate mange transaksjoner per sekund (TPS), og TPS vil øke når man øker prosessorantallet. Man kan også si at algoritmene da *skalerer* bra med tanke på antall prosessorer. Da låsekostnaden er relativt høy kan man forvente at algoritmer som anvender låser på en effektiv måte *både* vil ha høy TPS og lav responstid for et stort antall prosessorer.

Da indeksene ligger lagret i primærminnet er det stor mulighet for at de ulike algoritmene vil bruke så kort tid at de blir ferdig i løpet av ett tidsintervall på prosessoren. Dersom man kjører algoritmene på en maskin med èn prosessor vil man trolig ha svært lite reell samtidighet i indeksene siden transaksjonene gjør seg ferdig før noen andre transaksjoner får slippe til. Simulering med flere prosessorer vil derfor ha bedre mulighet til å simulere effekten av virkelig samtidighet i indeksene siden man kan ha transaksjoner som kjører på forskjellige prosessorer samtidig i indeksen.

5.1.1 Testoppsett

For hvert forsøk varieres antall prosessorer fra 1 til 32. Det like mange terminaler som sender forespørsler til systemet som det er prosessorer. Terminalene har ingen tenketid mellom forespørslene.

5.1.2 Omgivelse 1: OLAP

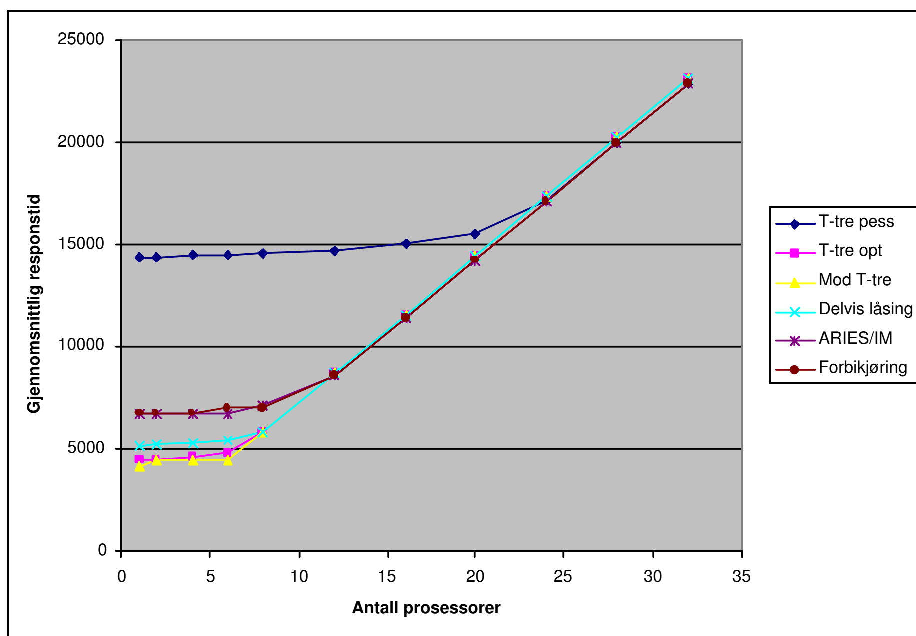
OLAP[1]-databaser, som for eksempel datavarehus, brukes vanligvis kun til analyse av data [9, 31]. På denne typen databaser kan det være hensiktsmessig å optimalisere aksessmetodene for søking, da oppdateringer kun skjer batchvis eller ved total gjenoppbygging av databasen. For å måle hvor godt de forskjellige algoritmene yter under slike omgivelser utfører man kun søkeoperasjoner under simuleringene. Det vil si alle transaksjonene som terminalene starter er søketransaksjoner.

Figur 5.1 viser gjennomsnittlige responstider for algoritmene. Man ser at de tre algoritmene for T-trær som anvender hjelpevariable beskyttet av en mutex[2] er de som har lavest responstid når man har færre enn 12 prosessorer. Man ser at modifisert T-tre yter marginalt bedre enn T-tre med optimistisk samtidighetskontroll og delvis låsing. Ved bruk av disse algoritmene oppnår man tilnærmet like god responstid for 1 til 6 prosessorer, mens responstiden begynner å stige relativt kraftig når man bruker 8 prosessorer. Ved

[1] On-Line Analytical Processing

[2] Disse algoritmene (T-tre med optimistisk samtidighetskontroll, T-tre med delvis låsing og modifisert T-tre) omtales heretter som *de optimistiske algoritmene*.

økning i antall prosessorer stiger responstiden til alle algoritmene lineært med tilnærmet likt stigningstall.

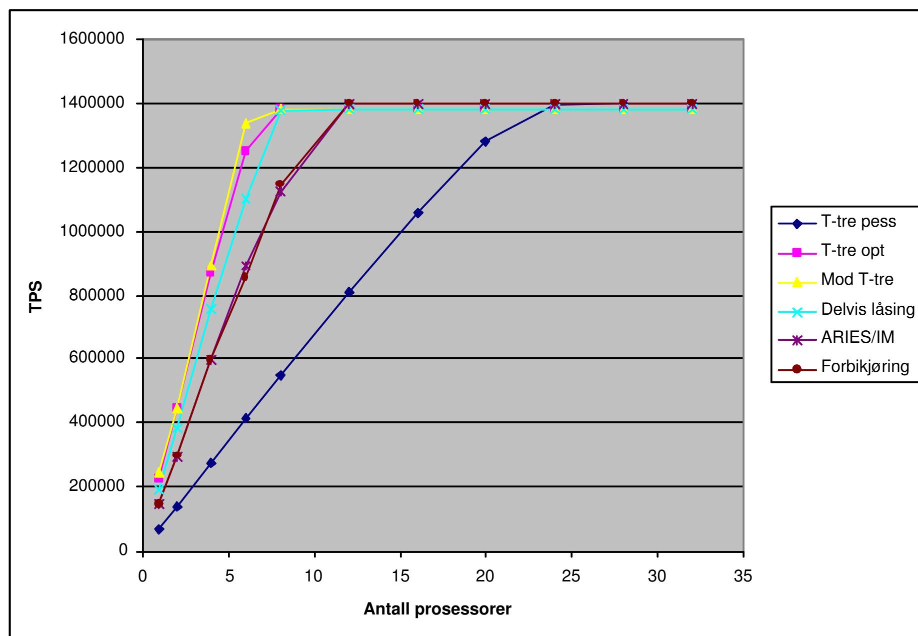


Figur 5.1: Responstider ved 100% søking (OLAP).

De to B-tre-algoritmene yter tilnærmet helt likt. De yter noe dårligere enn de tre optimistiske algoritmene, men responstiden er stabil til et høyere antall prosessorer. Man ser at den har tilnærmet lik responstid for 1 til 8 prosessorer, men fra 12 prosessorer og oppover stiger responstiden like bratt som for de tre optimistiske algoritmene. T-tre med pessimistisk samtidighetskontroll har klart dårligst ytelse, men responstiden er relativt stabil inntil 20 prosessorer. Fra 24 prosessorer og oppover stiger responstiden like bratt som for de andre algoritmene. Selv om de optimistiske algoritmene bare skalerer opp til 6 prosessorer ser man at disse algoritmene alltid har lavest eller tilnærmet lik responstid som de andre algoritmene. Man kan derfor si at de optimistiske algoritmene oppnår høyest mulig ytelse med færrest prosessorer i denne omgivelsen. Årsaken til at at responstiden til alle algoritmene er lik og stiger med samme stigningstall når man har tilstrekkelig mange prosessorer, beskrives nærmere under behandlingen av throughput i de neste avsnittene.

Man ser i figur 5.2 som viser antall transaksjoner per sekund for de ulike algoritmene de samme trendene som for responstidgrafene. De optimistiske algoritmene har flere transaksjoner per sekund enn de andre algoritmene opp til 12 prosessorer, men etter hvert som antallet prosessorer i systemet stiger, blir antallet transaksjoner per sekund tilnærmet likt for alle algoritmene, cirka rundt 1.400.000 transaksjoner. Man ser altså

at alle algoritmene har en felles øvre grense for TPS, men at de optimistisk algoritmene når denne grensen ved bruk av færrest prosessorer.



Figur 5.2: Transaksjoner per sekund ved 100% søking (OLAP).

Årsaken til at ingen av algoritmene skalerer lenger enn til 1.400.000 TPS er at for alle algoritemene er det èn mutex alle transaksjonene må gjennom *to ganger*. For de optimistiske algoritmene er dette mutexen som beskytter hjelpevariablene *count* og $fixing$ mens for B-trær og T-tre med pessimistisk samtidighetskontroll er dette mutexen som beskytter låsen på rotnoden. Når man har tilstrekkelig mange prosessorer blir denne mutexen en flaskehals som forhindrer videre skalering. Kostnaden ved å ta og slippe mutexer, C, er 357 nanosekunder. Dersom T representerer hvor lang tid flaskehalsmutexen er i bruk totalt og n er antall transaksjoner får man:

$$T = 2 \cdot C \cdot n = 2 \cdot 357 \cdot 10^{-9} \cdot 1.400.000 = 0.9996$$

Dette viser at ved 1.400.000 transaksjoner per sekund er mutexen i bruk i nesten hele tiden, og dermed er det ikke mulig å oppnå høyere ytelse ved bruk av flere prosessorer. Årsaken til at flaskehalsmutexen i T-tre med optimistisk samtidighetskontroll og B-trær, altså rotnoden, når metningspunktet senere enn mutexen i de optimistiske algoritmene er at disse algoritmene bruker lenger tid på traverseringen av treet, og dermed får man mindre press på den delte ressursen.

Når flaskehalsene oppstår ser man i figur 5.1 at alle algoritmene får tilnærmet lik responstid og at responstidene stiger med likt stigningstall ved økning av antall prosessorer. Dette skjer til tross for at de forskjellige algoritmene har forskjellig ytelse før flaskehalsene oppstår. Årsaken til dette er at forskjellen i ytelse jevnes ut med at man må vente på flaskehalsmutexen. De algoritmene som er mest effektive før flaskehalsene oppstår har lengst mutexkø da de bruker kortere tid i indeksen. De minst effektive algoritmene har kortest kø siden det er flere transaksjoner i indeksen.

5.1.3 Omgivelse 2: Oppdatering av datavarehus

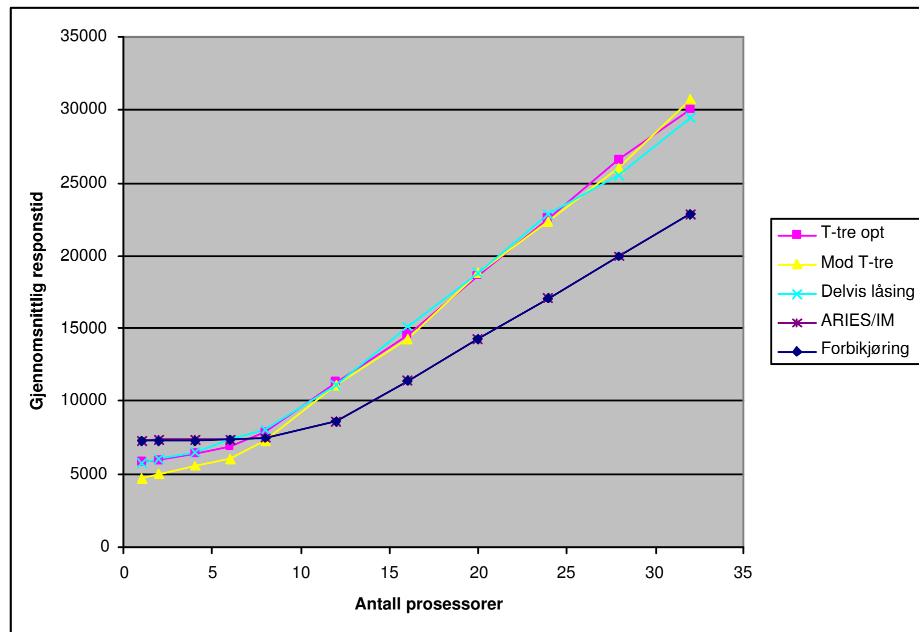
Datavarehus inneholder store mengder informasjon. Informasjonen legges gjerne inn batchvis, det vil at man periodisk legger inn nye data i databasen. Under slike periodiske innsettinger, hvor databasen gjerne er stengt for andre operasjonstyper, kan det være hensiktsmessig å ha aksessmetoder som er optimialisert for å støtte høy grad av innsettingsoperasjoner i databasen. For å måle ytelsen til de ulike algoritmene under slike omgivelser utføres simuleringen med kun innsettingstransaksjoner.

Figur 5.3 viser gjennomsnittlige responstider for algoritmene. T-tre med pessimistisk samtidighetskontroll både yter og skalerer klart dårligst. For å beholde en skala som gjør grafene lesbare i figuren er grafen for denne algoritmen tatt ut. For èn prosessor bruker T-tre med pessimistisk samtidighetskontroll gjennomsnittlig 14900 nanosekunder og for 32 prosessorer har algoritmen en gjennomsnittlig responstid på cirka 140.000 nanosekunder.

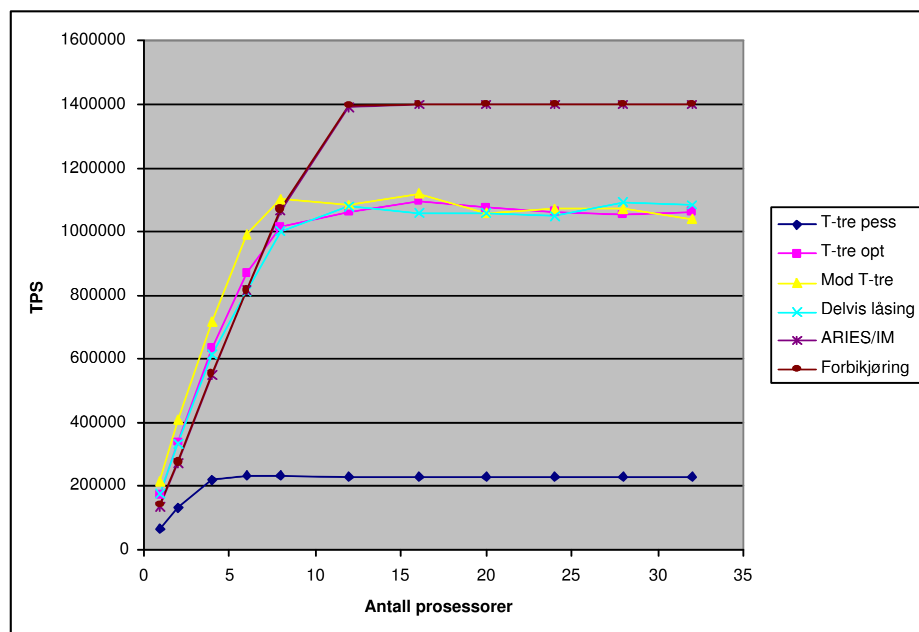
Man ser at for færre enn 8 prosessorer har modifisert T-tre noe lavere responstid enn de to andre optimistiske algoritmene. ARIES/IM og B-tre med forbikjøring har noe høyere responstid enn disse igjen. For 8 og flere prosessorer har de tre optimistiske algoritmene tilnærmet lik responstid. For flere enn 8 prosessorer er det B-tre-algoritmene som har lavest responstid av samtlige algoritmer. Man ser forøvrig også her at de to B-tre-algoritmene har tilnærmet helt lik ytelse.

I figur 5.4 ser man antall transaksjoner per sekund for alle algoritmene. Her ser man den klart underlegne ytelsen til T-trær med pessimistisk samtidighetskontroll. Hovedårsaken til at denne algoritmen yter så dårlig under disse omgivelsene er at man ved traverseringen av treet alltid antar at rotnoden er kritisk og følgelig hoder SIX-lås på denne noden helt til man har funnet en ny kritisk node. Dette kan ta relativt lang tid på grunn av at man må låse på hvert nivå man besøker i treet. Så lenge rotnoden er låst med SIX-lås, kan ingen andre innsettingstransaksjoner starte traverseringen av treet.

De tre optimistiske algoritmene skalerer relativt lineært opp til 6−8 prosessorer, her flater transaksjoner per sekund ut for alle de tre algoritmene rundt 1.050.000 transaksjoner per sekund. Årsaken til at disse algoritmene skalerer dårligere for innsetting enn for søking er at man låser hele treet dersom noder er helt fulle. Dette fører til at kun en prosessor kan



Figur 5.3: Responstider ved 100% innsetting. T-tre med pessimistisk samtidighetskontroll er tatt ut på grunn av lesbarhet.



Figur 5.4: Transaksjoner per sekund ved 100% innsetting.

bruke treet, mens resten av prosessorene er passive. I simuleringsmodellen skjer dette gjennomsnittlig for hver 64 innsettingstransaksjon. Da sannsynligheten for for fulle noder er satt på forhånd ved hjelp av en sannsynlighetsvariabel, vil det være knyttet en viss usikkerhet til hvorvidt throughputen i en virkelig indeks faktisk vil flate ut nøyaktig ved 1.050.000 TPS. Likevel viser denne simuleringen at på grunn av de optimistiske algoritmenes låsing av hele treet vil de trolig ikke kunne oppnå like høy TPS som B-trær.

B-trærne skalerer tilnærmet like bra for innsetting som for søking. Hovedårsaken til dette er at indeksen ikke holder flaskehalsressursen, altså mutexen på rotnoden, lenger enn det den gjør for søking. Transaksjoner per sekund flater derfor ut på rundt 1.400.000 TPS ved 12 prosessorer, som er det samme som for søking. En medvirkende årsak til at algoritmen skalerer så bra i simuleringen er at en forenkling i simuleringsmodellen gjør at de strukturelle endringene i treet aldri går lenger opp enn ett nivå over rotnoden. Dersom simuleringsmodellen hadde implementert et virkelig dynamisk tre ville noen endringer ført til at man måtte låse rotnoden over en tidsperiode mens man gjorde strukturelle endringer, men på grunn av B-treets egenskaper ville dette trolig skjedd relativt sjelden. Resultatet for B-trealgoritmene er derfor sannsynligvis ikke veldig forskjellig fra det man ville hatt i et dynamisk tre. Under denne omgivelsen kan man altså si at man med B-trealgoritmene vil kunne oppnå høyest throughput. Dersom man har færre enn 8 prosessorer vil de optimistiske algoritmene ha lavest responstid og høyest throughput.

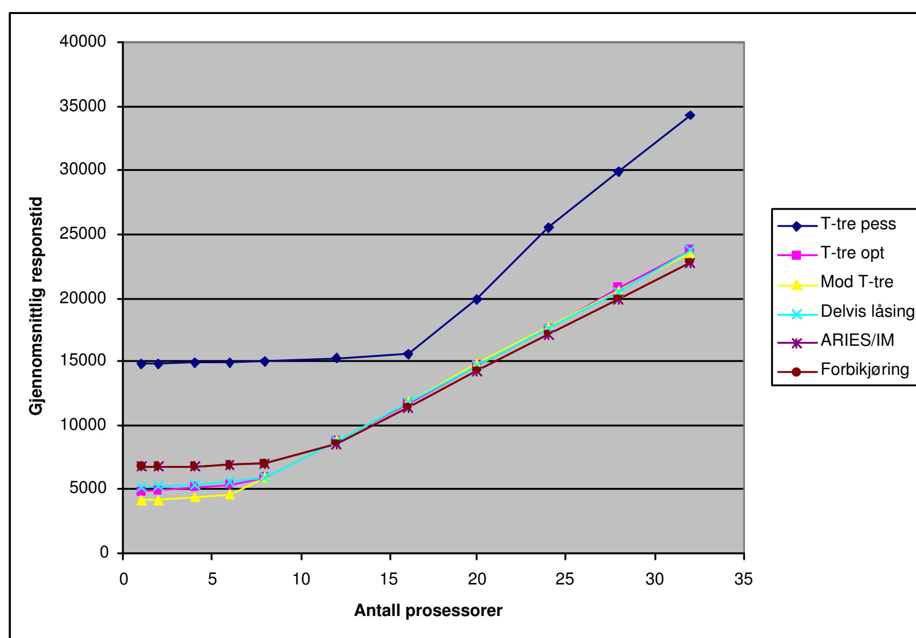
5.1.4 Omgivelse 3: Miks 1

For å måle ytelsen til algoritmene i omgivelser hvor størsteparten av operasjonene er enten søking eller oppdatering, utføres simuleringen med en miks av de ulike operasjonstypene. 50% av operasjonene er søking, 40% er oppdatering, 5% er innsetting og 5% er sletting. Dette er en omgivelse som er representativ for en lang rekke bruksområder for databaser. For eksempel kan dette være en realistisk omgivelse for en bedrifts kundedatabase; ved kontakt med kunder brukes søkefunksjonen for å få tilgang til kundeinformasjon og for å gjøre endringer på en kundes profil brukes oppdateringsfunksjonen. For å opprette nye, eller slette kunder brukes innsetting eller sletting henholdsvis.

Figur 5.5 viser responstider og figur 5.6 viser transaksjoner per sekund for algoritmene. Man ser at ytelsestrenden er relativt lik trenden for OLAP-omgivelsene i delkapittel 5.1.2. T-tre med pessimistisk samtidighetskontroll er den eneste algoritmen som har stor forskjell i ytelsen fra OLAP-omgivelsene i delkapittel 5.1.2, algoritmen skalerer nå langt dårligere. Grunnen til dette er at man for innsetting og sletting (10% av operasjonene) låser rotnoden med SIX-lås helt til man har funnet en ny kritisk node. Dette fører til nedsatt samtidighet for andre transaksjoner som utfører innsetting og sletting.

Man ser at de optimistiske algoritmene, med modifisert T-tre i spissen, har lavest re-

sponstid og høyest TPS for opptil 8 prosessorer. Disse algoritmene skalerer nesten lineært opptil dette prosessorantallet. ARIES/IM og B-tre med forbikjøring har noe dårligere ytelse enn de optimistiske algoritmene for opp til 8 prosessorer, men skalerer videre opp til 12 prosessorer hvor algoritmen marginalt har den høyeste throughputen. Årsaken til at B-treet skalerer noe bedre enn de optimistiske algoritmene er at når man har innsetting i helt fulle noder låser de optimistiske algoritmene hele treet.

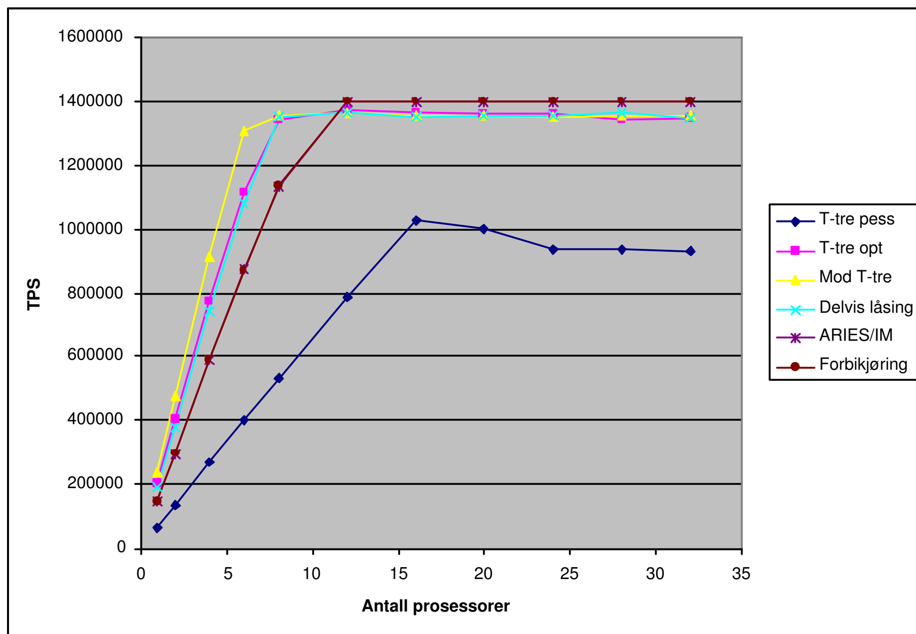


Figur 5.5: Responstider ved 50% søking, 40% oppdatering, 5% innsetting og 5% sletting.

5.1.5 Omgivelse 4: Miks 2

I [7] testet man ytelsen til aksessmetodene ved å simulere en datamaskin med èn prosessor hvor man utførte 80% søking, 10% innsetting og 10% sletting. B-tre med forbikjøring ble sammenliknet med T-tre med optimistisk samtidighetskontroll og T-tre med pessimistisk samtidighetskontroll. For å bedre kunne sammenlikne resultatene i dette prosjektet med resulatatene i [7], testes algoritmene under de samme omgivelsene. Som for operasjonsmiksen i forrige delkapittel representerer også denne miksen realistiske omgivelser for en lang rekke databaser.

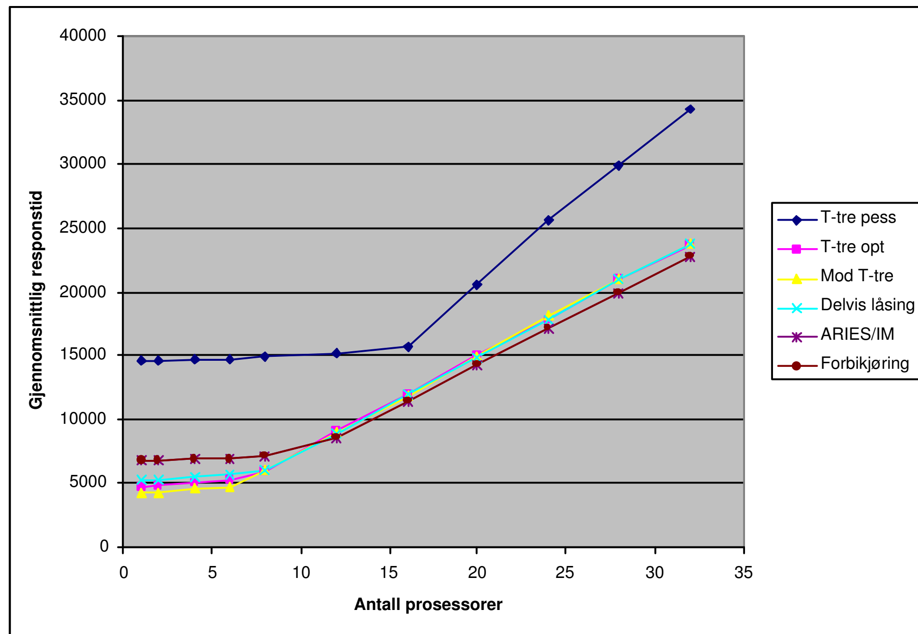
Man ser i figur 5.7 og 5.8 som viser responstider og transaksjoner per sekund, at resultatet er tilnæmet identisk med resultatene i forrige delkapittel. Der var det 50% søking og 40% oppdatering. Søking og oppdatering er tilnærmet identiske operasjoner. Den eneste



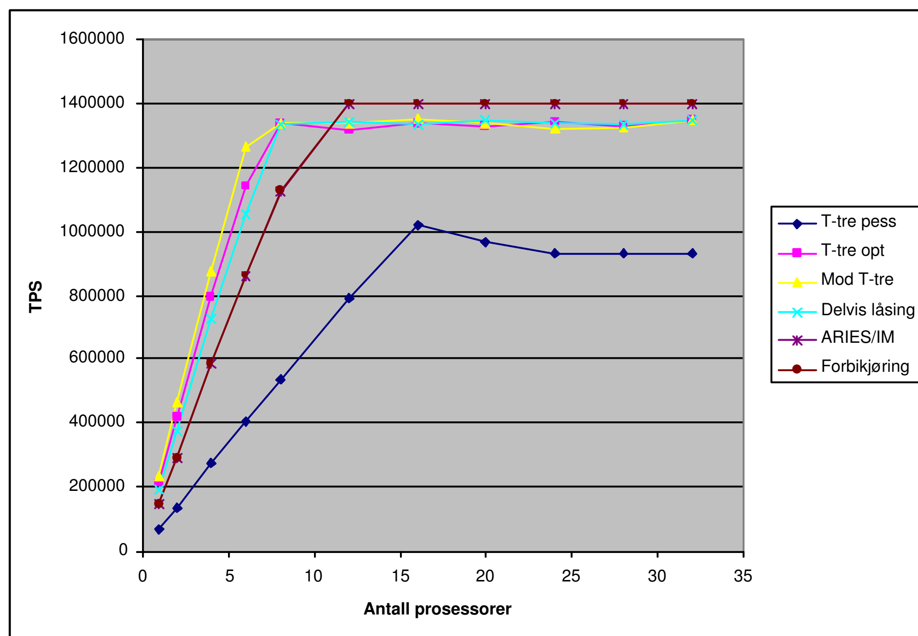
Figur 5.6: Transaksjoner per sekund ved 50% søking, 40% oppdatering, 5% innsetting og 5% sletting.

forskjellen er at man låser den avgrensende noden med X-lås i stedet for S-lås. Dette har som figurene viser liten innvirkning på ytelsen. Den større andelen av innsetting og sletting har også minimal innvirkning på responstiden til algoritmene.

I [7] utførte man som sagt simuleringer med den samme operasjonsmiksen og totalt anvendt tid, altså througput, ble presentert. Da denne simuleringen kun ble utført for datamaskiner med èn prosessor og er mangelfullt dokumentert er det vanskelig å gjøre noen detaljert sammenlikning med resultatene i figur 5.8. I simuleringene i [7] var det trolig flere samtidige transaksjoner i indeksen, mens i simuleringene som ligger til grunn for figur 5.8 har det bare vært èn transaksjon per prosessor samtidig. Denne forskjellen har som det ble vist i delkapittel 3.2 ingen innvirkning på throughput så lenge det bare er èn prosessor i systemet. Resultatene i [7] viste at B-tre med forbikjøring hadde cirka 3 ganger høyere throughput enn T-tre med optimistisk samtidighetskontroll, og cirka 35 ganger høyere throughput enn T-tre med pessimistisk samtidighetskontroll. Man ser at disse resultatene ikke stemmer overens med resultatene i figur 5.8. Her ser man at for èn prosessor har T-tre med optimistisk samtidighetskontroll cirka 30% høyere throughput enn B-trær med forbikjøring, mens den har cirka 70% høyere throughput enn T-tre med pessimistisk samtidighetskontroll. Hovedforskjellen fra resultatene i [7] er altså at T-tre med optimistisk samtidighetskontroll yter bedre enn B-trær. Det er vanskelig å finne noen sikker forklaring på dette, men en mulig årsak kan være at Lu, Ng og Tian har



Figur 5.7: Responstider ved 80% søking, 10% innsetting og 10% sletting.



Figur 5.8: Transaksjoner per sekund ved 80% søking, 10% innsetting og 10% sletting.

implementert B-tre med forbikjøring i [7] som beskrevet i [18]. Det vil si uten at det settes noen låser på noder som ikke skal endres. Som vist i delkapittel 2.3.3 gir trolig ikke denne algoritmen alltid forutsigbar oppførsel i MMDB. En annen forskjell er at T-tre med pessimistisk samtidighetskontroll ikke lenger er fullt så underlegen som vist i [7]. Årsaken til dette er at simuleringene som ligger til grunn for figur 5.8 opererer med en langt lavere låsekostnad.

5.2 Varierende mutexkostnad

Kostnaden knyttet til bruk av mutexer er noe usikker. I [7] brukte man semaforer som viste seg å være meget kostbare, mens man i [8] dokumenterte at bruk av mutexer på ènprosessorsystemer er relativt billig. Bruk av mutexer på multiprosessorsystemet kan også være mer kostbart enn det er på ènprosessorsystemer på grunn av at man må sørge for at hurtigminnet på de ulike prosessorene har et enhetlig bilde av mutexene, noe som kan føre til hyppig hurtigminneinvalidering. Invalidering av hurtigminnet er en kostbar operasjon [32]. For å undersøke effekten av varierende mutexkostnad simuleres algoritmene med varierende mutexkostnad. Mutexkostnaden som brukes i simuleringene i delkapittel 5.1 på 357 nanosekunder ble funnet i [8]. Da man ved hjelp av bedre hardwarestøtte for mutexer kan få mer effektive mutexer simuleres også effekten av lavere mutexkostnad.

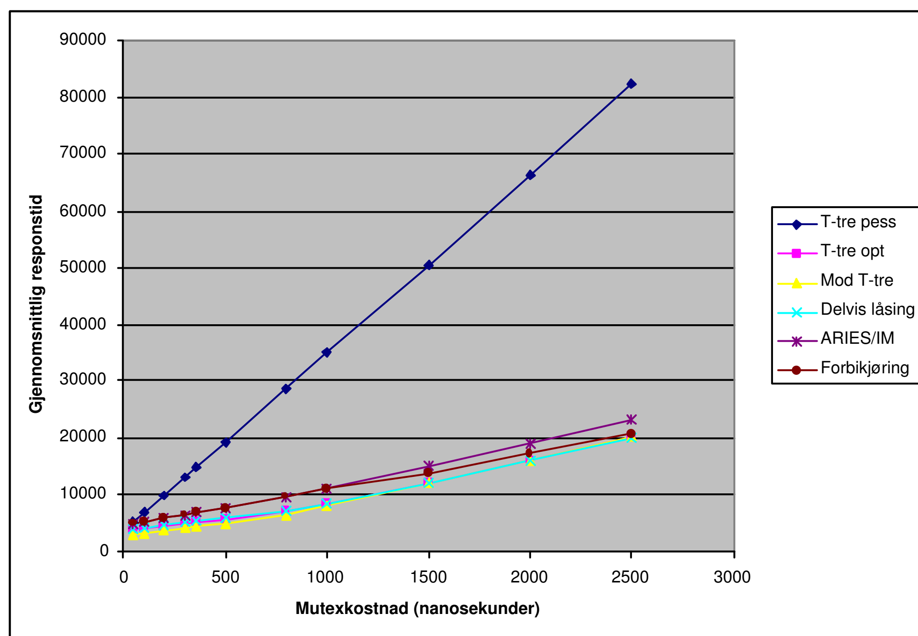
5.2.1 Testoppsett

For hvert forsøk varieres mutexkostnaden fra 50 til 2500 nanosekunder. Simuleringen gjennomføres to ganger; èn gang for 4 prosessorer og èn gang for 16 prosessorer. Man gjør dette for å se om algoritmene reagerer forskjellig på varierende mutexkostnad med ulikt prosessorantall. Det er like mange terminaler som det er prosessorer og terminalene har ingen tenketid mellom forespørslene. Man utfører 50% søking, 40% oppdatering, 5% innsetting og 5% sletting.

5.2.2 Omgivelse 1: 4 prosessorer

Figur 5.9 viser responstidene fra simuleringen for 4 prosessorer. Man ser at for mutexkostnader på opp til 1000 nanosekunder har modifisert T-tre noe lavere responstid enn de andre optimistiske algoritmene. For høyere mutexkostnader er responstiden tilnærmet lik for de tre optimisiske algoritmene. Årsaken til dette er at etterhvert som mutexkostnaden blir høyere, dominerer den så mye at den effektive traverseringen til modifisert T-tre ikke lenger har noen stor effekt. De to B-tre-algoritmene yter også her tilnærmet

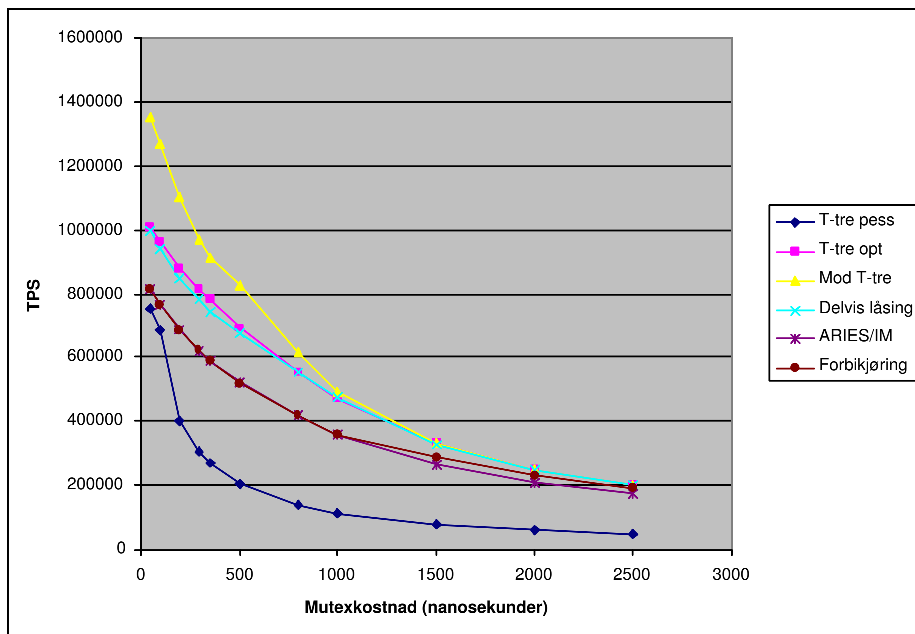
likt, og har noe høyere responstid enn de optimistiske algoritmene for alle mutexkostnader. For de lave mutexkostnadene skyldes dette at B-treet har kostbare binærsøk i hver node, mens for høyere mutexkostnader skyldes det at den setter 3 låser i treet og dermed bruker mutexer 2 ganger mer enn de optimistiske algoritmene. Man ser tydelig effekten av at T-tre med pessimistisk samtidigethskontroll setter låser på hvert nivå i grafen. Responstiden til algoritmen stiger klart raskest, og den blir klart underlegen med høye mutexkostnader.



Figur 5.9: Responstider ved varierende mutexkostnad med 4 prosessorer.

Figur 5.10 viser throughput fra simuleringen for 4 prosessorer. Man ser at for mutexkostnader på opp til 1000 nanosekunder har modifisert T-tre høyest througput. For de laveste mutexkostnadene har den rundt 35% høyere TPS enn de to andre optimistiske metodene. Årsaken til dette er at mutexkostnaden her ikke er den dominerende kostnaden og at man drar bedre nytte av de effektive søkene med modifisert T-tre. Optimistisk samtidighetskontroll har noe høyere TPS enn T-tre med delvis låsing når mutexkostnaden er lav. Årsaken til dette er at optimistisk samtidighetskontroll setter èn lås mindre ved leseoperasjoner. For mutexkostnad på over 1000 nanosekunder har de tre optimistiske algoritmene tilnærmet lik throughput. Årsaken til dette er at mutexen som beskytter hjelpevariablene blir flaskehals, på samme måte som vist i delkapittel 5.1.2. Mutexen blir en flaskehals allerede ved 4 prosessorer fordi når mutexkostnaden blir høyere, skal det færre transaksjoner per sekund til for at mutexen alltid blir opptatt. For mutexkostnad på 1500 nanosekunder har man cirka 330.000 transaksjoner per sekund. Setter man inn i formelen fra delkapittel 5.1.2 får man $T = 2 \cdot 1500 \cdot 10^{-9} \cdot 330.000 = 0.99$.

Altså er mutexen opptatt nesten hele tiden og danner derfor en flaskehals. For B-trærne danner ikke mutexen i rotnoden en like klar flaskehals. Årsaken til dette er at algoritmen bruker så lang tid lenger nede i treet med høyere mutexkostnader at ytelsen ikke er begrenset av kapasiteten til rotnoden. Det samme gjelder for T-tre med pessimistisk samtidighetskontroll som har langt lavere throughput enn de andre algoritmene for alle andre mutexkostnader enn de aller laveste. For de laveste mutexkostnadene ser man at algoritmen ikke har mye dårligere throughput enn B-tre-algoritmene. Årsaken til dette er man ikke taper fullt så mye på å sette mange låser når mutexkostnaden er lav.

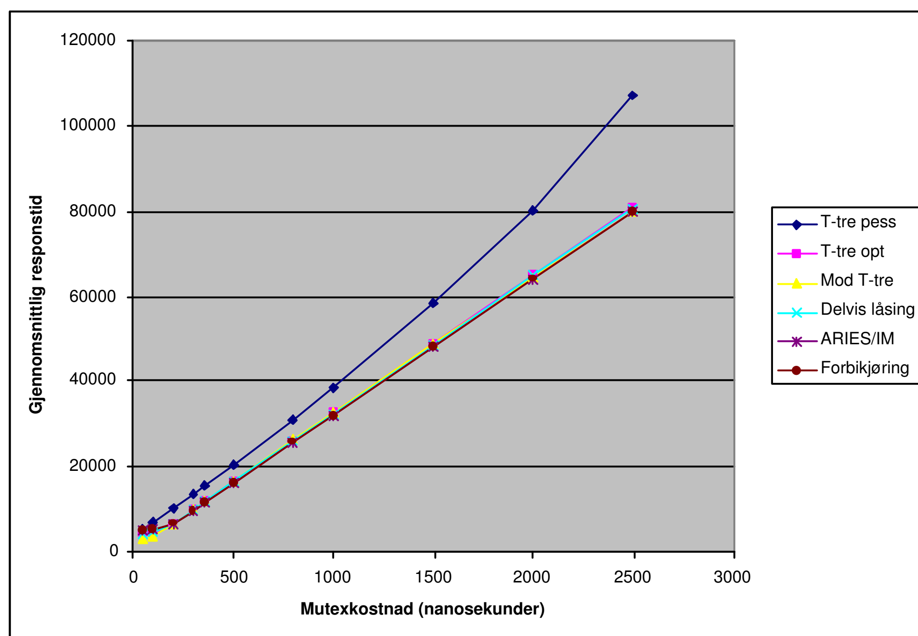


Figur 5.10: Transaksjoner per sekund ved varierende mutexkostnad med 4 prosessorer.

5.2.3 Omgivelse 1: 16 prosessorer

Figur 5.11 viser responstidene til algoritmene med 16 prosessorer. Man ser her at T-tre med pessimistisk samtidighetskontroll også her har høyest responstid, men man ser at den ikke er mye dårligere enn de andre algoritmene. Årsaken til dette er at for de andre algoritmene gjør flaskehalsmutexen seg gjeldene allerede fra relativt lave mutexkostnader. Som vist i delkapittel 5.1.2 får man ved bare lesing med mutexkostnad på 357 nanosekunder flaskehals for de optimistiske algoritmene allerede ved 8 prosessorer, og for B-trærne ved 12 prosessorer. Med 16 prosessorer vil denne flaskehalsen oppstå ved enda lavere mutexkostnader. I tillegg vil køen for å få tilgang til flaskehalsmutexen bli lang på grunn av den høye mutexkostnaden, og dermed vil også responstiden bli meget

høy.

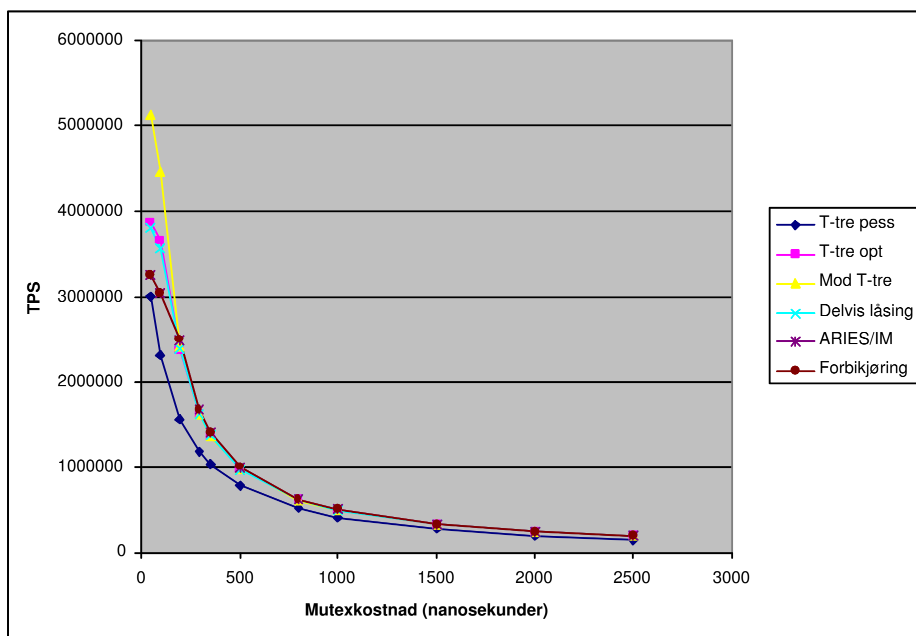


Figur 5.11: Responstider ved varierende mutexkostnad med 16 prosessorer.

Figur 5.12 viser throughput til algoritmene med 16 prosessorer. Man ser at for de aller laveste mutexkostnadene har throughputen for alle algoritmene skalert skalert godt fra 4 til 16 prosessorer, faktisk har TPS nesten blitt firedoblet for samtlige algoritmer med mutexkostnad på 50 nanosekunder. Dette viser igjen hvor stor effekt mutexkostnaden har. Som man så i delkapittel 5.1.4 hvor man varierte prosessorantallet med mutexkostnad på 357 nanosekunder, og hadde samme andel av de ulike transaksjonstypene, stoppet skaleringen de optimistiske algoritmene og B-tre-algoritmene ved 8 prosessorer. Med lavere mutexkostnad skalerer altså algoritmene langt bedre med tanke på antall prosessorer. Man ser også, som ventet, at så snart mutexkostnaden blir noe høyere faller througputen dramatisk. For mutexkostnad på 200 nanosekunder har har mutexen som beskytter hjelpevariablene i de optimistiske algoritmene blitt en flaskehals, og rotnoden har blitt flaskehals for B-trærne.

5.3 Diskusjon

Resultatene fra simuleringene i delkapittel 5.1 viser at de optimistiske algoritmene yter aller best eller tilnærmet like godt som B-trær for alle omgivelser utenom for 100% innsetting. De optimistiske algoritmene har lavest responstid og når høyest mulig throughput



Figur 5.12: Transaksjoner per sekund ved varierende mutexkostnad med 16 prosessorer.

ved færrest prosessorer. Når man har 100% innsetting har de optimistiske algoritmene lavest responstid og høyest throughput opp til 6 prosessorer, men for flere enn 6 prosessorer yter B-trær klart best. Årsaken til at B-tre-algoritmene går forbi de optimistiske algoritmene er at de optimistiske algoritmene låser hele treet ved overflyt og dermed skalerer dårligere. T-tre med pessimistisk samtidighetskontroll yter klart dårligst under nesten alle omgivelser.

Resultatene fra simulering med varierende mutexkostnad i delkapittel 5.2 viser at algoritmene er svært sensitive for mutexkostnaden. Med lave mutexkostnader yter de algoritmene som krever minst bruk av prosessoren best. I tillegg skalerer algoritmene lenger med lave mutexkostnader. Med de laveste mutexkostnadene skalerer alle algoritmene tilnærmet lineært opp til 16 prosessorer. Med høyere mutexkostnad er det antall låser som settes som er avgjørende for ytelsen og flaskehalsen oppstår ved bruk av færre prosessorer.

Alle algoritmene får ved tilstrekkelig belastning èn mutex som virker som en flaskehals som forhindrer videre skalering ved å bruke flere prosessorer. For de optimistiske algoritmene er det mutexen som beskytter hjelpevariablene som er flaskehals, mens for de andre algoritmene er det mutexen som beskytter låsen på rotnoden som er flaskehalsen. Når man har tilstrekkelig mange transaksjoner i systemet er flaskehalsmutexen alltid i bruk, og man kan ikke slippe flere transaksjoner inn. Ved å bruke flere prosessorer oppnår man

ikke høyere throughput, og det høyere antallet transaksjoner i systemet fører til at køen for å få tilgang til flaskehalsmutexen blir lenger. Som en følge av dette blir responstiden høyere. For å unngå denne flaskehalsen som forhindrer videre skalering, foreslås det i kapittel 6 metoder som unngår dette problemet.

Av de optimistiske algoritmene er det modifisert T-tre som yter aller best. Årsaken til dette er at den traverserer treet på en mer effektiv måte. Denne algoritmen ble introdusert i [8], men den bygger på en trestruktur som ikke er like velutprøvd som T-trær eller B-trær. For å vite hvorvidt denne strukturen virkelig er den beste, er det nødvendig å undersøke nærmere hvor ofte innsettinger fører til omorganiseringer. Dersom man ofte får omorganiseringer vil trolig denne algoritmen bli dårligere i omgivelser med innsetting på grunn av at hele treet må låses ofte.

De to B-tre-algoritmene, ARIES/IM og B-tre med forbikjøring, yter tilnærmet like godt under alle omgivelser. Den største forskjellen mellom disse algoritmene er at ARIES/IM i motsetning til B-tre med forbikjøring bruker louck coupling [13] ved traverseringen av treet. Algoritmen holder med andre ord låsene lenger siden den ikke kan slippe låsen på en node før den har låst nodens barnnode. Dette har som vist ingen stor effekt på ytelsen til algoritmen.

Kapittel 6

Algoritmer uten flaskehals

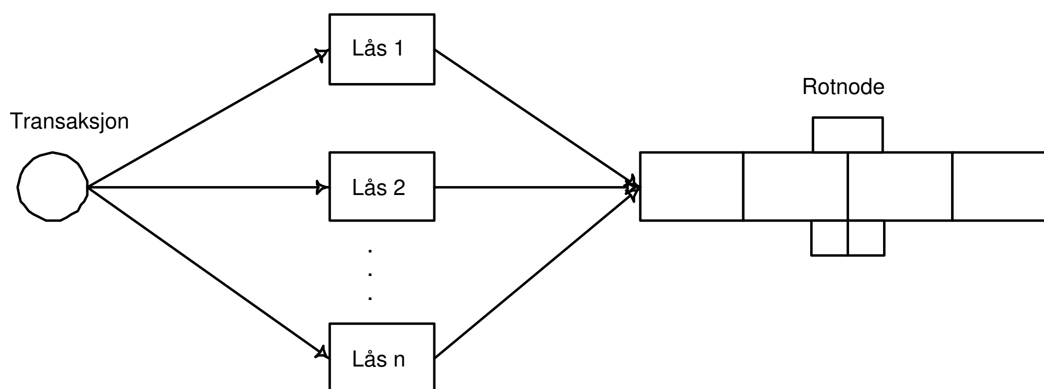
Resultatene fra simuleringene i kapittel 5 viste at throughputen til algoritmene ikke skalerer lenger enn til 8 prosessorer for de optimistiske algoritmene og 12 prosessorer for B-trær. Årsaken til dette er at man har en mutex som fungerer som flaskehals i alle algoritmene. For de optimistiske algoritmene er denne flaskehalsen mutexen som beskytter hjelpevariablene *count* og $fixing$. Denne mutexen bruker hver transaksjon to ganger; èn gang før man starter operasjonen på treet, og èn gang etter man er ferdig i treet. For B-trær er flaskehalsen mutexen som beskytter låsen på rotnoden. Også denne mutexen benyttes to ganger av hver transaksjon; en gang når låsen settes og en gang når den slippes.

For å unngå at algoritmene får en flaskehals som forhindrer videre skalering presenteres det i dette kapitlet løsninger på problemet både for B-trær og for de optimistiske algoritmene. I delkapittel 6.1 og 6.2 presenteres løsninger på dette problemet for B-trær. I delkapittel 6.3 presenteres en løsning på problemet for de optimistiske algoritmene. I delkapittel 6.4 presenteres resultater fra simulering av algoritmer som anvender løsningene på flaskehalsproblemet. Løsningene implementeres på B-tre med forbikjøring og T-tre med optimistisk samtidighetskontroll med delvis låsing. Ytelsen til de nye algoritmene sammenliknes med de opprinnelige algoritmene. Til slutt diskuteres og oppsummeres resultatene i delkapittel 6.5.

6.1 B-trær: Flere låser

For å unngå at mutexen som beskytter låsen til rotnoden i B-treet blir flaskehals, foreslås det her, som vist i figur 6.1, at man benytter seg av flere låser på rotnoden. Transaksjoner som skal sette S-lås på rotnoden trenger bare å låse èn av låsene, mens transaksjoner

som skal sette X-lås må låse *alle* låsene. Rotnoden er ikke låst med X-lås før alle låsene låst med X-lås. For å unngå vranglåser må alle transaksjoner som ønsker å sette X-lås starte med å låse *Lås 1* og låse de andre låsene i stigende rekkefølge. Transaksjoner som ønsker å låse noden med S-lås, velger en tilfeldig lås, og den kan få S-lås på rotnoden selv om noen av de andre låsene er låst med X-lås.

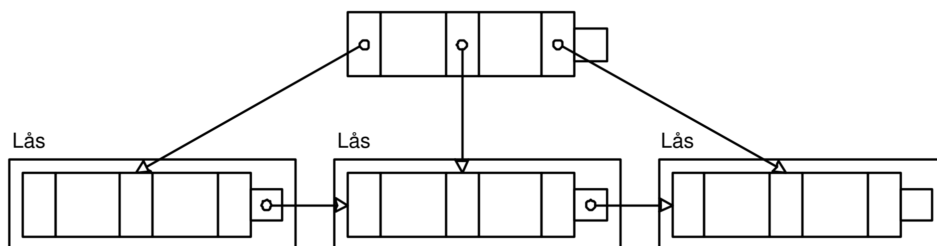


Figur 6.1: Node med flere låser.

Denne låsestrategien vil føre til at man ikke lenger har èn mutex som fungerer som flaskehals. Med n låser vil rotnoden i teorien kunne tillate en throughput som er n ganger høyere enn med bare èn lås. Transaksjoner som setter X-lås vil få høyere responstid enn ved vanlige B-trær, da de må sette flere låser på rotnoden. Dette er kostbart da mutexkostnaden er høy, og transaksjonene må vente på at låsene blir ledig. Man kan derfor forvente at denne låsestrategien vil fungere best i omgivelser hvor det er sjelden at man trenger å låse rotnoden med X-lås. Antallet låser på rotnoden bør velges ut fra en avveining av hvor mange prosessorer systemet har, mutexkostnaden og hvor hyppig man forventer at rotnoden vil bli låst med X-lås.

6.2 B-trær: Statisk rotnode

I omgivelser hvor nøkkelverdiene i indeksen ligger innenfor et kjent intervall, og fordelingen av nøklene i intervallet er kjent, vil nøklene og pekerne i rotnoden kunne velges på en måte som gjør at behovet for oppdatering av rotnoden er svært lite. I slike omgivelser kan man ha en statisk rotnode som på forhånd partisjonerer nøklene i subtrær. Figur 6.2 viser et B-link-tre med statisk rotnode. Som man ser trenger man ikke å beskytte rotnoden med lås da den aldri vil endres. Ved traversering av treet starter transaksjonene å låse nodene fra rotnodens barnnoder og nedover i treet. På denne måten vil flaskehalsen i rotnoden elimineres, og man vil kunne få høyere gjennomstrømning i treet.



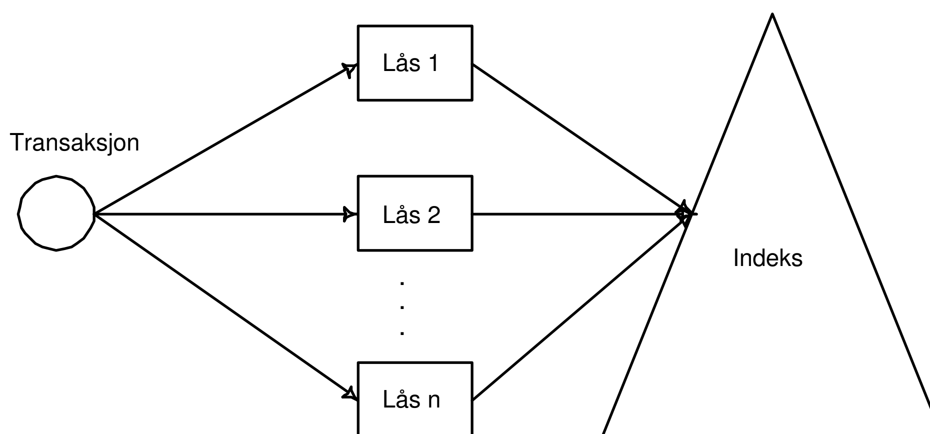
Figur 6.2: B-link-tre med statisk rotnode.

Denne metoden forutsetter altså at man kjenner til fordelingen av nøkkelverdiene. Dersom man velger denne metoden og nøkkelkarakteristikkene endrer seg slik at ett subtre får høyere belastning enn de andre subtrærne, vil den aktuelle barnnoden til rotnoden kunne bli flaskehals. Årsaken til dette er at man får et stort antall transaksjoner som skal bruke denne barnnoden, og dermed vil belastningen på låsen blir høy. Man kan altså få det samme problemet for barnnoder som man får for rotnoden i de opprinnelige B-trealgoritmene.

6.3 Optimistisk: Flere låser

For de optimistiske algoritmene er det mutexen som beskytter hjelpevariablene *count* og $fixing$ som er flaskehalsen som forhindrer videre skalering. Hensikten til disse to hjelpevariablene er egentlig den samme som til en vanlig nodelås; man kan ha et ubegrenset antall vanlige transaksjoner tilstede i treet, men transaksjoner som gjør endringer på trestrukturen må være alene.

I delkapittel 6.1 ble det presentert en løsning på flaskehalsproblemet for rotnoden i et B-tre. Den samme løsningen kan, som vist i figur 6.3 anvendes for de optimistiske algoritmene. Det vil si at man erstatter hjelpevariablene med et antall låser som beskytter treet; lesetransaksjoner setter S-lås på en tilfeldig valgt lås, mens transaksjoner som skal gjøre omorganiseringer låser alle låsene med X-lås. Med en slik låsestrategi oppnår man samme effekt som med hjelpevariablene *count* og $fixing$. Da transaksjoner som skal gjøre omorganiseringer må låse alle låsene, er det, som for B-tre med flere låser, nærliggende å forvente at denne løsningen vil fungere best i omgivelser hvor man sjelden trenger å gjøre omorganiseringer.



Figur 6.3: Indeks med flere låser.

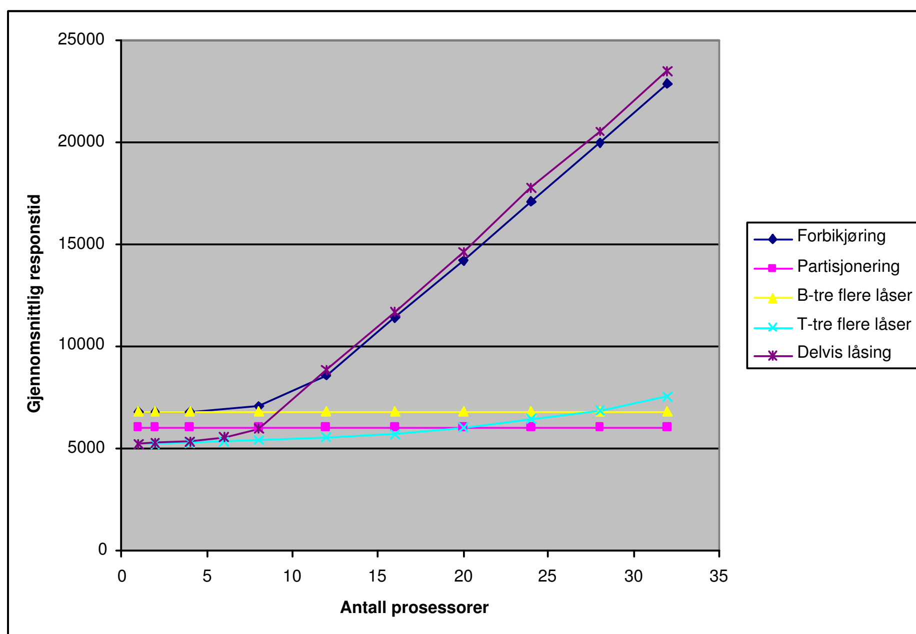
6.4 Simuleringsresultater

For å vurdere løsningene på flaskehalsproblemet presentert i delkapittel 6.2, 6.1 og 6.3 presenteres her resultater fra simulering av algoritmer som implementerer løsningene. Løsningene for B-trær anvendes på B-tre med forbikjøring og løsningen for de optimistiske algoritmene anvendes på T-tre med delvis låsing. For å se effekten av løsningene sammenliknes algoritmene med orginalalgoritmene. Algoritmene testes med varierende prosessorantall i de samme omgivelsene som ble brukt i delkapittel 5.1.4, det vil si 50% søking, 40% oppdatering, 5% innsetting og 5% sletting. Det antas at forutsetningene til omgivelsene til algoritmen med statisk rotnode er oppfyllt. Det anvendes 4 låser for rotnoden i B-treet og 4 trelåser for T-treet.

Som beskrevet i delkapittel 4.3.2 gjør B-tre-algoritmene i simulatoren aldri endringer høyere opp i treet enn ett nivå over løvnodene etter nodesplitt. Da man i simuleringen har 3 nivåer i B-treet vil man aldri få behov for å X-låse rotnoden. Dette fører til at for algoritmen med flere låser for B-trær vil man ikke få simulert effekten av transaksjoner som ønsker å låse rotnoden med X-lås. Dette vil trolig ha liten effekt på denne simuleringsomgivelsen da man kun utfører 5% innsetting, og endringer i rotnoden på et stort tre forekommer relativt sjeldent.

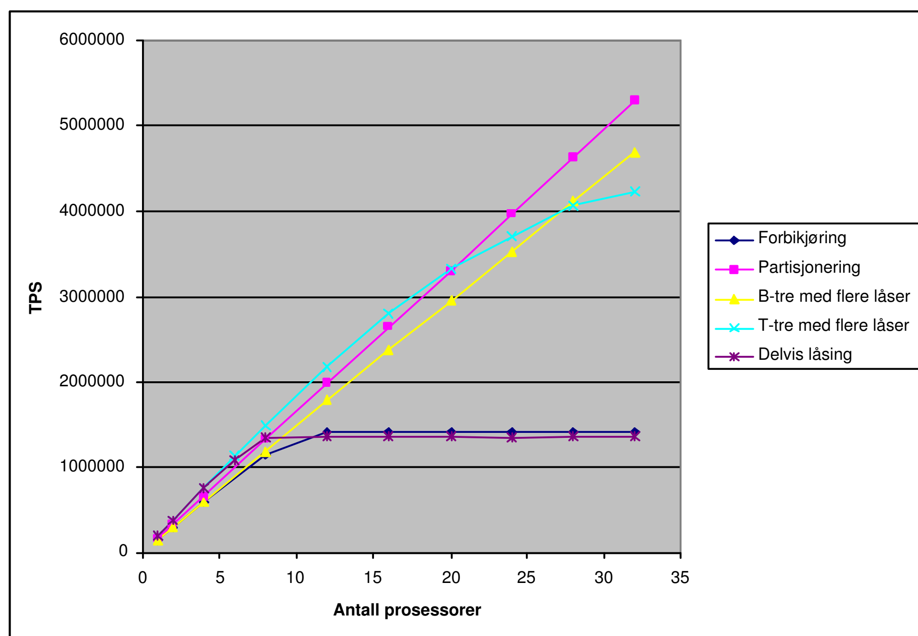
Figur 6.4 viser responstidene til algoritmene. Man ser at de to nye algoritmene for B-trær beholder den lave responstiden helt opp til 32 prosessorer. Årsaken til dette er at man har eliminert flaskehalsen i rotnoden. Algoritmen med statisk rotnode har noe lavere responstid enn algoritmen med flere låser siden den setter èn mindre lås ved traversering av treet. Vanlig B-tre med forbikjøring har tilnærmet lik responstid som algoritmen med flere låser opp til 8 prosessorer. For flere enn 8 prosessorer ser man at responstiden stiger.

Årsaken til dette er at mutexen som beskytter låsen på rotnoden har blitt flaskehals. Man ser at T-tre med flere låser også skalerer langt bedre enn orginalalgoritmen, delvis låsing. Opptil cirka 16 prosessorer beholder T-tre med flere låser den lave responstiden og har lavere responstid enn de to nye B-tre-algoritmene. For fler enn 16 prosessorer stiger responstiden noe, og for 28 og flere prosessorer yter den dårligere enn begge de nye B-tre-algoritmene. Årsaken til dette er trolig at man ved omorganiseringer får lang ventetid på trelåsene. Med dette menes at etter at en transaksjon har gjort omorganiseringer vil det være lange køer for å få tilgang til trelåsene, og transaksjonene får derfor høyere responstid.



Figur 6.4: Responstider ved 50% søking, 40% oppdatering, 5% innsetting og 5% sletting.

Figur 6.4 viser throughput til algoritmene. Man ser her den samme trenden som for responstider. Det vil si at throughputen til de to nye algoritmene for B-tre skalerer helt opp til 32 prosessorer, mens throughput flater ut ved rundt 8 prosessorer for vanlig B-tre med forbikjøring. Algoritmen med statisk rotnode har noe høyere throughput. Årsaken til dette er også her at man sparer tid på å ikke sette lås på rotnoden og dermed klarer å slippe flere transaksjoner gjennom systemet. T-tre med flere låser oppnår langt høyere throughput enn den orginale algoritmen. Likevel flater TPS til T-tre med flere låser noe ut ved cirka 16 prosessorer. Årsaken til dette er trolig også her at man på grunn av mange transaksjoner i systemet får lang ventetid ved trelåsene etter omorganiseringer. Som en følge av dette slipper færre transaksjoner gjennom systemet.



Figur 6.5: Throughput ved 50% søking, 40% oppdatering, 5% innsetting og 5% sletting.

6.5 Diskusjon

I kapittel 5 ble det vist at samtlige av algoritmene som ble simulert hadde en flaskehals som forhindret videre skalering med tanke på flere prosessorer. I dette kapitlet har det blitt presentert to løsninger på dette problemet for B-tre og èn løsning for de optimistiske algoritmene. Ved simulering ble det vist at algoritmene som anvender løsningene for B-tre skalerer lineært helt opp til 32 prosessorer. Løsningen for de optimistiske algoritmene med 4 trelåser skalerer langt bedre enn orginalalgoritmene, men skaleringen flatet noe ut for flere enn 16 prosessorer. Med andre ord kan man ved hjelp av enkle virkemidler få algoritmene som ble simulert i kapittel 5 til å skalere langt bedre. Alle de presenterte løsningene har begrensninger; med flere låser på rotnoden eller flere trelåser vil man i omgivelser hvor man ofte må låse låsene med X-låser trolig få markert dårligere ytelse. Med statisk rotnode er man avhengig av at nøkkelverdiene ligger innenfor et kjent intervall.

Metodene som er presentert i delkapittel 6.1, 6.2 og 6.3 er anvendt på B-tre med forbikjøring og T-tre med delvis låsing, men de er også anvendbare på de andre aksessmetodene. Med andre ord finnes det metoder for å oppnå bedre skalering på samtlige av algoritmene som er simulert i kapittel 5.

Kapittel 7

Konklusjon

Denne rapporten har presentert resultater fra simuleringer av ulike aksessmetoder som støtter samtidige transaksjoner for MMDB. Simuleringene har tatt i bruk resultater fra nyere microbenchmarker med lavere låsekostnader. Algoritmene har blitt simulert under omgivelser med ulike operasjonstyper og med varierende prosessorantall. Resultatene viser at samtidighetsalgoritmene med optimistisk låsestrategi, altså T-tre med optimistisk samtidighetskontroll, T-tre med delvis låsing og modifisert T-tre, yter bedre enn algoritmene for B-trær, ARIES/IM og B-tre med forbikjøring. De eneste omgivelsene hvor B-trealgoritmene yter bedre eller tilnærmet like godt som de optimistiske algoritmene er hvor det er en stor andel innsettingsoperasjoner. I slike omgivelser yter B-trær bedre enn de optimistiske algoritmene for 8 eller flere prosessorer. T-tre med pessimistisk samtidighetskontroll, som setter låser på alle nivå i treet, yter klart dårligst under nesten alle omgivelser.

Resultatene som er presentert i denne rapporten er tildels motstridende med resultatene presentert av Lu, Ng og Tian i [7]. Disse resultatene viste at B-trær hadde høyere throughput enn T-trær for alle omgivelser med èn prosessor. Simuleringene i denne rapporten viser at for èn prosessor og med de samme omgivelsene som ble brukt i [7], har T-tre med optimistisk samtidighetskontroll rundt 30% høyere throughput enn B-tre med forbikjøring. Likevel er det vanskelig å gjøre noen direkte sammenlikning med disse resultatene da simuleringene i [7], som vist i delkapittel 3.2, er noe mangelfullt dokumentert. I tillegg presenteres resultatene i [7] på en måte som ikke viser transaksjonenes responstider.

Modifisert T-tre er algoritmen som jevnt over har den beste ytelsen. For de aller fleste omgivelsene har den noe lavere responstid og høyere throughput enn de andre optimistiske algoritmene. Selv om denne algoritmen yter best i simuleringene er det knyttet noe usikkerhet til hvor godt algoritmen vil yte i omgivelser hvor man har en dynamisk indeks. Årsaken til dette er at algoritmen bruker en indeks som er noe annerledes enn

T-trær, og er derfor ikke like gjennomtestet som T-trær og B-trær.

For å avdekke hvor sensitive de ulike algoritmene er for kostnaden til synkroniseringsprimitivene, har det også blitt presentert resultater fra simuleringer med varierende mutexkostnad. Resultatene viser at for lave mutexkostnader yter algoritmene som er effektive med tanke på prosessorbruk best, mens for høyere mutexkostnader har effektiv prosessorbruk mindre å si. I omgivelser med høye mutexkostnader er det antall låser som settes i treet som avgjør ytelsen til algoritmene. Årsaken til dette er at mutexkostnaden blir en dominerende kostnadsfaktor.

Ved bruk av tilstrekkelig mange prosessorer får alle algoritmene èn mutex som virker som en flaskehals som forhindrer videre skalering. Ved bruk av flere prosessorer er det mulig å ha flere samtidige transaksjoner i aksessmetodene. På grunn av den økte belastningen fra flere transaksjoner er mutexen nesten i konstant bruk og indeksen har derfor ikke mulighet for høyere throughput. I kapittel 6 ble det presentert løsninger på dette problemet både for B-trær og for de optimistiske algoritmene for T-trær. Det ble presentert resultater fra simuleringer av algoritmer som tar løsningene i bruk. Resultatene viser at flaskehalsen elimineres og at man kan oppnå tilnærmet lineær skalering av throughput og tilnærmet konstant responstid opp til minst 32 prosessorer ved bruk av de nye algoritmene. Alle løsningene har begrensninger som gjør at de ikke vil yte optimalt under alle omgivelser.

Kapittel 8

Videre forskning

I denne rapporten, hvor det ikke har blitt tatt hensyn til hurtigminne, har det blitt vist at T-trær jevnt over yter bedre enn B-trær ved bruk av de optimistiske algoritmene for samtidighetskontroll. Resultater fra tidligere studier som tar hensyn til hurtigminne, men utelater samtidighetskontroll, viser at T-trær yter dårligere enn B-trær. T-trær er mindre hurtigminneoppmerksomme enn B-trær, men på grunn av den mer effektive låsingen er det uavklart om de vil yte bedre eller dårligere enn B-trær når også samtidighetskontroll tas med. For å få et fullstendig bilde av ytelsen til algoritmene vil det være nødvendig å utføre simuleringer som tar hensyn til både samtidighetskontroll og hurtigminne. Vi anbefaler dette som det neste steget innen forskningen på dette feltet.

Resultatene som er presentert i denne rapporten bygger på en simuleringsmodell som gjør enkelte forenklinger fra de virkelige aksessmetodene. Selv om resultatene trolig stemmer overens med det man ville fått i en virkelig implementasjon, vil det være nødvendig å implementere aksessmetodene i et virkelig system for å kunne konkludere med hvilken som er best. En slik implementasjon trenger trolig bare å fokusere på de algoritmene som kommer best ut i resultatene presentert i denne rapporten.

Simuleringene bygget på microbenchmarks utført på en datamaskin med noe utdatert maskinvare. Selv om den relative kostnaden til de ulike operasjonene trolig er gyldig også for nyere maskinvare, vil man få enda mer representative resultater ved bruk av microbenchmarker utført på en nyere datamaskin. Det mest optimale er at datamaskinen har like spesifikasjoner som systemet hvor aksessmetodene virkelig skal anvendes. I tillegg ble kostnaden for bruk av mutexer funnet ved microbenchmarking på en ènprosessormaskin. Kostnaden for bruk av mutexer kan være høyere på systemer med flere prosessorer. Derfor kan resultatene bli mer riktige ved bruk av kostnad funnet ved microbenchmarking på en maskin med flere prosessorer.

Modifisert T-tre, som jevnt over hadde best ytelse i simuleringene, er en ny aksessmetode som ikke er like velutprøvd som T-trær og B-trær. For å vite hvor godt aksessmetoden vil yte i et virkelig system, er det nødvendig å gjøre grundigere undersøkelser av aksessmetodens egenskaper. Dersom bruk av aksessmetoden for eksempel vil føre til hyppige omorganiseringer av treet ved innsetting vil ytelsen kunne bli langt dårligere.

Bibliografi

[1] Phil Bernstein, Michael Brodie, Stefano Ceri, David DeWitt, Mike Franklin, Hector Garcia-Molina, Jim Gray, Jerry Held, Joe Hellerstein, H. V. Jagadish, Michael Lesk, Dave Maier, Jeff Naughton, Hamid Pirahesh, Mike Stonebraker, and Jeff Ullman. The Asilomar report on database research. *SIGMOD Rec.*, 27(4):74–80, 1998.

[2] H. Garcia-Molina and K. Salem. Main Memory Database Systems: An Overview. *IEEE Transactions on Knowledge and Data Engineering*, 4(6):509–516, 1992.

[3] Tobin J. Lehman and Michael J. Carey. A Study of Index Structures for Main Memory Database Management Systems. In *Proceedings of the Twelfth International Conference on Very Large Data Bases*, pages 294–303. Morgan Kaufmann Publishers Inc., 1986.

[4] Times Ten. Times Ten Architectural Overview. Technical report, 2003.

[5] Er Vi Ew. DataBlitz Architectural Overview.

[6] 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.

[7] Hongjun Lu, Yuet Yeung Ng, and Zengping Tian. T-Tree or B-Tree: Main Memory Database Index Structure Revisited. In *Proceedings of the Australasian Database Conference*, page 65. IEEE Computer Society, 2000.

[8] Arne Eirik Nielsen. Effekten av samtidighetskontroll i aksessmetoder for main memory databaser. Technical report, Norges teknisk-naturvitenskapelige universitet, 2004.

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

[10] Vibby Gottemukkala and Tobin J. Lehman. Locking and Latching in a Memory-Resident Database System. In *Proceedings of the 18th International Conference on Very Large Data Bases*, pages 533–544. Morgan Kaufmann Publishers Inc., 1992.

[11] Jim Gray. Notes on Data Base Operating Systems. In *Operating Systems, An Advanced Course*, pages 393–481. Springer-Verlag, 1978.

[12] J.-L. Baer and B. Schwab. A comparison of tree-balancing algorithms. *Commun. ACM*, 20(5):322–330, 1977.

[13] Theodore Johnson and Dennis Sasha. The performance of current B-tree algorithms. *ACM Trans. Database Syst.*, 18(1):51–101, 1993.

[14] Douglas Comer. Ubiquitous B-tree. *ACM Comput. Surv.*, 11(2):121–137, 1979.

[15] 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, 1992.

[16] Philip L. Lehman and s. Bing Yao. Efficient locking for concurrent operations on B-trees. *ACM Trans. Database Syst.*, 6(4):650–670, 1981.

[17] C. Mohan and Frank Levine. ARIES/IM: an efficient and high concurrency index management method using write-ahead logging. In *Proceedings of the 1992 ACM SIGMOD international conference on Management of data*, pages 371–380. ACM Press, 1992.

[18] Yehoshua Sagiv. Concurrent operations on B-trees with overtaking. In *Proceedings of the fourth ACM SIGACT-SIGMOD symposium on Principles of database systems*, pages 28–37. ACM Press, 1985.

[19] Anastasia Analyti and Sakti Pramanik. Fast search in main memory databases. In *SIGMOD '92: Proceedings of the 1992 ACM SIGMOD international conference on Management of data*, pages 215–224, New York, NY, USA, 1992. ACM Press.

[20] V. Kumar. A concurrency control mechanism based on extendible hashing for main memory database systems. In *Proceedings of the seventeenth annual ACM conference on Computer science : Computing trends in the 1990's*, pages 109–113. ACM Press, 1989.

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

[22] Sang K. Cha, Sangyong Hwang, Kihong Kim, and Keunjoo Kwon. Cache-Conscious Concurrency Control of Main-Memory Indexes on Shared-Memory Multiprocessor Systems. In *VLDB '01: Proceedings of the 27th International Conference on Very*

Large Data Bases, pages 181–190, San Francisco, CA, USA, 2001. Morgan Kaufmann Publishers Inc.

[23] Derek Chiou, Prabhat Jain, Larry Rudolph, and Srinivas Devadas. Application-specific memory management for embedded systems using software-controlled caches. In *DAC '00: Proceedings of the 37th conference on Design automation*, pages 416–419, New York, NY, USA, 2000. ACM Press.

[24] John S. Carson. Introduction to simulation: introduction to modeling and simulation. In *WSC '03: Proceedings of the 35th conference on Winter simulation*, pages 7–13. Winter Simulation Conference, 2003.

[25] Jerry Banks and II John S. Carson. Introduction to discrete-event simulation. In *WSC '86: Proceedings of the 18th conference on Winter simulation*, pages 17–23, New York, NY, USA, 1986. ACM Press.

[26] Jerry Banks. Simulation fundamentals: simulation fundamentals. In *WSC '00: Proceedings of the 32nd conference on Winter simulation*, pages 9–16, San Diego, CA, USA, 2000. Society for Computer Simulation International.

[27] Jaroslav Kacer. Discrete event simulations with j-sim. In *PPPJ '02/IRE '02: Proceedings of the inaugural conference on the Principles and Practice of programming, 2002 and Proceedings of the second workshop on Intermediate representation engineering for virtual machines, 2002*, pages 13–18, Maynooth, County Kildare, Ireland, Ireland, 2002. National University of Ireland.

[28] Ole-Johan Dahl and Kristen Nygaard. SIMULA: an ALGOL-based simulation language. *Commun. ACM*, 9(9):671–678, 1966.

[29] Edsger W. Dijkstra. The structure of the "THE"-multiprogramming system. *Commun. ACM*, 11(5):341–346, 1968.

[30] Svein-Olaf Hvasshovd, Øystein Torbjørnsen, Svein Erik Bratsberg, and Per Holager. The ClustRa Telecom Database: High Availability, High Throughput, and Real-Time Response. In *VLDB '95: Proceedings of the 21th International Conference on Very Large Data Bases*, pages 469–477, San Francisco, CA, USA, 1995. Morgan Kaufmann Publishers Inc.

[31] Surajit Chaudhuri and Umeshwar Dayal. An overview of data warehousing and OLAP technology. *SIGMOD Rec.*, 26(1):65–74, 1997.

[32] Joonwon Lee and Umakishore Ramachandran. Synchronization with multiprocessor caches. In *ISCA '90: Proceedings of the 17th annual international symposium on Computer Architecture*, pages 27–37, New York, NY, USA, 1990. ACM Press.