Minuet: Egy Skálázható Elosztott MultiverziósB-Fa

Tanulmány Benjamin Sowell, WojciechGolab és MehulA. Shah:
„Minuet:A ScalableDistributedMultiversionB-Tree" című cikke alapján [1].

Hodosy Gábor, Fokin Miklós, Tóth Tamás
Adatbázisrendszerek elméleti alapjai, ELTE IK 2012

ABSZTRAKT

Az adat kezelő rendszereket általában úgy tervezik, hogy vagy hosszú távú elemző lekérdezéseket, vagy rövid életű tranzakciókat támogassanak. Manapság viszont egyre több alkalmazásnak van szüksége, arra hogy képes legyen mindkettő hatékony végrehajtására. A cikkben az írók bevezetik a Minuet-etegy olyan elosztott, központi memóriát használó B-fát, ami támogat tranzakciókat is és íráskor másolódó (copy-on-write) pillanatképeket is futás közbeni (in-situ) elemzéshez. A Minuet központi memóriát használ, ami elősegíti a tranzakció műveletek alacsony késleltetési idővel történő végrehajtását és elemző lekérdezések futtatását is anélkül, hogy befolyásolnák a tranzakciók teljesítményét. A Minuet továbbá lehetővé teszi írható klónok segítségével többféle,szerteágazó verzió létrehozását az adatokról. A témában végzett tesztelések azt mutatják, hogy a Minuet több ponton is jobban teljesít az eddigi központi memóriát használó adatbázisoknál. Skálázhatósága több száz magig terjed és több terra bájtnyi memóriáig, valamint képes feldolgozni több százezer B-fa műveletet másodpercenként miközben hosszú futási idejű vizsgálatokat végez.

1. BEVEZETÉS

A modern alkalmazások egyre nagyobb hangsúlyt fektetnek az adatbevitel és kiértékelés között eltelt idő csökkentésére. Nem csak a műveletek gyors végrehajtására van szükség, de a keletkezett adatokat gyorsan elemezni is kell, hogy ezáltal ajánlásokat és becsléseket tudjanak nyújtani, amik javítják a felhasználói élményt. Jó példát kínálnak az ilyen esetekre az online kereskedelmi oldalak, amik nem csak nyomon követik és elvégzik a felhasználó vásárlásait, hanem az így keletkezett információk alapján javaslatokat is tesznek további termékek vásárlására. Egy másik példával szolgálnak az online multiplayer játékok, ahol a rendszer nyomon követheti a játékosok pozícióját és cselekedeteit, majd ezek alapján virtuális javakat kínál fel nekik megvásárlásra. Hitelkártya társaságoknak ki kell szűrniük a csalásokat, hirdető cégeknek pedig fel kell mérniük az aktuális trendeket a reklámok célzott kivitelezéséhez. Az előbbiek alapján láthatjuk, hogy a mai modern üzletek számára az adatok gyors kiértékelése és optimalizálása már elengedhetetlenné vált.

A legtöbb kivitelezésben a legfejlettebb módszer, hogy két teljesen különböző rendszert alkalmaznak a működési állapot és az elemzések kezelésére. Tranzakciós rendszereket és újabban kulcs-érték tárakat használnak a rövid életű műveletek végrehajtásához, a hosszú ideig futó elemzésekhez pedig adat tárházakat. Ezek az elkülönítések a munka igények különbözőségébőladódnak és általában több órányi vagy akár napnyi késleltetést vonhatnakmaguk után.

A két rendszer közötti rés megszűntetésére vezetik be az írók a Minuet-et, melynek kiemelkedő vonásait a következő három pontban adják meg:

1. **Teljesítmény és skálázhatóság:** Több százezer tranzakció végrehajtása 1ms körüli késleltetéssel. A cikkben megmutatják, hogy a teljesítmény szinte lineárisan skálázható több száz maghoz és több terra bájtnyi memóriához.
2. **Íráskor másolódó pillanatképek**: Konzisztens pillanatképek biztosítása a futás közbeni elemzéshez, anélkül hogy rontaná a teljesítményét a folyamatban lévő tranzakcióknak. Csökkenti a felesleges helyfoglalást a pillanatképek létrehozásának szabályozásával és a konkurens lekérdezések közti pillanatkép megosztással.
3. **Írható klónok**: Maguk a pillanatképek írhatók, így szerteágazó verziók hozhatóak létre az adatokból. Ez a funkció sok helyen hasznos lehet, például „mi történik, ha" típusú elemzések és előrejelzések kiértékelésekor, vagy széles körű rendszer másoláskor, megosztáskor, archiváláskor.

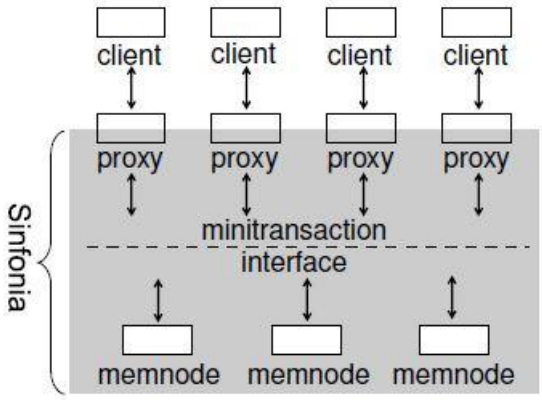
A Minuet ezeket a funkciókat mind biztosítani tudja azáltal, hogy minden adatot teljes mértékben a memóriában tárol. A lemezekkel ellentétben a memória egyidejűleg tud biztosítani alacsony késleltetési idővel rendelkező véletlenszerű elérést és gyors szekvenciális elérést is. Ez a megközelítés a mai nagyméretű központi memóriák mellett egy praktikus megoldássá válik.

A Minuet alapjául egy korábbi cikkben leírt [2] skálázható elosztott B-fa megvalósítás szolgál, ami pedig a Sinfonia adat megosztó szolgáltatás [3] segítségével került megvalósításra (ugyanezt használja a Minuet is). Ezek bemutatása a következő fejezetben történik. A Minuet teljes megvalósítása az előbb említett módszerek kiegészítésével történik. Egy új konkurencia kezelés bevezetésére kerül sor, valamint a másoláskor íródó pillanatképek bevezetésével biztosításra kerül a támogatás a hosszú vizsgálatú műveletekhez.

2. KAPCSOLÓDÓ MUNKÁK

A Minuet egy skálázható elosztott multiverziósB-fába szervezi az adatokat és elérhetővé tesz egy kulcs-érték párokon alapuló interfészt, amin keresztül támogatást biztosít tartományon végezhető lekérdezések végzéséhez. A rendszer struktúrája három fő elemből épül fel, ezek:

1. **Kliensek**: A kliensektől származnak a kérések a B-fa utasítások végrehajtására.
2. **Proxy-k**: Ezek hajtják végre a B-fa utasításokat a kliensek nevében.
3. **Memnode-ok**: Ezeken tárolódnak a B-fa állapotok, amiket a Proxy-klekérdeznek és a műveleteket végzik rajtuk.



1. ábra

A rendszer magjában, ahogy az *1.ábrán* látható, a Sinfonia [3] szolgáltatás áll, ami a memória és a proxy-k közötti kommunikációt valósítja meg.

Sinfonia

Sinfonia egy adat megosztó szolgáltatás, ami tároló helyek egy halmazából (memnode-ok) és egy a műveletek végrehajtásáért felelős alkalmazás könyvtárból áll. Minden memnode exportál egy struktúrálatlan bájt-címzésű tároló helyet és az alkalmazás könyvtár ezekhez a helyekhez kínál hibatűrő tranzakciós hozzáférést.

Sinfonia működése mini-tranzakciókon keresztül valósul meg. Egy mini-tranzakció olvashat, összehasonlíthat vagy feltételesen frissíthet valamilyen adatot, ami különböző memória helyeken helyezkedhet el, valószínűleg különböző memnode-okon, amik különböző szervereken futnak. Az alkalmazás előre meghatározza az érintett memória címeket és a frissítés csak akkor lép érvénybe, ha minden kiértékelés pozitív eredménnyel tért vissza (vagy nem is volt kiértékelés). A Minuetben ez azt jelenti, hogy egy proxy használhat egy mini-tranzakciót, hogy olvasson egy B-fa csúcsot egy memnode-ból, vagy frissíthet egy vagy több B-fa csúcsot módosításkor vagy beszúráskor.

A Sinfonia egy két fázisú protokollt használ a mini-tranzakciók végrehajtásához és érvényesítéséhez. Ennek a lépései a következőek:

1. Először zárolja az érintett memória területeket és kiértékeli az összehasonlításokat.
    a. Amennyiben egy memória cím zárolva van: megszakítja a tranzakciót és automatikusan újra próbálkozik.
    b. Egy kiértékelés megbukik: megszakítja a tranzakciót és visszaadja az alkalmazásnak, hogy melyik feltétel nem teljesült.
2. Ha minden feltétel teljesült, végrehajtásra kerül a mini-tranzakció.

Dinamikus Tranzakció Réteg

Mivel az érintett memória helyeket előre meg kell határozni a mini-tranzakciók előbb leírt végrehajtásához, nem lehet egy mini-tranzakcióval bejárni egy B-fát. A Minuet alapjául szolgáló B-fa leírásánál [2], megadnak egy módszert, ami segítségével mini-tranzakciók felhasználásával dinamikus tranzakciók konstruálhatóak, amik tetszőlegesen írhatnak és olvashatnak objektumokat. Ezek a dinamikus tranzakciók optimista konkurencia vezérlést fognak használni és validálásuk visszafelé irányuló ellenőrzéssel történik.

Minden dinamikus tranzakció karbantart egy olvasási és egy írási halmazt. Ezek felhasználásával a következő módon történik a végrehajtás:

1. Olvasásnál először lokálisan keresi az adatot az olvasási vagy írási halmazban
    a. Amennyiben nem találta lokális halmazokban: elindít egy mini-tranzakciót, ami bekéri egy memnode-ból és hozzáadja a saját olvasási halmazához.
2. Írásnál a lokális írási halmazba helyezi az objektumot és elhalasztja a memnode-ok frissítését, amíg nem kerül végrehajtásra a teljes dinamikus tranzakció.
3. A végrehajtás maga után von egy mini-tranzakciót, ami:
    a. Érvényesíti az olvasási halmazt.
    b. Amennyiben az érvényesítés sikeres volt, bemásolja az írási halmaz elemeit a memnode-okba.

Az olvasási halmaz ily módon történő teljes ellenőrzése biztosítja, hogy a dinamikus tranzakciók sorba rendezhetőek. A processzor és hálózati működés optimalizálása érdekében bevezethetőek sorszámok az objektumokon, amik folyamatosan növekednek a frissítéskor, így az összehasonlítások ezekre a sorszámokra egyszerűsödhetnek. Továbbá megjegyzendő, hogy az olvasási halmaz validálása elhagyható, amikor az írási halmaz üres.

A dinamikus tranzakciók felhasználhatóak bármilyen centralizált adat struktúra implementációjának transzformálására egy olyanra, ami elosztott több különböző memnode között és elérhető sok kliens és proxy által.

Elosztott B-fa

A Minuet alapjául szolgáló, fentebb említett elosztott B-fa(fontos megjegyezni, hogy bár az eredeti cikk nem említi, de a hivatkozott [2] írás igen, itt valójában B+-fákról van szó annak ellenére, hogy végig csak B-faként hivatkoznak rájuk a szerzők) megvalósítása is dinamikus tranzakciókkal történik [2]. Az implementációban a fa az által válik elosztottá, hogy a csúcsait különböző szervereken tárolja. Ezt a tárolást egy elosztott memória kiosztás vezérlő végzi, kiegyenlített módon.

Az alapvető B-fa műveleteket úgy kapjuk, hogy dinamikus tranzakciókat körítünk az egy szálú kód köré. Ez azt is eredményezi, hogy a tranzakcionális írás/olvasás után biztonsági ellenőrzéseket végzünk, hogy megbizonyosodjunk róla, hogy a kód nem omlik össze vagy akad el, amikor egy tranzakció inkonzisztens adattal tér vissza. Ezen kívül a fő implementációs feladat a teljesítmény javítása, amit egyfelől a hálózaton bejárt útvonal csökkentésével és a versengés elkerülésével érhetünk el.

Két fontos optimalizálást említ a cikk. Elsőként a kommunikációs költség csökkentéséhez, eltárolja a belső (nem levél) B-fa csúcsokat a proxykban. Ezek az adatok a proxy-k belső alkalmazás kódjának részei és nem biztosított a koherencia a különböző proxy-kban tárolt objektumok között.Ezen kívül a validációs lépés felgyorsítására, minden belső csomópont sorszámát eltárolják minden memnode-ban is. Ennek köszönhetően, az érvényesítés megtörténhet egyetlen memnode-ban (amennyiben nem volt csomópont szakadás), így nem csak a hálózati késleltetések, de a memnode-ok által keltett zárolások ideje is nagyban csökken.

3. ALAPFOGALMAK

A következőekben bemutatom azokat a fogalmakat, módszereket, amelyek a fő részeit képezik a kiegészítéseknek, amiket a fő cikk írói hozzátettek az alapul vett módszerekhez.

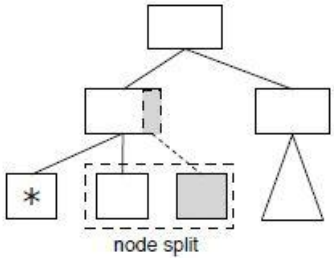
„Piszkos olvasás” (dirtyread)

Az eddigiekben vázolt konkurencia vezérlési módszer két jelentős hátulütővel rendelkezik.

1. A sorszámok memnode-okba történő másolása miatt, minden olyan művelet, ami a sorszámok frissítését vonja maga után (például csomópont szakadás jön létre) nagyon költségesen végrehajthatóak.
2. A konkurencia vezérlés bizonyos esetekben feleslegesen szakíthatja meg a tranzakciót.

Utóbbi esetre látható példa a 2.ábrán, ahol egy tranzakció a *-al jelölt csomópontot olvassa, egy a gyökérből induló bejárás során, ha ekkor egy konkurens tranzakció egy testvér csomópontot szakadásra kényszerít, akkor az eredeti tranzakció abortálni fog, hiszen

megváltozott a szülő csomópontja és az érvényesítés nem fog egyező eredményt adni. Megjegyzendő itt, hogy a félbeszakítás annak ellenére is létrejön, hogy a tranzakció megtalálta a helyes levelet.

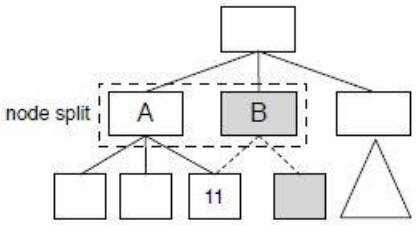


2. ábra

Ezek a problémák kiküszöbölésére vezeti be a cikk az úgynevezett „piszkos olvasásokat”, ami egy újfajta, a dinamikus tranzakciók által végrehajtható művelet lesz. Ennél a fajta olvasásnál a tranzakciónak nem kell az olvasási halmazához adnia a lokális tárról vagy memnode-ról lekérdezett objektumot. Helyette csak, ha később írásra kerül, akkor adódik hozzá az olvasási halmazhoz (még az írás előtt). Ezzel az új művelettel a dinamikus tranzakciók már nem biztos, hogy sorba rendezhetők maradnak, de még mindig lehetséges a B-fa műveletek szigorúan sorba rendezhetővé tétele még akkor is, ha a dinamikus tranzakciók gyengébb elszigeteltséget biztosítanak.

A B-fa műveletek végrehajtásához a proxy-k a gyökértől indulva piszkos olvasásokat használnak egészen a levél előtti szintig, majd a levelet hagyományos olvasási művelettel kérik le. Ennek eredményeképpen egy általános esetben (amikor nincs szakadás) az olvasási halmazban csak a levél csomópont fog szerepelni a teljes út helyett. Ez nyilvánvalóan nagyban csökkenti a validálásra kerülő csomópontok halmazát, valamint már nincs szükség a memnode-okban a belső csúcsok sorszámainak tárolására, ami a beszúrási műveleteket sokkal gyorsabbá teszi és megszünteti a feleslegesen felhasznált területeket.

A piszkos olvasások bevezetése viszont további ellenőrzéseket von maga után, hiszen most már az inkonzisztens adatok nem kényszerítik abortálásra a tranzakciót, aminek következtében hibás információt kaphatunk egy tranzakciótól. Erre látható egy példa a 3.ábrán, ahol egy tranzakció a 11-es kulcsot keresi, egy másik pedig beszúr egy új értéket, ami szakadásra kényszerít egy belső csomópontot. Ezeknek a tranzakcióknak lehetséges egy olyan végrehajtási sorrendje, amiben az első beolvassa az ’A’ csomópontot a szakadás után és helytelenül azt állapítja meg, hogy a 11-es kulcs nincs jelen a fában.



3. ábra

Ezeknek az ellentmondásoknak a kiküszöbölésére úgynevezett határoló kulcsokat (fencekeys) használnak az írók, amik Philip L. Lehman és s. BingYao írásában [4] kerültek bevezetésre. Ezek a

korlátoló kulcsok azt jelölik, hogy egy adott csomópont mely kulcsértékekért felelős függetlenül attól, hogy azok jelen vannak-e a fában vagy sem. Egy keresés során ezeket a korlátokat mindig összevetjük a keresett kulcs értékkel és abortáljuk a tranzakciót, ha az kívül esik az intervallumon. Ennek következtében garantálható, hogy vagy megtaláljuk a helyes levelet, vagy megszakad a tranzakció és így elkerülhetjük az esetleges további problémákat.

A 4.ábrán látható egy teljes fabejárás algoritmus, amely piszkos olvasásokat használ.

Function Traverse(R, k, T)

Input: R – pointer to root of B-tree with at least two levels
k – search key
T – dynamic transaction
Output: sequence of B-tree nodes traversed from root to the leaf responsible for key k in B-tree rooted at R, or else $\perp$ if T aborted

```
1  curPtr := R
2  curNode := T.DirtyRead(internal node at curPtr)
3  ret := ⟨curNode⟩
4  while isInternal(curNode) do
5      if k < lowFence(curNode) OR k > highFence(curNode) then
6          T.Abort(), return ⊥
7      else
8          nextPtr := child of curNode responsible for key k
9          if height(curNode) > 1 then
10             nextNode := T.DirtyRead(internal node at nextPtr)
11         else
12             nextNode := T.Read(leaf node at nextPtr)
13         end
14         ret := ret ∘ ⟨nextNode⟩
15         if height(nextNode) ≠ height(curNode) − 1 then
               // Fatal inconsistency!
16             T.Abort(), return ⊥
17         end
18         curNode := nextNode
19         curPtr := nextPtr
20     end
21 end
   // Reached leaf node
22 if k < lowFence(cur) OR k > highFence(cur) then
23     T.Abort(), return ⊥
24 end
25 return ret
```

4. ábra

Pillanatképek

A sorba rendezhető tranzakciók nagymértékben leegyszerűsítik a komplex elosztott alkalmazások fejlesztését, viszont velejárójuk lehet az is hogy költségesek és gyenge teljesítményt eredményeznek különösen olyan igénybevételeknél, amik hosszú ideig futó tranzakciókból állnak.

Hasonlóan néhány korszerű rendszerhez [4, 5], a Minuet is úgy oldja meg ezt a kihívást, hogy a hosszú ideig futó lekérdezéseket az adatok konzisztens pillanatképein futtatja. A proxy-k úgy készítik a pillanatképeket, hogy a lekérdezések mindig úgy tűnnek, mintha a legfrissebb adaton dolgoznának, ezzel garantálva a szigorú sorba rendezhetőséget.

A pillanatképek egy adott idő pillanatban biztosítanak konzisztens képet a B-fáról, aminek legfontosabb alkalmazhatósága, hogy elkülöníthetjük az elemző lekérdezéseket az OLTP (online tranzakció feldolgozás) terhelésétől.

Írható klónok

A pillanatképek használata hasznos sok elemzési feladathoz, de komplex problémákhoz gyakran előnyös, ha közvetlenül tudjuk kezelni az adatok párhuzamos verzióit. Sokfelé ismeretes a szerteágaztatás funkciója, ami megtalálható a legtöbb verziókezelő rendszerben és ez a módszer igen hasznos lehet egyes kifinomultabb elemzések végrehajtásakor. Például egy elemző lehet, hogy kísérletezni szeretne kicsit módosított adatokkal és összevetni a különböző eredményeket. Ezt megtehetné úgy, hogy exportálja az adatokat és külön módosítja azokat, de ennél előnyösebb megoldás, ha helyette egy elágazást készít ugyanazon a rendszeren belül. Első sorban, ha másoláskor íródó megközelítést használunk időben és helyfoglalásban is hatékonyabb működést kapunk, hiszen elég egy kis részét módosítani az adatoknak, a teljes exportálás helyett. Továbbá a rendszer több verziójának karbantartása hasznos lehet integritási ellenőrzések futtatásakor, vagy elemzések eredményeinek összehasonlításához.

4. MEGOLDÁSOK, EREDMÉNYEK

Az előző fejezetben megadtuk az alapvető elveket, amiket felhasznált a cikk az alapul szolgáló módszerek javítására. A továbbiakban kifejtjük, hogy ezeken az elveken belül milyen megvalósításokat, további módszereket alkalmaztak.

Íráskor másolódó pillanatképek (copy-on-writesnapshots)

Kihasználva, hogy a Minuet elosztott B-fákban tárolja az adatokat, alkalmazhatunk már létező technikákat a konzisztens pillanatképek hatékony létrehozásához [6]. Egy magas szinten, amikor egy új pillanatkép kerül létrehozásra, minden B-facsomópont másolásra kerül mielőtt frissítve lenne, így a pillanatkép nem kerül felülírásra.

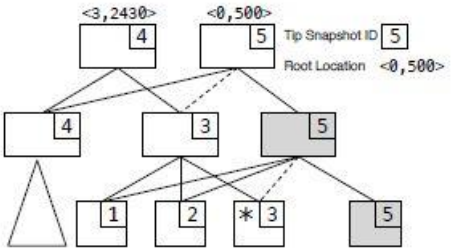
Minden pillanatképhez rendelünk egy sorszámot (*snapshotid: 64bit-es egész*), amik szigorúan monoton módon követik egymást és azt reprezentálják, hogy a pillanatképek milyen sorrendben keletkeztek. A legújabb pillanatképet fogjuk a csúcs képnek hívni (tipsnapshot) és mindig csak ez lesz írható, az összes többi csak olvasható. Egy új pillanatkép létrehozás tehát azt vonja maga után, hogy az aktuális csúcs képet csak olvashatóvá tesszük és az újonnan kreált pillanatkép lesz az új csúcs kép (nagyobb sorszámmal, mint az előző), ami minden B-fa csomópontot megoszt az előzővel egészen addig, amíg nem kerül felülírásra egy csomópont ebben az új csúcs képben. Minden B-fa csomóponthoz számon tartjuk, hogy melyik pillanatképben jött létre. Az aktuális csúcs kép azonosítóját és a hozzá tartozó gyökér csomópontot egy jól definiált helyen tartjuk számon a Sinfonia cím rendszerében, így az mindig elérhető.

A tranzakciók bármely pillanatképről olvashatnak, de a szigorúan sorba rendezhető és az aktuális (rendszerben up-to-date) adatokkal dolgozó olvasásoknak mindig a csúcs képet kell használniuk. Ilyenkor az olvasás sikertelen, ha konkurensen egy újabb pillanatkép jön létre.

Amikor egy B-fa csomópontot frissítünk egy *s* sorszámú pillanatképben, *s*-t összehasonlítjuk a csomópontban tárolt pillanatkép azonosítóval, ha *s*nagyobb mint ez az azonosító, akkor másolásra kerül ez a csomópont és a másolatot frissítjük, valamint a pillanatkép azonosítóját *s*-re állítjuk. Ez után a csomópont szülőjét is frissíteni kell, hogy az új másolatra mutasson, ami lehet, hogy egy újabb másolást fog kiváltani. Tehát egy levél frissítésekor lehetséges, hogy a gyökérig

minden csomópontot másolni kell majd, kivéve a gyökeret, hisz az már másolásra került a pillanatkép létrehozásakor.

Erre látható példa a 5.ábrán, ahol minden pillanatkép gyökere a memória helyének címkéjével van ellátva (<3,2430> és <0,500>). Minden csomópontnak jelölve van a pillanatkép azonosítója és a *-al jelölt csomópont kerül frissítésre az 5-ös pillanatképben. Mivel a frissítendő csomópont a 3-as pillanatképben jött létre, ami kisebb, mint az aktuális, másolásra kerül egy új csomópontba (szürkével jelölve). Hasonlóan a szülő csomópontot is másolni kell, mielőtt a mutatóját frissíteni tudjuk és a gyökér mutatója is frissítésre kerül, de az már nem kerül másolásra, mert eleve az 5-ös pillanatképben van. (a régi kapcsolatokat szaggatott vonalak jelzik)



5. ábra

Ez az íráskor másolódó elvű eljárás a Minuetben implementálható a fentebb említett dinamikus tranzakciók segítségével.Az ide kapcsolódó algoritmus a 6.ábrán látható.

Function CreateSnapshot(*sid*, *loc*, *T*)

Input: T – dynamic transaction
Output: sid – snapshot id of the snapshot created
loc – location of root node for the snapshot created

```
1  T.Read(tipSnapshotID)
2  T.Read(tipSnapshotRootLoc)
3  sid := tipSnapshotID
4  loc := tipSnapshotRootLoc
5  tipSnapshotID += 1
6  newRootLoc := Allocate(NodeSize, T)
7  CopyRoot(tipSnapshotID, tipSnapshotRootLoc, newRootLoc, T)
8  tipSnapshotRootLoc := newRootLoc
9  T.Write(tipSnapshotID)
10 T.Write(tipSnapshotRootLoc)
11 return (sid, loc)
```

6. ábra

Piszkos olvasások és pillanatképek kapcsolata

Az egyik meghatározó eltérés a Minuetben az eddigi munkákhoz képest, a piszkos olvasások alkalmazása. Ennek a technikának az implementálásánál azonban óvatosnak kell lennünk, amikor pillanatképekkel együtt alkalmazzuk. Felmerül ugyanis az a problémás lehetőség, hogy egy bejárás a helyes levél csomópontnál áll meg, de egy rossz pillanatképben.

Ennek a problémának a kiszűrésére bevezetünk egy újabb állapotot minden B-fa csomópontban: a pillanatkép azonosító, amibe a csomópont másolva lett (ha van ilyen). Megjegyzendő, hogy minden csomópont maximum egyszer kerülhet másolásra, ezért a plusz tárolandó értékek száma jól definiáltan meghatározható.

Egy írás vagy olvasás során az *s* pillanatképben, ha a keresés talál egy csomópontot, ami másolva lett egy pillanatképben, aminek azonosítója kisebb vagy egyenlő mint *s*, akkor abortál, mivel a csomópontnak már a másolatát kéne vizsgálnia.

Bár ez az eljárás garantálja a helyességet, az eddig leírt helyesség ellenőrzési technikával teljesítménybeli problémákhoz vezethet.

Kölcsön vett pillanatképek

Habár a Minuet működése során nem számítunk gyakori pillanatkép készítésre, mégse elhanyagolható hogy ezek nagyon költséges műveletek és viszonylag kevés konkurens pillanatkép létrehozás is nagyban ronthatja a teljesítményt. Ennek a problémának a kiküszöbölésére további két optimalizálási módszert vezetnek be a szerzők.

1. Minden pillanatképet egyesével hozunk létre egy centralizált szolgáltatás segítségével, ezzel nagyban csökkentve a versengést.
2. Megfigyelhető, hogy mivel minden pillanatkép, amit ez a szolgáltatás visszaad csak olvashatóak, ezért megoszthatnak egymás közt azonos pillanatképeket, anélkül hogy egymást akadályoznák.

Annak eldöntésére, hogy melyik lekérdezéseknek kéne megosztva használnia egy pillanatképet bevezetünk egy új technikát, ami ezt automatikusan meghatározza. A technikát kölcsön vett pillanatképeknek fogjuk hívni és ez lehetővé teszi, hogy egy lekérdezés egy olyan pillanatképet használjon, amit egy konkurens lekérdezés hozott létre, amennyiben ezzel megmarad a szigorúan sorba rendezhetőség tulajdonság.

A kölcsönzés a következő képpen történik:

1. A és B kliens konkurensen próbálnak létrehozni egy pillanatképet.
2. B kérése A után kerül sorra az ütemezésben.
3. Ha A pillanatképe akkor jön létre, amikor B már várakozik a sorban, akkor az A által létrehozott kép olyan esetet reprezentál, ami B kérésének végrehajtása közben is érvényes.
4. Ekkor B kölcsön veheti az A által készített pillanatképet, ahelyett hogy sajátot készítene.

Function CreateSnapshotProc()

Output: sid – the snapshot id that was created or borrowed
loc – Sinfonia address of the root node for sid
Variables: sid, loc – snapshot id and root node address, shared/static (initially $sid = 0$ and loc is the address of the initial root node)
$mutex$ – mutual exclusion lock, shared/static
$numSnapshots$ – atomic integer, shared/static (initially 0)

```
1  tmpNum1 := numSnapshots
2  lock(mutex)
3  tmpNum2 := numSnapshots
4  if tmpNum2 < tmpNum1 + 2 then
       // unable to borrow, must create new snapshot
5      Dynamic transaction T
6      CreateSnapshot(sid, loc, T) // see Figure 6
7      if T.Commit() = false then continue at line 5
8      ++numSnapshots
9  else
       // safe to borrow last snapshot, nothing to do
10 end
11 retSid := sid
12 retLoc := loc
13 unlock(mutex)
14 return retSid, retLoc
```

7. ábra

A 7.ábrán, a fentebb leírt kölcsönzést bemutató pszeudó kód látható.

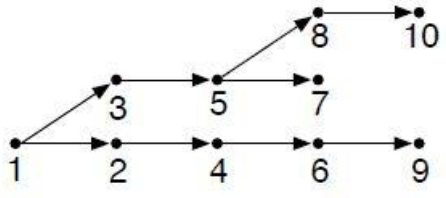
Megjegyzendő, hogy az új pillanatkép készítésének eldöntése megtehető a fenti módon a pillanatkép kreáló szolgáltatással, vagy elosztott módon a proxy-kon belül.

Szemétgyűjtés

A rendszer futása közben a pillanatképek létrehozása miatt előbb-utóbb elfogy a memória. Ennek kiküszöbölésére folyamatosan törölni kell a régi pillanatképeket. A Minuet nyilván tart egy minimum pillanatkép azonosítót és ennél kisebbre nem fogad el kéréseket a kliensektől. Egy mögöttes folyamat folyamatosan pásztázza a csomópontokat és összegyűjti azokat, amik már másolva lettek egy pillanatképbe, aminek azonosítója nagyobb vagy egyenlő, mint a globális legalacsonyabb. Ezek a csomópontok biztonságosan törölhetőek.

Szerteágazó verziók

Az elágazásokat ugyanaz a másoláskor íródó logika használatával támogatja a rendszer, ami már a pillanatképeknél leírásra került. Annyi a különbség, hogy a monoton sorszámozással történő pillanatkép létrehozás mellett lehetőség van már létező pillanatképből egy elágazást létrehozni. Ennek következtében egy logikai fát kapunk, aminek elemei fizikai B-fa verziók. A logikai fa belső csúcsai a csak olvasható pillanatképeket jelentik, míg a levelek az írható, azaz csúcs képeket. Így például a 8.ábra alapján a kliensek a 7,9,10-es pillanatképeket írhatják, míg a többit csak olvashatják.



8. ábra

Mivel ezzel megszűnik a pillanatképek természetes egymáshoz viszonyított sorrendje, már nem egyértelmű, hogy hogyan osszuk ki az azonosítókat. Megmaradunk annál a megoldásnál, hogy a létrehozás szerinti sorrendet tároljuk, ám ez esetben nem egyértelmű, hogy melyik a megfelelő csúcs képe egy korábbi pillanatképnek, például a 8.ábrán az 5-ös képnek lehet a 7-es és a 10-es is. A rendszer alapvetően az előbb létrejött ággal próbálkozik, de a kliens ezt megadhatja másképp is.

A verziós fa nyilvántartásához bevezet a cikk egy pillanatkép katalógust, ami meta-információkat tárol minden pillanatképről. Tárolja az azonosítóját, a gyökér helyét és az először létrehozott ágát (ha van), ezt ág azonosítónak hívjuk (branchid) és amennyiben ez NULL, nincs elágazás abból a pillanatképből.A Sinfonia rendszerben a katalógus egy különálló B-fában tárolódik, amire már nem engedélyezett a pillanatképek készítése.

Új pillanatkép készítéséhez növeljük a globális azonosító számot és létrehozunk egy új bejegyzést a katalógusba. Továbbá szükség van egy új gyökér csúcs hozzárendelésére az új pillanatképhez és frissíteni kell az ág azonosítóját annak a pillanatképnek, amelyikből létre lett hozva az új. Ezek a műveletek atomosan hajtódnak végre egy dinamikus tranzakcióval.

A kölcsön vett pillanatképek technikája ugyanolyan jól használható az elágazó verziókon is, mint eddig. Sőt a katalógus csak egyszerűsíti ezt, mivel minden egyes levélen a verzió fában függetlenül jöhetnek létre kölcsönzések.

A piszkos olvasás viszont sajnos már nem működik egyértelműen a verzió fa bevezetésével. Ennek taglalása a következő fejezetben történik.

Piszkos bejárások

A korábban definiált módszer, miszerint egy bejárás egészen addig jó, amíg nem talál egy olyan csomópontot, ami a vizsgált azonosítónál kisebb vagy egyenlő sorszámú pillanatképbe lett másolva, itt nem érvényes hiszen lehet, hogy egy másik ágon van a pillanatkép, ahova a másolás történt.

Egy egyszerű megoldás lenne, hogy egy teljes leszármazott halmazt tárolunk minden csomóponthoz, amiben benne van minden pillanatkép azonosítója, amibe a csomópont másolva lett. Ezzel a módszerrel meghatározható ugyan, hogy mikor kell abortálni és mikor lehet folytatni, viszont a leszármazott halmaz korlátok nélkül nőhet így nem egy praktikus megoldás.

Részbeni megoldás az előző technika kiegészítése egy konstans β korlát bevezetésével, ami meghatározza, hogy maximum hány elágazás lehet egy csomópontból. Ez még mindig nem elég, szükség van továbbá egy invariáns feltétel folyamatos betarttatására is, ami a következő:

„Ha egy az *x* pillanatképben készült B-facsomópont másolásra kerül egy *x* leszármazottjainak (verziós fában) valamilyen *C* részhalmazába, akkor létezik egy olyan $C' \subseteq C$ maximum βelemszámú részhalmaz, amelyre $\forall y \in C : C'$ tartalmaz egyet *y* ősei közül."

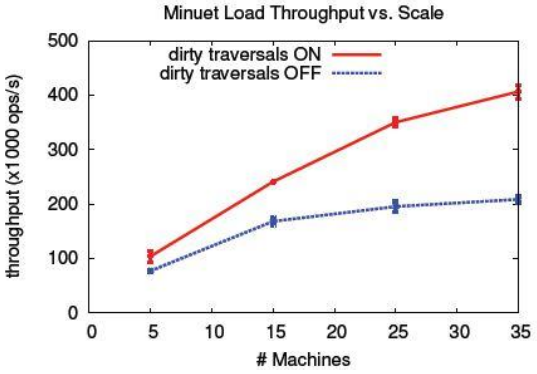
Ennek a tulajdonságnak a betartatása diszkrecionális másoláskor íródó műveletekkel történik (discretionalcopy-on-writeoperations). Ez a következő képpen működik, a 8.ábra példáját véve. Tegyük fel, hogy egy B-fa csomópont *x*=1 pillanatképben jött lére és aztán íráskor másolva lett 7,9,10-re.Ekkor a β=2 korlát és az invariáns megköveteli, hogy a csomópont másolva legyen 7 és 10 egy közös ősére is, ami nem *x*, azaz lehet a 3-as vagy az 5-ös. Lehetséges például, hogy *C*={3,7,9,10} és *C'*={3,9}. Tehát ha egy csomópont másolva lett 7-esre és 10-esre is, akkor a későbbi másolás kivált egy diszkrecionális másolást például a 3-as pillanatképbe és frissíti a leszármazott halmazait 1-esnek és 3-asnak, így az eredmény olyan mintha 1-esben létrejött csomópont először 3-asba majd 7-esbe végül 10-esbe lett volna másolva.

Tesztel és mérési eredmények

A leírt módszerhez számos teszt eset került kipróbálásra, amik kivitelezéséhez két eszközt használtak. A nyílt forráskódú Yahoo CloudServing Benchmark (YCSB) [7] segítségével történt a szabványos és ismételhető munkaterhelés generálása. A Minuet-tel összehasonlítandó rendszer pedig egy modern központi memóriát használó kereskedelmi adatbázis volt, amire csak a CDB rövidítéssel hivatkoznak.

1. **Késleltetési idő és átviteli teljesítmény**

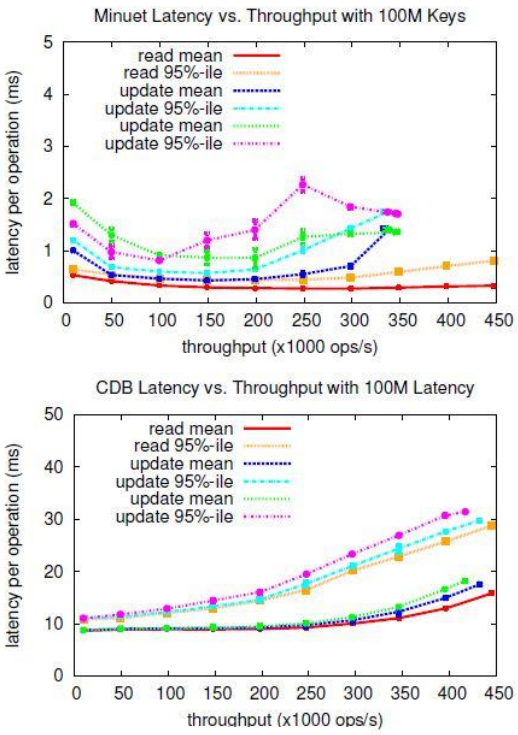
   A piszkos olvasások hasznosságának kiértékeléséhez a Minuet átviteli teljesítményét mérik, miközben egységesen véletlenszerű kulcsokkal töltenek fel egy kezdetben üres B-fát. Külön futtatásokat végeztek, ahol engedélyezték a piszkos olvasásokat és nem többszörözték a belső csomópontok sorszámait, valamint ahol nem engedélyezték a piszkos olvasásokat, viszont többszörösen eltárolták a sorszámokat. Az eredményeket a 9.ábra mutatja.



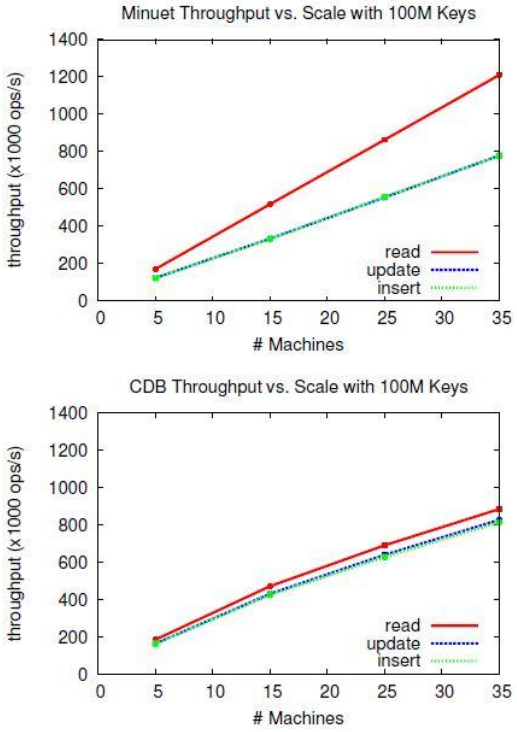
9. ábra

A 10.ábrán a késleltetés-átviteli teljesítmény görbék láthatóak. A Minuet olvasási műveletek esetén az átlag késleltetés 0.4ms alatt van, ami egy nagyságrendel jobb a CDB-nél mérteknél. Beszúrás és frissítés műveleteknél pedig a Minuet kevesebb mint 1 ms késleltetést kap 20% és 80% közötti átviteleken, ami szinten több mint egy nagyságrendel jobb a CDB teljesítményénél.

A 11.ábrán a skálázhatóságot láthatjuk miszerint mindkét rendszer jól teljesít, de a Minuet kicsit lineárisabban növekszik és kicsit gyorsabb olvasásai vannak magasabb szinteken. Megfigyelhető még, hogy nagyobb a különbség az írási és olvasási műveletek között is a Minuet esetében, több mint 50%-al gyorsabbak.



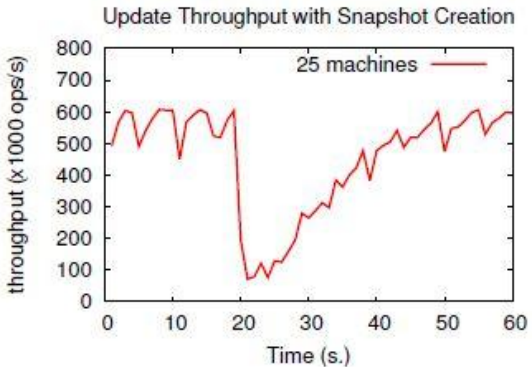
10. ábra



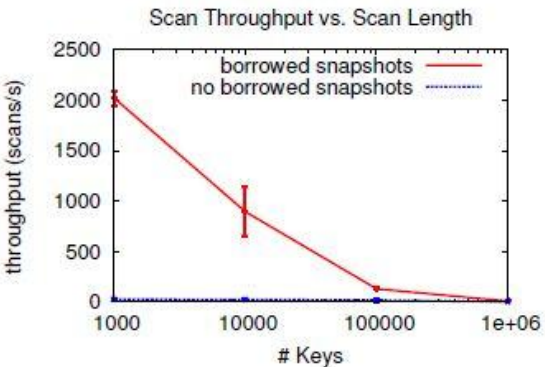
11. ábra

2. Pillanatkép tesztek

A 12.ábra egy egyszeri pillanatkép hatását mutatja az átviteli teljesítményre. Másodpercenkénti egyenletes terhelések során mindig mérik az átvitelt, majd a 20-dik másodpercben kibocsátottak egy pillanatkép kérést, ami hatására látványosan csökkent az átvitel. Ennek ellenére az tapasztalható, hogy viszonylag gyorsan visszaáll az eredeti működésre a rendszer.



13. ábra



12. ábra

A 13.ábra pedig a kölcsön vett pillanatképek hatékonyságát mutatja, amin azt tapasztalhatjuk, hogy viszonylag rövid vizsgálatoknál (1000kulcs) drasztikusan javítanak a teljesítményen, viszont ahogy a kulcsok száma növekszik, egyre közelíti a kölcsönzés nélküli pillanatképek teljesítményét és végül 100.000.000 körüli kulcs esetén már szinte megegyezik a kettő.

5. ÖSSZEGZÉS, TOVÁBBI LEHETŐSÉGEK

Minuet lehetővé teszi az adat centrikus üzletágak számára, hogy eleget tegyenek stratégiai szükségleteiknek egy olyan skálázható platform biztosításával, ami mind a tranzakcionális mind az elemző munkaterhelésnek megfelel. Emellett erős eszközöket biztosít kifinomult elemzések, adat megosztás, archiválás és adat védelmi feladatok végrehajtásához. A konvencionális rendszerekkel ellentétben skálázható elemző lekérdezések futtatását is lehetővé teszi. Ez a megoldás egyesíti a tradicionálisan különálló rendszereket és munkaterheléseket egyetlen közös platformba, ami remélhetően használati esetek és szolgáltatások egy teljesen új osztályát fogja bevezetni.

6. IRODALOMJEGYZÉK

[1] Benjamin Sowell, WojciechGolab, MehulA. Shah:
Minuet: A scalabledistributedmultiversionB-tree
Proceedings of the VLDB Endowment, Vol. 5, No. 9, Pages 884-895, May 2012

[2] M. K. Aguilera, W. Golab, M. A. Shah:
A practicalscalabledistributedB-tree. PVLDB, 1(1):598–609, 2008.

[3] M. K. Aguilera, A. Merchant, M. A. Shah, A. C. Veitch, C. T.Karamanolis:
Sinfonia: A newparadigmfor building scalabledistributedsystems,
ACM Trans. Comput. Syst., 27(3):5:1–5:48,2009.

[4]AlfonsKemper, Thomas Neumann:
HyPer: A hybrid OLTP&OLAP main
memorydatabasesystembasedonvirtualmemorysnapshots
ICDE '11 Proceedings of the 2011 IEEE 27th International Conferenceon Data
Engineering, Pages 195-206

[5]YuCao, ChunChen, FeiGuo, DaweiJiang, YutingLin, BengChinOoi, HoangTamVo, Sai Wu, QuanqingXu:
ES2: A clouddatastoragesystemforsupportingboth OLTP and OLAP
ICDE '11 Proceedings of the 2011 IEEE 27th International Conferenceon Data
Engineering, Pages 291-302

[6] OhadRodeh: B-trees, shadowing, and clones
ACM Transactionson Storage, Volume 3, Issue 4, February 2008

[7] Brian F. Cooper, Adam Silberstein, Erwin Tam, RaghuRamakrishnan, Russel Sears:
Benchmarking cloudservingsystemswith YCSB
SoCC '10 Proceedings of the 1st ACM symposiumonCloudcomputing, Pages 143-154