Hibrid tárkezelési módszer adatbázis rendszerekhez

Xin Liu (University of Waterloo, Canada, x39liu@uwaterloo.ca),
Kenneth Salem (University of Waterloo, Canada, ksalem@uwaterloo.ca)

Hybrid Storage Management for Database Systems

Cikke alapján a tanulmányt készítette:

Kukovecz János
Nyári István
Szentkirályi Károly
Tverdota Dávid

2013. November

Absztrakt leírás

A flash alapú szilárdtest meghajtók (SSD~solid state drive) használata egyre inkább növekszik az adattárolás területén.

Az SSD-k adatbázisban történő felhasználásakor előforduló kritikusabb problémák:

- Az SSD kezelése - menedzselése
- Rendelkezésre álló jelenlegi buffer tárért (current buffer pool) felelő algoritmusok effektíven működnek-e majd az új rendszerben

A tanulmány egy olyan hibrid tárhely kezelő rendszer kialakítását célozza, mely az eddigi általános tárak mellett (HDD) a szilárdtest meghajtókat (SSD) is felhasználja az adatbázisban. Az így elkészült rendszer olyan költség-tudatos csere algoritmusokat alkalmazna, melyek figyelembe veszik mind a HDD, mind pedig az SDD esetén megjelenő I/O költségeket és azok közti különbségeket. Az ilyen hibrid rendszerekben az SSD-t célzó fizikai elérések mintája (SSD-hez történő fizikai hozzáférések mintája - statisztikája) nagyban függ az adatbázis kezelő rendszer buffer tárának (buffer pool) kezelésétől, annak menedzselésétől. Az említett minták gyakorlati tanulmányozásából származó adatok alapján a szerzők létrehoztak egy költség-beállított (cost-adjusted ~ költséghez állított) irányelvet (eljárásmódot) mely hatékonyan kezeli az SSD-t. A továbbiakban tárgyalt algoritmusokat és eljárásokat MySQL's InnoDB tár motoron implementálták és TPC-C terhelés felhasználásával demonstrálták, hogy a költség-tudatos algoritmusok messze jobb eredményeket hoznak, mint az algoritmusok nem költségtudatos – egyszerű - változataik.

Tartalomjegyzék

1.Bemutatkozás 3
2. Rendszer áttekintés 4
3. Buffer Pool Menedzsment (kezelés) 6
3.1 A GD2L MySQL implementációja 7
4. A költség tudatos cache-elés hatása 8
5. SSD kezelés 9
5.1 CAC: Cost-Adjusted Caching~Költséghez igazított Cache-elés 10
5.2 A „miss rate expansion factor” 12
5.3 Szekvenciális I/O 13
5.4 Hibakezelés 13
6. Értékelés 14
6.1 Metodológia 15
6.2 Paraméter költség kalibráció 16
6.3 GD2L és CAC analízis 17
6.3.1 GD2L vs LRU 17
6.3.1 CAC vs CC 18
6.4 LRU2 és MV-FIFO összehasonlítása 19
6.5 Az eldobási zóna hatásai 20
7. Kapcsolódó munkák 20
8. Összegzés 21
9. Referenciák 22

1.Bemutatkozás

A flash memóriák használata a fogyasztói társadalomban már évek óta elterjedt főleg telefonokban, kamerákban, olyan eszközökben, ahol a hordozhatóság és a kevés mozgatható/mozgó alkatrész előnyt élvez. Bár az SSD-k tároló kapacitásukhoz képest jóval költségesebbek, mint - a nagyteljesítményű, nagyméretű adatok véletlen elérésekor jól teljesítő - nagytestvéreik a HDD-k, rendkívül kedvező I/O műveleti költségeiknek köszönhetően egyre inkább teret nyernek, és begyűrűznek a szerverüzemeltetési környezetekbe is.

A tanulmányban a szerzők egy olyan rendszert alkotnak, ahol az adatbázis kezelő rendszer mindkét tároló típust látja (SSD, HDD) és kezeli, azaz ő dönti el mikor, melyiket használja, adott kritériumok mellett. Két fő probléma kerül előtérbe ebben az esetben, melyek:

Eldönteni mely adat maradjon meg a buffer tárban (DBMS – adatbázis kezelő rendszer, buffer pool)

Az első problémára adott megoldás nagyban függ a rendszer hibrid jellegétől, hiszen a buffer-ből az SSD-re kilakoltatott/kidobott (evict) blokkokat sokkal gyorsabb onnan visszakérni (beolvasni) mint HDD-ről. Épp emiatt lesz hatásos a költség-tudatos adat kezelés, mely figyelembe veszi a kettő közti különbséget (korábban említett költségtudatos algoritmusok)

Eldönteni mely adatok kerüljenek az SSD-re

Ez nyílván abból következik, hogy az SSD mérete nem lesz elegendő egy teljes adatbázis tárolására (lássuk be, hogy vannak ilyen ritka esetek☺). Az eldöntés majd az adatok fizikai hozzáférés mintájából (statisztika – kit milyen gyakran olvasunk/írunk) adódik majd, ami az adatbázis kezelő rendszer (DBMS) terhelés kezelésétől és a buffer tárhely (buffer pool) kezelésétől függ.

Vegyük figyelembe, hogy a tárgyalt problémák egymástól is függnek. A buffer pool-on végbemenő cseréket illető döntések függnek a cserélni kívánt lapok (adatok) helyétől (SSD vagy HDD), tekintve hogy az mind a kiírást (kilakoltatást/kidobást [eviction]) és a beolvasást is érinti.

Tehát a döntés, hogy egy adott oldalt az SSD-re helyezhetünk, erősen függ attól, hogy az adott lapot miként használjuk, milyen gyakran olvassuk be vagy írunk bele, ezek pedig a buffer manager – buffer menedzsertől függnek majd.

Például a tanulmányban bemutatott GD2L (GreedyDual2Level) algoritmus (csere eljárás – lapcsere) egy a HDD-ről az SSD-re áthelyezett lap fizikai írás és olvasási rátájának jelentős megemelkedését eredményezi majd, tekintve, hogy a GD2L hajlamos gyorsan kilakoltatni/kidobni a lapokat a buffer pool-ból az SSD-re.

A tanulmány ezeket a függőségeket egy előrelátó (becslő) SSD menedzser alkalmazásával közelíti meg, illetve próbálja kezelni. Mikor a rendszer egy olyan döntés elé áll, hogy mozgasson-e egy adott lapot az SSD-re, a CAC (Cost Adjusted Cache, ez az SSD menedzserben használt algoritmus) előrelátó csere eljárás megkísérel becslést adni arra (meghatározza kb), hogy hogyan változik a lap fizikai I/O terhelése az SSD-re kerülés esetében. A lap ezután csak akkor kerül az SSD-re, ha CAC úgy ítélte meg ezek alapján, hogy megfelelő „alany”. Ezt

követően az adatbázis kezelő rendszer buffer menedzsere meghozza az ő saját költség-tudatos csere döntéseit a meglévő lapokra, melyek a bufferben találhatóak.

A tanulmány fő „termékei”:

GD2L költség-tudatos algoritmus az adatbázis kezelő rendszer buffer pool menedzselésére, hibrid rendszerekhez, mely figyelembe veszi mind az általános buffer menedzsment feladatait (exploiting locaility, scan resisitance), mind pedig a tényt hogy hibrid rendszerükben a tárolóeszközök eltérőek és eltérően viselkednek, teljesítenek. A GD2L a GreedyDual algoritmus egy megszorított, az adott rendszerhez illeszkedő változata.

CAC előrelátó költség-alapú (becslő) technika az SSD menedzseléséhez, mely jól együtt működik a GD2L-el. A CAC feltételezi, hogy egy lap SSD-re kerülésekor megváltozik annak elérhetőségi költsége, és ezt a költséget becsli, a rendszer jobb működésének érdekében.

Mindkét fentebb említett és körvonalazott technikát MySQL InnoDB adatbázis keretben implementálva TCP-C terhelés alatt tesztelve, az adatokból teljesítmény bemutatás készült – összehasonlítás más eszközökkel, kiértékelés.

2. Rendszer áttekintés

Az „1. ábrán” látható a rendszer egy vázlatos terve. Az adatbázis kezelő rendszer két különböző típusú tároló egységet lát a HDD-t, melyen minden adatbázis lapot tárolunk (a kezelő rendszer másodlagos tárolási-elrendezési irányelve alapján), és az SSD-t, melybe a CAC és a kezelő

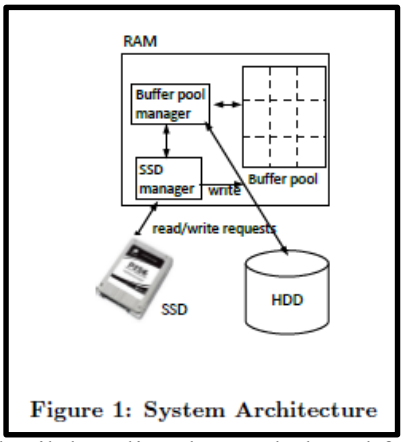


Figure 1: System Architecture

buffer menedzsere dönti majd el mik kerüljenek, a terheléstől függően. Az SSD-n tehát a HDD-n tárolt lapok közül valahány található, pontosabban azok másolatai. Ugyanígy igaz ez az adatbázis kezelő rendszer (innentől DBMS) buffer pool-jára is. Tehát az adatbázist alkotó lapokból a HDD-n mindenképp tárolunk egy példányt, továbbá ezekből (méretekhez igazodva – SSD és buffer méret) valamennyi éppen átmenetileg tárolódhat az SSD-n és/vagy a buffer pool-ban.

Amikor a DBMS-nek szüksége van egy lapra, először is megnézi, hogy a saját buffer pool-jában meg van-e a kérdéses lap. Amennyiben megvan, beolvassa, ha nincs akkor az SSD-n tárolt lapok között keres (mivel innen gyorsabb olvasni, az SSD-n lévő lapokat az SSD

menedzsere nyílván tartja) – ha itt megtalálja, beolvassa a buffer pool-ba az SSD-ről. Végső esetben a HDD-ről olvassa be a keresett lapot a buffer-be.

Abban az esetben, ha a kért lap nem található a buffer pool-ban - tehát be kell olvasnunk valahonnan (SSD vagy HDD) - de a buffer pool éppen tele van, akkor a buffer-ből ki kell lakoltatnunk, el kell dobnunk (ki kell írni) egy lapot, a menedzser lapcsere irányelve alapján (3-as rész foglakozik vele részletesebben). Az így kidobott lapot az SSD menedzsere felülvizsgálja abból a célból, hogy az SSD-re rakhatjuk-e (admission policy), amennyiben még ott nem szerepel. Tekintve, hogy van hely az SSD-n és minden fentebbi feltétel teljesül, a lap az SSD-re íródik, ha nincs hely, akkor az SSD menedzserének döntenie kell, hogy már az SSD-n lévő lapok közül melyiket dobja ki (evict), hogy az új beférjen (replacement policy alapján, az említettekkel az 5-ös rész foglalkozik részletesebben). Az SSD-ről történő eldobáskor/kilakoltatáskor, ha az adott lap frissebb, mint a HDD-n lévő másolata, akkor a frissebbet a HDD-ra kell írni a művelet elvégzése előtt, hogy a változtatások ne tűnjenek el. Mindezt az SSD menedzsere egy átmeneti bufferen keresztül intézi a memóriába.

Feltételezés szerint a DBMS buffer menedzsere aszinkron laptisztító technikát használ (ami a DBMS alkalmazások írási latency-jének elrejtésére egy elterjedt módszer). Mikor a DBMS buffer menedzsere kiválaszt egy dirty page-et (~piszkos lapot) megtisztításra (változtatások visszaírása), akkor előbb az SSD-re íródik, amennyiben a lapot ott is tároljuk. Előfordulhat, hogy a kérdéses lapot az SSD-ből már kidobtuk valamilyen oknál fogva, ekkor ugyanúgy, ahogy már korábban a lapok kialkotatásakor leírtuk, az SSD menedzser eldönti, hogy a lap az SSD-re kerülhet-e (admission policy). Amennyiben ráírhatjuk az SSD-re, akkor az SSD-re kerülnek a változtatások (flush), más különben a HDD-ra kell írnunk (ami ugye költségesebb).

A buffer és SSD kezelés előbb leírt módjának két kulcs tulajdonsága:

A lapok SSD-re engedélyezése (írása – admission) csak két esetben történhet: bufferből való kilakoltatás/eldobás vagy tisztítás. Azaz az SSD-ra csak a buffer-ből érkezhetnek adatok, a HDD-ről nem töltünk lapokat az SSD-re. Ennek oka, hogy ezáltal minimalizálható a „cache inclusion” (cache-ek egymásba ágyazása, burkolása) előfordulása [[pl.: lapok duplikálódása az SSD-n és a buffer pool-ban]].

Dirty page-ek tisztításakor, a DBMS buffer pool-jából a változtatott adatok vagy az SSD-re vagy pedig a HDD-re mennek, de nem mindkét tárolóra egyszerre.

Szemben azzal a megközelítéssel, hogy az SSD-n keresztül történjen a beolvasás is (HDD-ről), a fentebb taglaltaknak az a rendkívüli előnye van, hogy így az SSD nagyban javíthatja a DBMS írási és ezáltal az egész rendszer teljesítményét (legalábbis, amíg azok SSD-re irányulnak).

Hátrány azonban, hogy a legfrissebb (a pillanathoz legközelebbi) még nem „bebufferelt” lap mindkét eszközön megtalálható lehet. Mindazonáltal, mivel a DBMS a dirty page-eket mindig az SSD-re írja tisztításkor - ha azok már ott vannak, vagyis tisztításkor az SSD-re kerülnek a változtatások is, mivel a lap már az SSD-n volt – egész biztos, hogy az SSD változata a lapnak legalább annyira aktuális, mint a HDD-n tárolt változat. Így, hogy biztosítsuk, hogy a DBMS mindig a legutóbb írt változatot megszerezhesse, elég, ha tudja mely lapok másolatai tárolódnak az SSD-n (már ha vannak ilyen másolatok ott). Ennek támogatására az SSD menedzsere egy hash map-et tárol, melyben a tárgyalt lapok valamiylen azonosítói szerepelnek. Hiba utáni helyreállítás céljából az SSD checkpoint-os technikát alkalmaz (bővebben 5.4-es rész) a hash map gyors helyreállításához.

3. Buffer Pool Menedzsment (kezelés)

Több költség-tudatos fájl cache-elésre létező algoritmus jelent már meg, mint a balance vagy a GreedyDual, melyek a fájlok méretét és a fájlok elérési idejét is figyelembe veszik. (egyenlő[e] méretű objektumok[Obj]- változó[v] elérési idő[Eid], v Obj – v Eid, e Obj – e Eid) A GreedyDual (mostantól GD) az egyenlő méretű objektumok, de azok változó költségű elérésének problémáját célozza. A tanulmány során tárgyalt GD2L, a GD egy 2 szintű változata (Greedy Dual 2 level).

A GD valójában több algoritmus általánosítása (LRU [Least Recently Used], FIFO [First In First OUT]). Működésekor minden p cache-elt laphoz egy nem negatív H értéket rendel. Amikor a lap a cache-be kerül H értéke beállítódóik az adott lap elérési költségére. Mikor a buffer betelik, és új lap érkezik, a legkisebb H (~H_{min}) értékű lapot eldobjuk/kilakoltatjuk, a többi lap H értéket pedig H_{min}-nel csökkentjük. Ezzel egyfajta „lap-öregedési” mechanizmust alkotunk, mely garantálja, hogy legközelebb azt a lapot dobjuk ki, melyet már nagyon régen nem használtunk. (Így egyben zökkenőmentesen integráltuk a lokalitást [locality] is.)

A GD-t leggyakrabban a lapok elsődleges sorával implementálják, (a priority queue kulcsa a H értékek lesznek). Így egy találat és eldobás O(log k) költségű. Egy másik költség a H értékek csökkentése az imént említett művelet után, ami k darab kivonást jelent. Azonban egy technika alkalmazásával elkerülhető a kivonásokkal járó költségek. Az ötlet, egy L inflációs érték bevezetése, melynek értékével eltoljuk az összes jövőbeli H érték beállítását (tehát mikor új lap kerül a bufferbe, nem az elérési költségét kapja H-jába, hanem H+L-et).

Az „1. táblázaton” láthatóak a következőkben használt jelölések feloldásai:

Jelölés	Jelentés
R_D	lap olvasási költsége HDD-ről
W_D	~ írási költsége HDD-re
R_S	~ olvasási költsége SSD-ről
R_W	~ írási költsége SSD-re

táblázat

A GD2L két (prioritásos) sort használ a buffer-ben tárolt lapok nyilvántartására. Q_S az SSD-n lévő lapokat, míg Q_D a nem SSD-n lévő lapokat ellenőrzi, mindkét sort LRU algoritmus menedzseli. A korábban említett inflációs érték és a hozzátartozó ötlet segítségével mind a találat (~hit), mind pedig a kilakoltatás O(1) költségűre csökken.

```
1     if p is not cached
2        compare LRU page of Q_S with LRU page of Q_D
3        evict the page q that has the smaller H
4        set L = H(q)
5        bring p into the cache
6     if p is on the SSD
7        H(p) = L + R_S
8        put p to the MRU of Q_S
9     else if p is on HDD
10       H(p) = L + R_D
11       put p to the MRU of Q_D
```

Figure 2: GD2L Algorithm For Reading Page p.

A „2. ábra” az algoritmus leírását mutatja, q legkisebb H értékkel rendelkező lap.

Mikor a GD2L kilakoltatja/eldobja a buffer pool-ból a legkisebb H értékkel rendelkező lapot, L-et H értékűre állítja. Ezt követően, ha az újonnan érkező lap az SSD-n rajta van, akkor a Q_S sor MRU (Most Recently Used) végére kerül (legnagyobb H értékű vég), H-ja pedig $L+R_S$-re állítódik. Ha nincs az SSD-n, akkor a Q_D sor MRU végére kerül, H-ja pedig $L+R_D$ lesz. Mivel L értéke fokozatosan nő, ahogy egyre nagyobb H értékű lapok kerülnek eldobásra, Q_S és Q_D sorok lapjai H értékeik szerint rendezve lesznek. A legkisebb H értékkel rendelkező végeket a sorok LRU végeinek nevezzük. Ezeket az LRU értékeket (Q_S és Q_D) összehasonlítva a GD2L egyértelműen meghatározhatja a buffer pool-ban lévő következő kilakoltatás/eldobás áldozatát (legkisebb H értékkel rendelkező lap), ezt a kérdéses lapot fogjuk eldobni, ha nem lesz több hely a buffer pool-ban egy új lap számára.

3.1 A GD2L MySQL implementációja

Az implementációt, a tanulmány készítői MySQL adatbázis rendszer InnoDB alapértelmezett motorján készítették el. Ez a motor az LRU algoritmus egy változatát használja, a korábbiakban tárgyaltakhoz hasonló módon. A lapokat lista adatszerkezetben tárolja.

Mikor új lapot kell betennünk a betelt bufferbe, egy másikat ki kell dobnunk, lehetőleg olyat amit már régen nem használtunk. Azon lapokat, melyeket kérésre olvasunk be (load on demand) a lista MRU végére helyezzük, míg az előre olvasottakat (prefetched – cachel-és gyorsítására) a középpont köré próbáljuk helyezni (3/8 rész távolságnyira az LRU végtől). Utóbbiakat a lista MRU végére mozgatjuk, ha később olvasás történik rajtuk. (Megvalósul a scan resistance tulajdonság.)

A GD2L implementáláshoz az InnoDB LRU listáját kétfelé vágjuk, és külön kezeljük. A már bevezetett jelölésekkel Q_S az SSD-s cache-elt lapokat, Q_D a HDD cache-elt lapokat tartják nyílván. Új lap érkezésekor, az a megfelelő lista MRU végére vagy középpontjára kerül (load on demand, vagy prefetch). Prefetch esetén az új lap H értékét az aktuálisan a középponton lévő lap költségére (H) állítjuk.

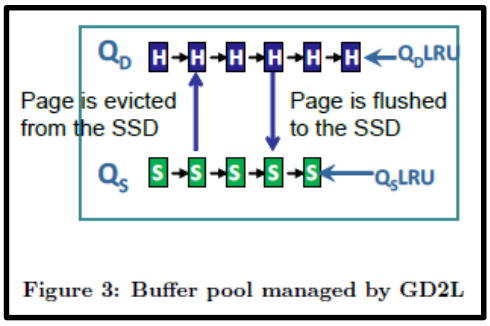


Figure 3: Buffer pool managed by GD2L

Az InnoDB-ben a dirty page-ek keletkezésekor, azok nem kerülnek azonnali visszaírásra a tároló(k)ra. E helyett, laptisztító thread-ek aszinkron módon írják vissza fokozatosan a dirty page-eket. A thread-ek két féle írást hajthatnak végre: csereírás és helyrehozó írás. Az előbbit akkor, mikor a vizsgált dirty page egy eldobandó (evict) lap is egyben (törekszünk arra, hogy a cserére szánt lapok tiszták legyenek mikor ténylegesen cserére kerül sor). Helyrehozó írást a lehető legrégebben módosított lapokra hajtunk végre, hogy biztosítsuk a helyreállíthatóságot és, hogy a helyreállítás ideje ne haladjon meg egy küszöböt. (Az InnoDB előre író logolási helyreállítási protokolt követ.)

Mikor az InnoDB buffer pool-jában a lapok mérete meghalad egy küszöböt a laptisztító szálak az LRU lista végéről elkezdik vizsgálni a lapokat és ha dirty-t találnak, akkor kiírják őket a tár(ak)ra (flush) – ezek a csereírások.

A tanulmány kedvéért változtatott InnoDB-ben ezt a mechanizmust is hozzá kellet simítani a már tárgyalt eljárásokhoz. A GD2L két listát tart nyílván Q_S és Q_D, a laptisztítók ezen listák végén kezdik a vizsgálatot, és a legkisebb H értékkel rendelkező dirty lapot fogják megtisztítani (a legkisebb H értékűt valószínűleg nem fogjuk a közeljövőben használni). Tehát megvizsgálják a két lista végét, és ha mindkettőn találnak dirty page-t akkor a kisebb H értékűt flush-olják, így már lesz hely a buffer-ben az új lapnak. (A helyreállító írások esetében nem kellett változtatni.)

A GD - habár eltérő elérésű költségekkel dolgozik – azt nem kezeli, hogy egy-egy lap elérési költsége megváltozik. A tanulmányban bemutatott rendszerben azonban, mikor egy lapot az SSD-re helyezünk pont ez fog történni. A Q_D listáról egy lap Q_S-re való helyezése akkor következik be, ha a lap dirty volt a buffer-ben és éppen meg akarjuk tisztítani – plussz az SSD menedzser is engedélyezi az áthelyezést -, de a lap másolata nincs az SSD-n. Ha a tisztító thread ezt a műveletet, mint csereírást hajtja végre, akkor a kérdéses lap feltehetően eldobásra is vár egyben (egy jó eldobási alany lesz). Ebben az esetben az áthelyezéskor a lap Q_S lista LRU végére kerül (innen úgy is eldobjuk az elemeket, mikor szükséges lesz). Ha a thread helyreállítási írás keretén belül tisztítja a lapot – és az SSD menedzser „beengedi” a tárhelyére – akkor az LRU lista középpontjára helyezzük a lapot és H értékét az előző középponton lévő lap H értékére állítja. Mivel a Q_S H értékek szerint rendezve van meg is kereshetnénk ennek a kérdéses lapnak a pontos helyét, de a helyreállítási írások nem annyira gyakoriak és a fent bemutatott módszer is megoldja a problémát, ezért erre nincs szükség (sőt, ez a megoldás gyorsabb is, mivel nem kell minden H értéket összehasonlítanunk).

Előfordulhat, hogy az SSD-ről (mondjuk hely hiány miatt) kilakoltatunk/eldobunk egy lapot, ami a buffer pool-ban még benne van, ekkor a Q_S listáról át kell mozgatnunk a megfelelő lapot a Q_D listánkra. Mivel ez az eset is igen ritka, ezért átmozgatáskor egyszerűen csak a Q_D lista középpontjára helyezzük. Pont, mint a helyreállítási írások esetén, a H érté az előző H értéke lesz.

4. A költség tudatos cache-elés hatása

Mivel a GD2L algoritmus a lapokat különböző eszközökre helyezi, a lapok fizikai I/O költségei megváltoznak, attól függően aktuálisan melyik eszközön vannak. A tanulmány e fejezetében a vizsgálat arra irányul, hogy megfelelő terhelés mellett a lapok fizikai elérés mintája (tehát a fizikai elérések száma - statisztika), valamint a kiírási sebesség hogyan változik.

A vizsgálathoz használt terhelés a TCP-C volt (10-es faktorral – gondolom ez „ilyen” durva terhelés – sok tranzakció). A teszt adatbázis kezdeti mérete ~1GB, SSD kezelésre az 5-ös fejezetben részletezett módszert alkalmazták.

Buffer pool méret 200MB, SSD 400MB, futási idő 60 perc.

A futás alatt figyelemmel kísérték, egyes lapok mennyi időt töltenek az SSD-n, továbbá hogy ez idő alatt mekkorák az I/O költségei a lapoknak, illetve hogy ezek az értékek mekkorák mikor a lapok nem az SSD-n vannak.

Közel 2500 lap töltött legalább 20-20 percet az SSD-n és nem az SSD-n (HDD és/vagy buffer pool). Megvizsgálták ezeken a lapokon, a buffer pool „hiány arányát” (miss rate) és a fizikai írások arányát. Egy logikai olvasási kérés egy adott lapon felfogható, mint egy fizikai olvasás mikor a lap hiányzik a buffer pool-ban. A lap hiány aránya (miss rate) a buffer pool-ban tehát, fizikai olvasásoknak felfogott logikai írások százalékaként definiálható.

A „4. ábra” első grafikonja a csak a HDD-n lévő lapok buffer pool hiány arányát mutatja az SSD-n tárolt lapokéval szemben, míg a második ugyanezen helyzet írási arányait szemlélteti.

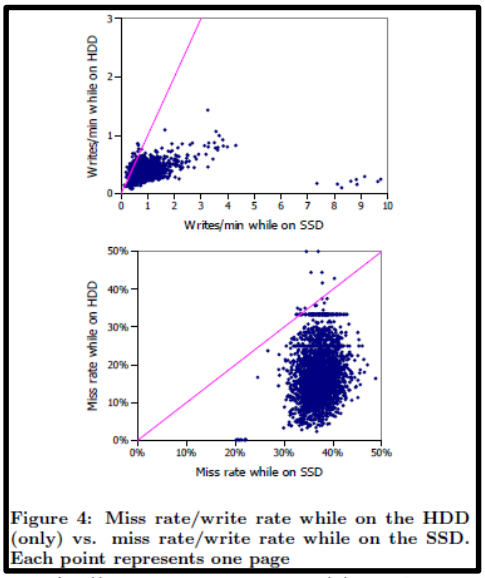


Figure 4: Miss rate/write rate while on the HDD (only) vs. miss rate/write rate while on the SSD. Each point represents one page

Az ábráról leolvasható, hogy mindkét arány magasabb mikor a lapot az SSD-n tárolták. Ez nyílván várható volt, hiszen mikor egy lap átkerül az SSD-re, a buffer pool-ban kilakoltathatóvá válik (jó alannyá válik erre a műveletre) hiszen az SSD-ről gyorsan visszaolvasható. Továbbá, mivel az SSD lapok jobb alanyok a kilakoltatásra/eldobásra (mint a többi lap, ami nem az SSD-n van) a lap tisztítóknak ki kell írniuk ezeket a lapokat a tárhelyre (flush a HDD-re) mielőtt az eldobás megtörténne. Emiatt pedig az SSD-n lévő lapok I/O műveleteinek aránya megnő.

5. SSD kezelés

Az 5-ös fejezet az SSD menedzser működésével foglalkozik majd. Pontosabban azzal, hogy ez a menedzser miként dönti el, hogy befogad-e új, a buffer pool-ból érkező lapokat az SSD-re (vagy nem), illetve az ott lévőket mikor dobja el (admission & replacement policies), továbbá felületesen bemutatja azt a checkpoint alapú technikát, amit az SSD meta adatainak helyreállítását használnak hiba esetén.

Adott lap SSD-re kerülésének két oka lehet, eldobtuk vagy kitisztítottuk a buffer pool-ból. (Eldobjuk a buffer-ból, ha nem történt változtatás rajta és újat akarunk betölteni oda, de nincs szabad hely, tisztításra hasonló helyzetben kerül sor, annyi különbséggel, hogy ekkor a lapon már változtattunk.) Minden esetben engedélyezzük a lapok SSD-re kerülését, amennyiben van

hely, ha nincs, akkor az „invalidáció” következtében jutunk hozzá a megfelelő méretű szabad helyhez az SSD-n.

E folyamat bemutatásához vegyünk egy p tiszta (nem dirty) lapot a buffer pool-ból. Eszközöljünk változtatást a lapon, így az most már dirty, a vitatott p tiszta lapnak volt másolata az SSD-n. Ez a másolat, amely megegyezik a HDD-n tárolt változattal most már nem aktuális, mivel a HDD-re a változtatott adatokat kell majd visszaírnunk, nem pedig a korábbi változatot, ami most is az SSD-n van. Tehát az SSD-n lévő lapot invalidálhatjuk, azaz felszabadíthatjuk és így helyet biztosíthatunk az új lapoknak. Mindazonáltal, ha az SSD-n lévő lap nem identikus a HDD-n tárolt verzióval (azaz az SSD-n már egy kitisztított változat van), akkor az SSD lap nem invalidálható amíg azt ki nem írjuk a merevlemezre. Mivel ez az I/O műveletek megnövekedésével és így a hatékonyság vesztésével járna, a tanulmány rendszere ilyen esetben egyszerűen elkerüli az invalidációt.

Ha nincs szabad hely az SSD-n, akkor a flash lemez menedzserének dönteni kell, hogy egyáltalán a kérdéses lap a buffer pool-ból kerüljön-e az SSD-re, és ha igen, akkor melyik korábban már az SSD-n tárolt laptól szabaduljon meg, az új letárolásának érdekében. Ezt a döntést egy hatékonysági becsléssel hozza meg. A menedzser megbecsüli, hogy az új lap SSD-re kerülése hatékonyabbá teszi-e a működést az SSD-ről eldobandó lap hatékonyságával szemben (I/O művelet költségnek szempontjából vizsgált hatékonyságról beszélünk). Minden esetben olyan lapokat akarunk az SSD-n tárolni, melyek ilyen becslése a leghatékonyabb működést biztosítja. (Tehát az SSD-n azok a lapok lesznek, melyekről sokat olvasunk, melyekre sokat írunk, azaz a leggyakrabban használunk. Ha csökken ez a frekventáció, akkor azt a lapot lecseréljük egy másikra, újabbra, melyen további gyakori munka várható.)

5.1 CAC: Cost-Adjusted Caching~Költséghez igazított Cache-elés

A fejezetben használt rövidítések feloldása az „5. táblázaton” láthatóak:

Szimbólum	Jelentés
r_D, w_D	mért, fizikai olvasás/írás számok nem SSD-n
r_S, w_S	~ az SSD-n
$\widehat{r_D}$, $\widehat{w_D}$	becsült, fizikai olvasások/írások száma nem SSD-n
$\widehat{r_S}$, $\widehat{w_S}$	~ az SSD-n
m_S	buffer cache hiány aránya SSD lapokra (buffer cache miss rate)
m_D	~ nem SSD lapokra
α	„miss rate expansion factor”

5. táblázat

A CAC, hogy segítse az SSD menedzsert a döntésben az említett becslést egy B érték meghatározásával éri el. Egy kérdéses p lap akkor kerül az SSD-re, ha létezik egy p’ lap már az SSD-n, melyre teljesül, hogy B(p’) < B(p). (Tehát p’ hatékonysága kisebb, mint p-jé, vagyis p’ egy megfelelő alany az SSD-ről történő eldobásra, a hely felszabadításra.)

Vegyük p lapra vonatkozó fizikai olvasási r(p) és írási w(p) műveletek számát (valamennyi idővel korábban, mint hogy az SSD-re történő felvételét kérvényeznénk ~ buffer pool-ból SSD-

re való kiírásról való döntés [admission decesion]). Ha ezen információk jó becslő adatoknak minősülnek, akkor megbecsülhető a lap helyváltozásától függő hatékonyság [(1)].

$B(p) = r(p)(R_D-R_S)+w(p)(W_D-W_S)$

A tanulmány rendszerében viszont ezek az információk nem számítanak jó becslő adatoknak vagy egyszerűen nagyon gyengének nevezhetőek, mivel az r(p) és w(p) értékek nem állandóak adott lap esetében. Az értékek változhatnak attól függően, hogy a lap az SSD-re kerül (megnő az I/O-k gyakorisága), vagy onnan eldobjuk (lecsökken).

Egy lap hatékonyságának jó becslésének eléréshez arra lenne szükség, hogy tudjuk, milyen lenne az I/O terhelés, ha tényleg az SSD-re kerülne (a lap). Legyenek $\widehat{r_S}(p)$ és $\widehat{w_S}(p)$ p lap fizikai I/O műveleteinek számai feltételezve, hogy a lap az SSD-n van. Továbbá $\widehat{r_D}(p)$ és $\widehat{w_D}(p)$ ugyanazok az értékek, de olyan lapra, mely nem az SSD-n van. (Ezek becsült értékek, mivel a tényleges értékeket csak az áthelyezés után láthatjuk.)

$$B(p) = (\widehat{r_D}(p)R_D - \widehat{r_S}(p)R_S) + (\widehat{w_D}(p)W_D - \widehat{w_S}(p)W_S)$$

Ezzel a második becslő módszerrel a probléma leszűkül az újonnan bevezetett kalapos tagok becslésére.

$\widehat{r_S}(p)$ becslésére két mérhető olvasási értéket használunk, ezek az $r_S(p)$ és $r_D(p)$ melyek a fizikális olvasási számokat reprezentálják egy adott p lapra, míg az az SSD-n (S index), illetve nem az SSD-n (D index) tartózkodott. A CAC, hogy megbecsülje, mennyi lenne az olvasások száma a p lapra, ha az állandóan az SSD-n lenne (kalapos, S indexű r), a következőt használja:

$$\widehat{r_S}(p) = r_S(p) + \alpha r_D(p)$$

A kifejezésben azon olvasási műveleteket, melyek akkor érték a lapot, míg az nem az SSD-n tartózkodott alfa értékkel szoroztuk, mellyel a két eszköz (SSD és HDD) között lévő különbséget szeretnénk kiküszöbölni. (Mint korábban már utaltunk rá, az SSD-n lévő lapokon több I/O műveletet végezhetünk el, adott idő alatt). Erre az alfa értékre úgy is hivatkozhatunk, mint a „hiány arány kiterjesztési faktor” (~miss rate expansion rate). A többi becsülendő értékek alakja (a p lapra történő hivatkozást ~(p) ezúttal lehagyjuk, mivel egyértelmű):

$$\widehat{r_D} = r_D + \frac{r_S}{\alpha}$$

$$\widehat{w_S} = w_S + \alpha w_D$$

$$\widehat{w_D} = w_D + \frac{w_S}{\alpha}$$

Más becslő módszerekkel szemben a fentebb felállítottak előnye, hogy akkor is működnek, ha a lap melyre alkalmazzuk, nem töltött időt az SSD-n.

A tanulmányban szereplő rendszer nyilván tartja a „referencia számokat” (~reference counts – when falls to zero the block/file/page can be deallocated) azon lapokra, melyek az SSD-n és/vagy a buffer pool-ban vannak. Bizonyos mennyiségű lap számára külön sor adatszerkezetben tároljuk ezeket az értékeket - N_{outq}. A sorba a buffer pool-ból eldobott, de SSD-re nem helyezett, illetve az SSD-ből eldobott lapokról készült statisztikai adatokat helyezünk rekordok formájában. Mikor betelik a sor maximális mérete, az újonnan érkező rekordokat, a sor elején lévő elemekre helyezzük felülírva a legrégebben berakott elemet.

5.2 A „miss rate expansion factor”

Az alfával jelölt elem feladata hogy megbecsülje, miként változik egy adott lap (amelyiket éppen vizsgáljuk) I/O műveleteinek száma, ha a lap az SSD-re kerül.

$$\alpha = \frac{m_S}{m_D}$$

Az m_S tag reprezentálja a logikai olvasások teljes hiány arányát (~overall miss rate) az SSD lapokra (SSD lapok fizikai olvasás száma osztva a logikai olvasások számával). Hasonlóképpen az m_D tag ugyanilyen értéket képvisel, csak nem az SSD-n lévő lapokra.

Például egy 3 értékű alfa azt fejezi ki, hogy az SSD-n tárolt lapok 3-szor magasabb hiány aránnyal (~miss rate) rendelkeznek, mint nem SSD-n tárolt lapjaink.

Ezzel a megközelítéssel azonban az a gond, hogy azt feltételezi, hogy minden lapnak azonos lesz ez kiterjesztési faktora (~expansion factor). Ez azonban nem igaz, mivel egy adattábla több lapból állhat és nem biztos, hogy a tábla minden lapjára szükségünk lesz egy-egy tranzakció – és a velejáró I/O műveletek - végrehajtásakor. Erre példa a „6-os ábra”.

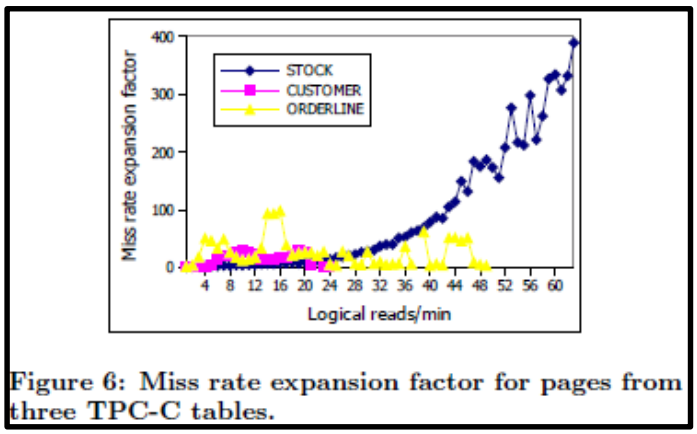


Figure 6: Miss rate expansion factor for pages from three TPC-C tables.

Az ábrán TPC-C STOCK, CUSTOMER és ORDERLINE táblák lapjainak alfa értékeiket (miss rate expansion factor) ábrázoltuk táblánként és logikai olvasásonként csoportosítva.

Látszik tehát, hogy az alfa értékeket lapok adott csoportjaihoz rendelve éri meg kezelni. Az ábránál elkészített csoportokat fogjuk figyelembe venni a későbbiekben is, mivel ez egy jó csoportosítás lesz, és ezen csoportok alfa értékeit követjük. Tehát a lapok csoportosítása: egy adott adatbázis objektumhoz való tartozás (pl.:tábla) és logikai olvasási arány (logical read rate) alapján történik. A hiány arányt (miss rate) egyenlő méretű altartományokra osztjuk. Ezek alapján a csoportokat úgy készítjük, hogy vesszük az azonos adatbázis objektumhoz tartozó és az azonos logikai olvasási hiány arány altartományba eső lapokat. Így ha az altartományokat 1 logikai olvasás/ perc megszorítással definiáljuk, akkor, ha egy tábla max logikai olvasási aránya 1000 akkor 1000 csoport keletkezik.

A lap csoportokat g-vel jelölve:

$$\alpha(g) = \frac{m_S(g)}{m_D(g)}$$

m_S és m_D (g)-s alakja az eredeti jelölések kiterjesztése az adott g lap csoportra. Minden lap fizikai és logikai olvasási számát követjük, szintúgy a csoportok hiány arányát. Míg az előbbieket minden egyes olvasási kérelemkor frissítjük, a csoportokra vonatkozó hiány arányt csak akkor, ha a buffer pool-ból eldobtunk egy lapot, kiírtunk egy dirty page-t vagy az SSD-ről dobtunk el egy lapot (ami a csoportba tartozik nyilván). A csoportok meghatározásából adódik, hogy bizonyos lapok időről-időre kieshetnek az adott csoportból, amibe korábban besoroltuk őket. Ilyenkor a régi csoport értékeiből kivonjuk a távozó lap értékeit, majd az új csoportjának értékeihez hozzáadjuk.

Az is előfordulhat, hogy a csoportra szorított m_S és m_D értékek bizonyos esetben definiálatlanok maradnak. Például egy csoportra, ami olyan lapokat tömörít melyeket sosem dobtunk el a buffer pool-ból: $m_D(g) = 0$. Egy másik csoport esetén, mely lapjai sosem kerültek az SSD-re $m_S(g) = 1$.

A megoldás hátránya az alfa értékek meghatározására szükséges statisztikák begyűjtése és tárolása. A csoportok számától a statisztikák mérete erősen függ, a csoportok számát, pedig mint láthattuk az altartományok beállítása befolyásolja. Ha nem állítunk be ilyen altartományokat, minden táblához egy ilyen fog tartozni, ha felosztjuk kisebb altartományokra az alaptartományt akkor pontosabb alfa értékeket kaphatunk. A tanulmányban használt alfa értékek meghatározásához használt csoportok statisztikái elenyésző méretet foglaltak el, így a problémától eltekinthetünk. (Ebből következően az altartomány felosztás is megfelelő volt, mivel jó alfa értékeket kaptunk, és a statisztikák sem foglaltak jelentős méretet.)

5.3 Szekvenciális I/O

A HDD-k jóval hatékonyabbak szekvenciális olvasásokra, mint véletlenszerűekre, pontosan emiatt mikor a CAC megbecsüli eg lap SSD-re helyezésének előnyét, hatékonyságát, akkor csak a véletlenszerű olvasásokat veszi figyelembe (r_S és r_D csak véletlen olvasásokat mér). Ehhez az kell, hogy az SSD menedzser besorolja az olvasási kérelmeket a két olvasási típus valamelyikébe (szekvenciális/véletlen). A tanulmányban erre a célra azt az elvet alkalmazzák, hogy egy olvasást szekvenciálisnak tekintenek, ha a lap előre olvasás (prefetch) keretein belül kerül beolvasásra, más különben véletlenszerű olvasásról beszélhetünk.

5.4 Hibakezelés

Az SSD-n tárolt adatok mindig frissebbek, mint a HDD-n lévő másolatok. Ha változtatást hajtunk végre bizonyos lapokon, akkor az előbb az SSD-re kerül, majd később a HDD-re. Amennyiben a rendszer működése közben hiba lép fel, biztosítanunk kell azt a helyzetet, hogy helyreálláskor beazonosíthassuk az SSD-n tárolt elemeket.

A tanulmányban bemutatott rendszer feltételezi, hogy az SSD-n lévő lapok egy hiba esetén túlélik a helyreállást (nem történik adatveszteség), és a helyreállást követően az itt tárolt lapokat innen olvassa be. Ebben a megközelítésben a nehézséget az SSD menedzser belső memóriájában tárolt hash map (melyben az SSD-n lévő lapokat tároljuk – azonosítóit) helyreállítása jelenti. A tanulmány készítői a problémát checkpoint-okkal és az SSD-re történő írások log-olásával oldották meg. A hash map helyreállítását az SSD leolvasásával a teljes

rendszer helyreállásával egyidőben végzi a menedzser. Az SSD leolvasásakor (scan) az SSD-n tárolt lapok fejléceit keresi, és ez által építi a hash map-ot. A fajlécekkel beazonosíthatóvá válnak a korábban itt tárolt lapok. Ez a megoldás azonban növelheti a helyreállítási időt, mivel egy-egy ilyen leolvasás az SSD méretétől függően több percbe is kerülhet (pl.:32GB ~ 3 perc). Ezt az időt minimalizálandó, a CAC periódikusan készíti a checkpoint-okat a hash map-ről, továbbá k db alacsony prioritású (ritkábban használt) lapot megjelöl, mint alkalmas alanyt a kilakoltatásra/eldobásra (eviction zone – k a mérete). Míg el nem érkezik az idő egy újabb checkpoint elkészítéséhez, addig csak ebből a k db lapból dob el a rendszer. Hibakor, a CAC a legutóbbi checkpoint-ból inicializálja a hash map-et és megnézi, milyen lapok vannak a k méretű zónában, frissítve a hash map-et az információ alapján, ha szükséges. A checkpoint készítés egy periódusát az eldobási zóna (eviction zone) hossza adja - k. Mikor a zónában lévő összes lap eldobásra került, egy checkpoint készül, majd a zóna elemei újra kiválasztásra kerülnek. Ebből adódik, hogy kisebb zóna méretek gyakoribb hash map checkpoint-okat eredményeznek (ezáltal biztonságosabbá válik a helyreállás), de ezzel megnő a helyreállás ideje is. (vica versa).

6. Értékelés

A kiértékelés célja főként az, hogy az olvasó bepillantást nyerjen a szerzők által javasolt megoldási módszer működésébe, ami egyfajta kombinációja a már jól ismert GD2L és CAC algoritmusoknak. Előbbi a buffer pool-, még utóbbi az SSD-vel kapcsolatos műveletekért felelős. Ezen belül két fő kérdés merülhet fel bennünk, ami bővebb magyarázatot kíván. Először foglalkozni kell a GD2L algoritmus hatékonyságával, összehasonlítva ezt az úgynevezett ”nem költség figyelmes” buffer menedzsmenttel. Valamint a GD2L buffer pool szabályozásánál annak a lépésnek a fontosságával, hogy egy előrelátó SSD kezelőt (pl.: CAC) használjunk, ami felismeri az SSD és a HDD közötti lapváltásnál történő lap hozzáférési minta változását.

A fejezet második fontos feladata az, hogy könnyedén össze tudjuk hasonlítani az előbb említett algoritmus (GD2L, CAC-t használva) teljesítményét a többi, nem közelmúltban kifejlesztett és megismert adatbázis rendszerekben történő SSD kezelő technikákkal szemben.

Ezen kérdések megválaszolására és az eredmények szemléltetésére a cikk szerzői implementáltak több különböző már ismert algoritmust a MySQL InnoDB tároló rendszerébe. Az adatbázis kezelő rendszer buffer pool-ra vonatkozóan két választási lehetőség is adott, az InnoDB által nyújtott eredeti szabályhalmaz (LRU) és a szerzők saját (GL2D) algoritmusa. SSD kezelői oldalról pedig adott a CAC algoritmus, valamint ehhez hasonló 3 alternatíva, ezek: CC, LRU2 és az MV-FIFO, amiket a következő pontokban részletesebben is tárgyalunk.

CC: Költségalapú, mint a CAC, viszont nem előrelátó. Vagyis, a CAC - al ellentétben nem próbálja meg kitalálni a lap I/O mintájának változását annak SSD és HDD közötti mozgatásánál. Az 5. fejezetben megismert egyenlőségek közül az elsőt használja arra, hogy megbecsülje az lap SSD - re helyezésének hasznosságát, valamint szükség esetén áthelyezze / kidobja az SSD - ről a számára legkevésbé hasznos lapot. A CC megközelítése az SSD - re való lapelhelyezés előnyének becslésére megegyezik a TAC által használttal, habár a TAC statisztikáit nem csak lap-szerte készíti és ellenőrzi. A másik nagy különbség, hogy a CC a

lapokat azoknak tisztításakor / buffer poolból való törlésekor lépteti az SSD - re, a TAC pedig olvasáskor. Megemlítendő még, hogy a TAC az SSD - t write-through cache - ként (”keresztülíró gyorsítótár”), a CC pedig a CAC-hoz hasonlóan write-back cache - ként (”visszaíró gyorsítótár”) kezeli.

LRU2: Az algoritmus nem költségalapú és nem is előrelátó. A lapokat azoknak tisztításakor / buffer poolból való törlésekor lépteti az SSD - re, amit write-back cache-ként kezel. Ennek a speciális LRU algoritmusnak az SSD - ről való laptörlés logikája meggyezik a legjobb teljesítményű Lazy Cleaning (LC) technikával, amit J. Do és társai publikáltak ”Turbocharging dbms buffer pool using ssds” címmel. Ezen LRU algoritmus érvénytelenítési eljárása azonban kis mértékben különbözik, a példaként is tekintett LC algoritmusétól úgy, hogy ebben az esetben az LRU2 csak akkor érvénytelenít egy SSD lapot, ha az teljes mértékben megegyezik a HDD-n lévő lappal.

MV-FIFO: A legnagyobb változás a többi ismert technikával szemben, hogy ezen algoritmus úgy kezeli / érzékeli az SSD - t, mint lapok egy FIFO sorát, ez biztosítja azt, hogy minden az SSD - re való minden írás szekvenciálisan – így biztosítottan gyorsan – történik. Ez az algoritmus és a kifejlesztett technika fő előnye. A lapokat azoknak tisztításakor / buffer poolból való törlésekor lépteti az SSD - re. Ha e folyamatok közben azt érzékeli, hogy a feldolgozandó lap már létezik az SSD - n, úgy a korábban elkészült verziót érvényteleníti. Az algoritmus nem költségalapú és nem is előrelátó.

Bármelyik buffer pool technika vegyíthető valamennyi SSD kezelővel, így a továbbiakban, ha egy X buffer pool kezelőhöz egy Y SSD kezelő algoritmust használunk, azt X+Y - al jelöljük.

6.1 Metodológia

A használt adatbázis rendszer a MySQL 5.1.45-ös verzió InnoDB motorral, amit úgy módosítottak a szerzők, hogy képes legyen befogadni a saját buffer- és SSD kezelő rendszereiket. A szerverben találhatóbb főbb hardverek, szoftverek és szerepük:

- 6db 2.5GHz Intel Xeon core
- 4Gb memória
- Ubuntu 10.10 Linux 2.6.35-22-generic kernel verzióval.
- 1 db 500GB RPM SCSI hard disk, ami az összes rendszerhez szükséges szoftvert tartalmazza, beleértve a MySQL-t és a teszt adatbázist.
- 1 db 500GB RPM SCSI hard disk, ami a tranzakciós naplófájlokat őrzi
- 32GB Intel X25-E SATA SSD – az adatbázis SSD cache egy fájlként lett implementálva.

Minden tesztelés TPC-C terhelést használt. A konkrét említetteken kívül a tesztelésnél és kísérleti próbálkozásoknál több féle másodlagos metrikák is kipróbálásra kerültek, például különböző eszközök felhasználása az I/O műveletek számontartására, mindezt operációs rendszer és adatbázis szinten is támogatva. A próbák hossza általában 4 és 7 óra között volt, amit általában az az idő határozott meg, amennyi alatt sikerült egy sikeres és stabil kapcsolatot kiépíteni a tesztelési környezet elemei között. Minden egyes futás után a DBMS - t újraindították, ezzel biztosítva az üres buffer poolt, valamint az adatbázist lecserélték egy

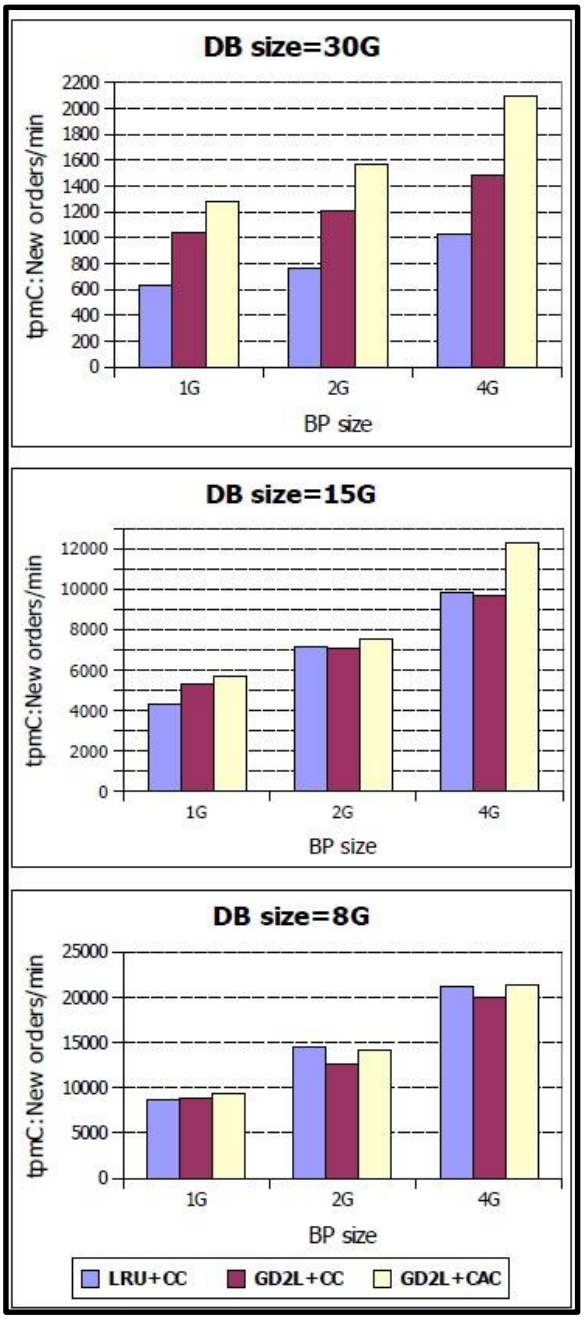
olyanra, ami csak a szükséges alapadatokat tartalmazza. Ahogyan J. Do és társai, e cikk szerzői is három forgatókönyv szerint tervezték be a kísérleti futásokat:

- Az adatbázis lényegesen nagyobb, mint az SSD cache mérete
- Az adatbázis valamennyivel nagyobb, mint az SSD cache mérete
- Az adatbázis kisebb, mint az SSD cache mérete.

Azoknál a próbafutásoknál, amikben szerepel CAC vagy CC algoritmus, ott az outqueue belépések maximális számának értéke az SSD cache - be férő lapok számára lett beállítva. Minden CAC-t tartalmazó tesztelésnél az SSD cache méretének 10%-a lett használva eldobási zónának.

6.2 Paraméter költség kalibráció

A 3. és 5. fejezetben bemutatottak szerint a GD2L és CAC is az eszközökön végrehajtott olvasási és írási műveletekre támaszkodnak. Az SSD egyik legfontosabb tulajdonsága az I/O asszimetria, vagyis az a tulajdonság, hogy gyorsabban tud adatokat olvasni, mint írni. Ez azért történhet meg, mert az adatok írásakor olykor egy bizonyos törlési késleltetést kell várni, az olvasás pedig szabadon, mindenféle kötelezően előírt várási időtől mentesen történhet. Az említett költségeket lehet számolni és nyomon követni, jelölésük: Rs *és* Ws, előbbi az olvasásra, utóbbi az írásra. Ezek mérésére a cikk írói futtattak egy TPC-C terhelési tesztet *diskstats* nevezetű Linux programmal párhuzamosan, ami a teszt alatt tudta felmérni és összegezni az I/O műveleti költségeket. InnoDB segítségével pedig pontosan meg lehet mondani, hogy ezalatt a teszt alatt mennyi írási és mennyi olvasási kérés érkezett. A pontos mérések ellenére azonban a *diskstats* sajnos a kiszolgálási idejét nem méri a műveleteknek, viszont a kéréseket több profilban többszöri futással, az eredményekből pontosan ki lehet számolni ezeket az eddig nem ismert paramétereket, majd az összehasonlításokat a már pontos értékeken végezni. A méréseket elvégezték HDD - n és SSD – n is, az eredmények a normalizálások után: R_S=1, R_D=70, W_S=1, W_D=50 (W: írás, R: olvasás, S: SSD, H: HDD).



6.3 GD2L és CAC analízis

A GD2L és CAC teljesítményének megértéséhez a cikk szerzői az alábbi három kombinációban futtatták az algoritmusokat, teljesen monitorozva: LRU+CC, GD2L+CC, GD2L+CAC. Az első kettő összehasonlításánál a költségfeledékeny algoritmusról (LRU) költségfigyelmes algoritmusra (GD2L) való váltásnak próbálták a hatását felmérni. Az utóbbi kettő hasonlításánál pedig egy nem előrelátó SSD menedzsert váltottak fel egy előrelátóval. A szöveg melletti ábrán ezen méréseknek látható az eredménye vizuális kimutatás segítségével, amit a cikk további részei tárgyalnak részletesebben.

6.3.1 GD2L vs LRU

Az ábrából egyértelműen látható, hogy a GD2L hatékonysága egyenes arányosságban van az adatbázis méretével. Vagyis, amíg az adatbázis mérete kicsi, a két algoritmus teljesítménye nagyjából megegyezik, viszont a méret növelésével az LRU hatékonysága csökken, míg az GD2L stabilan teljesít (40-75% - al jobban, mint az LRU). A két táblázat mutatja az eszközök felhasználását, a buffer poolt, a tévesztési arányt és a normalizált I/O időt, 15 illetve 30GB – os adatbázis esetén.

Alg & BP size (GB)	HDD util (%)	HDD I/O (ms)	SSD util (%)	SSD I/O (ms)	total I/O (ms)	BP miss rate (%)
LRU+CC						
1G	93	88.6	12	11.1	99.7	6.6
2G	93	72.8	8	6.0	78.8	4.4
4G	94	55.1	6	3.3	58.4	2.4
GD2L+CC						
1G	92	53.1	21	12.1	65.2	8.8
2G	90	44.3	20	9.7	54.0	7.4
4G	90	36.3	14	5.8	42.1	4.7
GD2L+CAC						
1G	85	39.6	19	8.8	48.4	7.4
2G	83	31.8	20	7.8	39.6	6.3
4G	82	23.5	20	5.8	29.3	4.8

Device Utilizations, Buffer Pool Miss Rate, and Normalized I/O Time (DB size=30GB)
I/O is reported as ms. per New Order transaction.

Alg & BP size (GB)	HDD util (%)	HDD I/O (ms)	SSD util (%)	SSD I/O (ms)	total I/O (ms)	BP miss rate (%)
LRU+CC						
1G	79	11.1	38	5.4	16.5	4.2
2G	68	5.7	47	4.0	9.7	2.7
4G	73	4.4	43	2.6	7.0	1.3
GD2L+CC						
1G	21	2.5	68	8.0	10.4	6.1
2G	18	1.5	62	5.3	6.8	3.7
4G	14	0.9	61	3.8	4.7	2.3
GD2L+CAC						
1G	30	3.2	73	7.8	11.0	5.7
2G	21	1.6	78	6.2	7.8	4.0
4G	48	2.3	60	2.9	5.3	2.0

Device Utilizations, Buffer Pool Miss Rate, and Normalized I/O Time (DB size=15GB)
I/O is reported as ms. per New Order transaction.

A 30GB - os esetben az adatokból jól lehet látni, hogy a GD2L a hihetetlenül gyors olvasási sebessége miatt tudott az LRU - n felülkerekedni, annak ellenére, hogy nagyságrendekben nagyobb hibaaránnyal dolgozhat. Ez azért lehetséges, mert a GD2L+CC esetben az olvasás nagy része az SSD - ről történt, ami kevesebb időt vett igénybe, mint ha HDD - ről történt volna, így ha még hibás művelet is hajtódott végre, még mindig hamarabb tudta az egész feladatot újrakezdeni és hibamentesen megoldani, mint ha egy biztonságos, ám lassú olvasás történt volna. Tehát levonva a tanulságot, a nagyobb adatbázisnál megéri jobban SSD - t az olvasási műveletekre használni.

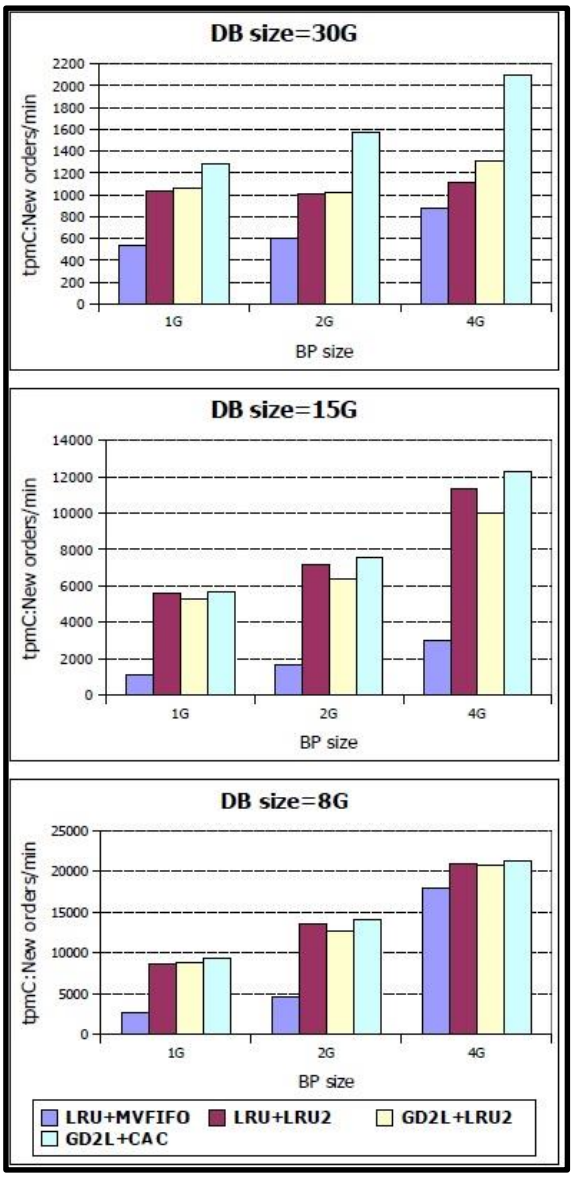
A 15GB - os adatbázis esetében is látszik, hogy a GD2L tranzakciónként kevesebb költségből tudta az I/O műveleteket elvégezni, szintén az SSD - n való tárolás lehetősége miatt. Azonban ez az előny itt nem annyira számottevő, mint egy nagyméretű adatbázisnál. Ahogy a diagramból is látszik, itt a két algoritmus közel megegyező eredményeket produkált. Ez azért tapasztalható, mert a tesztek futtatásánál az SSD elkezdett itt telítődni és ilyenkor előjönnek a már megismert hátrányai. A 30DB - os esetben azért nem kezdett el telítődni az SSD, mivel az egész adatbázis legfontosabb, gyakrabban használt részei megtalálhatóak voltak, így könnyű volt az adatlekérdezés útkeresése.

Az eredményekből arra lehet következtetni, hogy egy nagyobb SSD beszerzése után a TPC-C tesztet megismételve, az eredmények úgy alakultak volna, mint a 30GB - os esetben.

A 8GB - os esetben teljesen egyformák voltak az eredmények, ugyanis a teljes adatbázis elfért az SSD - n, így az egyik sor (Q_D) mindig majdnem üres tud maradni.

6.3.1 CAC vs CC

A diagramból jól látszik, hogy GD2L+CAC kombináció lényegesen több és jobb értékeket mutat, mint a GD2L+CC nagyméretű adatbázis esetén. Együtt a GD2L és a CAC nagyon nagy teljesítménynövekedést mutattak be a 30GB - os adatbázis esetében. A 15GB - os esetben ez a teljesítmény növekedés nem volt annyira jelentős, a 8GB - os esetben pedig teljes mértékben kimutathatatlan volt, amiből arra lehet logikusan következtetni, hogy az algoritmusok összhangjának sikeressége az adatbázis nagyméretűségén múlik. A táblázatokból látható, hogy mind a nagyobb, mint a közepes méretű adatbázisnál sikerült a GD2L+CAC használatával jelentősebben csökkenteni az I/O költségeket. Ha jól megvizsgáljuk az eredményeket az is kideríthető, hogy ugyanolyan tévesztési arány mellett tudott dolgozni a CAC - al és a CC - vel is a GD2L algoritmus. Ebből az következtethető, hogy a költségcsökkenés valóban jelentős, a CAC algoritmus által meghozott döntések a lap SSD - n való használatáról jobbnak bizonyulnak vetélytársánál. Hogy ezeket a döntéseket jobban meg tudjuk érteni, vagyis rájönni, hogy a CAC esetében az I/O költséget hogyan sikerült ilyen jelentős mértékben lecsökkenteni, a rendszer működéséről szerzett log fájlok analizálása volt a feladat. Az egyik legérdekesebb különbség az, ahogyan eldöntik, hogy melyik lapot helyezzék el az SSD - n, szabad hely keletkezésekor (például amikor egy lapot törölt a buffer menedzser). Ebben az esetben a CC úgy dönt, hogy a lapot törli az SSD - ről, a buffer poolból pedig folyamatosan felveszi, illetve törli, használatától függően, mivel ha ez elég nagy, okkal került oda. Így próbálja nagyméretű, sűrűn használt fájlon keresztül csökkenteni az I/O költséget úgy, hogy nem rendel el változtatásokat, mert látja, hogy ami eddig jó és kis költségű volt, ezután is megfelelő. Ezzel szemben a CAC mohó módon kitörli az SSD - ről viszont a buffer poolban tartja, és abban reménykedik, hogy a következő lap, ami hely felszabadulása miatt az SSD - re íródik sokkal hasznosabb lesz az algoritmus számára, nagyobb mérete vagy



fontossága miatt több költséget tud megspórolni. A kisebb adatbázisoknál pedig teljesen az elvártakhoz hűen, teljesen megegyeztek a futási statisztikái, az I / O művelet költségei a két algoritmusnak. Erre tudatosan készülni lehetett, ugyanis ebben az esetben már az SSD teljes (vagy majdnem teljes) mértékben tudta tartalmazni az egész adatbázist, így nem kellett kilogikáznia az algoritmusoknak, hogy melyik részét próbálják meg ott tárolni.

6.4 LRU2 és MV-FIFO összehasonlítása

Ebben a részben összehasonlításra kerül a GD2L+CAC nem rég megjelent két SSD menedzser technikákkal, név szerint a Lazy Cleaning (LC) - el és a FaCE - el. Pontosabban az összehasonlítás az LRU2 és MV-FIFO algoritmusokhoz történik, amik megegyeznek az LC és FaCE kombinációval és InnoDB implementációjuk lehetővé teszi a pontosabb összehasonlítást. LRU2 és MV-FIFO kombinálásra kerül az InnoDB alapértelmezett buffer menedzserével, ami az alábbi kombinált algoritmusokat eredményezi: LRU+LRU2, LRU+MV-FIFO. Ezen felül a tesztelés alá került a GD2L+LRU2 is. Az előző ponthoz hasonlóan a diagram mutatja mind a 4 algoritmus TPC-C - n elért eredményeit mind a három különböző adatbázisrendszer-méretre, amin a tesztelés zajlott. Összességében azt találtuk ismét felfedezni, hogy a GD2L+CAC algoritmusok kombinációja jelentősen felülmúlja a többi páros által elért eredményeket a nagyméretű (30GB) adatbázisrendszer esetében. A legszorosabb versenytárs a GD2L+LRU2 kombinációja volt, csak a nagyméretű buffer pool menedzselésénél maradt alul. A közepes (15DB) adatbázis méreténél a GD2L+CAC csak alig volt gyorsabb, mint az LRU+LRU2, a legkisebb adatbázisméretnél (8GB) pedig nem volt észrevehető a különbség az algoritmusok futási ideje között.

Alg & BP size (GB)	HDD util (%)	HDD I/O (ms)	SSD util (%)	SSD I/O (ms)	total I/O (ms)	BP miss rate (%)
GD2L+CAC						
1G	85	39.6	19	8.8	48.4	7.4
2G	83	31.8	20	7.8	39.6	6.3
4G	82	23.5	20	5.8	29.3	4.8
LRU+LRU2						
1G	85	49.4	15	8.6	58.0	6.7
2G	87	50.3	12	6.7	57.0	4.4
4G	90	48.5	9	4.7	53.2	2.4
GD2L+LRU2						
1G	73	41.1	43	24.6	65.8	10.7
2G	79	46.5	32	18.9	65.3	8.8
4G	79	34.4	32	13.8	48.2	7.6
LRU+FIFO						
1G	91	101.3	8	9.2	110.5	6.6
2G	92	92.3	6	5.8	98.1	4.4
4G	92	62.9	5	3.7	66.6	2.5

Device Utilizations, Buffer Pool Miss Rate, and Normalized I/O Time (DB size=30GB) I/O is reported as ms. per New Order transaction.

LRU+MV-FIFO teljesített az összes algoritmus közül a legrosszabbul, minden egyes egymástól független tesztre, minden paraméterrel, feltehetően a HDD miatt. Az algoritmus célja, hogy növelje az SSD hatékonyságát szekvenciális írásokkal. Igaz ezt sikeresen tudja alkalmazni, de még így is a teszt rendszerben az SSD viszonylag kevésbé kihasznált, tehát az MV-FIFO optimalizációja nem növeli az egész algoritmus TPC-C teljesítményét. Ami viszont érdekes az az, hogy az LRU+MV-FIFO még a legkisebb (8GB) teszt adatbázisnál is elég gyengén szerepel és csakis a HDD teljesítményére hagyatkozhat. Ennek két oka van, az első az, hogy az SSD - n felszabadult tárhelyet felhasználó része az MV-FIFO algoritmusnak gyengébb, mint az LRU2 és CAC azonos részei. A második ok, hogy az SSD - ről kidobott / felszabadított helyeire történő lemezírást újrahasznosítja az MV-FIFO. A fenti táblázat hasonlóképp foglalja össze az adatokat, mint a GD2L és LRU összehasonlításánál. A GD2L+LRU2 rosszabbul teljesített, mint a GD2L+CAC a legnagyobb adatbázisméretnél, mivel tranzakciónként magasabb I/O költségekkel számol, ráadásul ebben az esetben a HDD - re aránytalanul több munka jutott, ami szintén megnehezített, lassította a munkát. Bár a táblázat nem mutatja, de GD2L+CAC tranzakciónként kevesebb olvasást hajtott végre a HDD - n és többet az SSD - n, mint az LRU+LRU2. Ez részben a CAC SSD elhelyező algoritmus részének köszönhető, részben pedig

a GD2L azon részének, ami az SSD - ről dobja el a már nem használt részeket. Megfigyelhető, hogy az LRU2 SSD - n való tévesztési rációja növekedett a buffer pool méretének növelésével. Továbbá, hogy az LRU+LRU2 több írást végez a HDD - n, mivel nem rendelkeznek annyira jól kiforrott hely felszabadítási stratégiával az SSD - n, mint a GD2L+CAC.

A GD2L+LRU páros rosszabb teljesítményt nyújtott, mint az LRU+LRU2 és a GD2L+CAC a legtöbb esetben. A teljesítmény táblázat alapján látható, hogy a GD2L+LRU2 megnövekedett I/O költségekkel tud működni, ami az SSD - n való I/O költségek növekedése miatt következett be. A CAC - al és CC - vel ellentétben az LRU2 az összes dirty page - et az SSD - re tölti. Mikor ezek az SSD - re kerülnek, a GD2L kitörli őket a buffer poolból, majd gyorsan újratölti azt. Ennek eredményeképp a GD2L+LRU hajlamos megtartani a hot page - eket az SSD - n és folyamatosan törli az buffer poolból, majd visszarakja. A költségfigyelmes szabályai miatt ezek a lapok több I/O költséget okoznak az SSD - n, mint a HDD - n.

A közepes (15GB) adatbázisméretnél a GD2L+CAC előnye eltűnik. Ebben a rendszerben mindkét algoritmus tranzakciónkénti I/O költsége azonos. A GD2L+CAC műveleteinek nagyobb részét próbálja az SSD - n végezni, mint az LRU+LRU2, viszont a különbség nem volt a kísérletekben számottevő jelentőségű. A legkisebb (8GB) adatbázisméretnél egyáltalán nem volt semmilyen különbség az eredmények között.

6.5 Az eldobási zóna hatásai

Hogy lehessen látni az eldobási zóna hatásait, GD2L+CAC algoritmus volt használva tesztfuttatásokra, különböző eldobási zóna méretekkel. A kísérletekben az adatbázis méret 1GB, a buffer pool méret 100MB és az SSD cache méret 400MB volt. A teszteken *k* az SSD méretének különböző százalékaira volt beállítva (1%, 2%, 5%, 10%). Ezek a tesztek azt mutatták, hogy *k* értékének ebben az intervallumban nincs hatása a TPC-C eredményekre. Az InnoDB implementációban a lapazonosító 8 byte és minden lap mérete 16KB. Tehát a 400MB - os SSD hash map 10 oldalra fér fel. A szerzők meghatározták a rátát, ami szerint az SSD hash map feltöltődött, és azt tapasztalták, hogy még *k*=1% - ra is, a hash map legmagasabb ellenőrzési ideje bármelyik ismert három SSD menedzser (CAC, CC, LRU2) közül is kevesebb, mint 3 másodperc. Így ellenőrzőpont kirakása a hash map felső részére elhanyagolható.

7. Kapcsolódó munkák

Az ötlet, hogy sűrűbben és ritkábban használt adatok tárolási helye más legyen, nem újdonság. Erre a problémára megvalósult adattárolási technika a HSM (Hierarchical Storage Management). Ennek alapvetően a lényege az volt, hogy egyazon tárhelyet használta magas és alacsony műveleti költségű anyagok tárolására úgy, hogy a nagyméretű magas I/O költségű adattárolón alakított ki cache - t a gyors műveletek végrehajtására. Az SSD - vel új korszak kezdődött, ugyanis egy teljesen külön tárolót lehet (kell!) használni cache - ként, megsokszorozva a teljesítményt. Ezzel a régi szalagos tárolókat teljesen kiszorította a piacról, a HDD - k vették át a helyüket, és a legújabb, legjobb teljesítményű (és legnehezebben hozzájuthatóbb…) tárolóvá vált az SSD.

Néhány más kutatás arra irányult, hogy hogyan lehetne részben lecserélni a merevlemezt SSD - re az adatbázisrendszerben.
Például Koltsidas és szerzőtársai azt bizonyították, hogy véletlenszerű olvasások az SSD - ről tízszer gyorsabbak, mint véletlenszerű írások a HDD - re, valamint még véletlenszerű írások az SSD - re tízszer lassabbak, mint véletlenszerű írások a HDD - re. Ezért olyan algoritmust dolgoztak ki, amik az olvasásigényes lapokat az SSD - re, míg az írásigényesebbeket a HDD - re helyezik.
Canim és szerzőtársai egy objektum-elhelyezési tanácsadót mutattak be az adatbázisrendszerhez. Futási időben vett statisztikát az I/O műveletekről a buffer menedzser tárol, ezt feldolgozva az implementált tanácsadó segít az adatbázis adminisztrátornak meghatározni az SSD megfelelő méretét, és hogy erre a limitált felhasználási tárterületre milyen (táblák, indexek, …) és melyik objektumokat helyezze.
Ozmen és szerzőtársai bemutattak egy adatbázis elrendezés optimalizálót, ami terhelést próbálja stabilizálni és elkerülni az interferenciát az objektumok között. Ez a módszer képes heterogén tárolási konfigurációkra - amik például SSD - t is tartalmaznak - is elrendezéseket generálni és optimalizálni.
A flash memória még mindig több helyen próbálja alsóbb kategóriás cach - ként megállni a helyét. Például FlashCache, a Facebook egyik terméke, próbál a valódi, elsődleges memória és a felhasznált lemezek között egy LRU/FIFO szabályrendszert használó cache - ként működni. Vagy a FlashStore, ami képes SSD - t write-back cache - ként használni a RAM és a HDD között. Az adatokat kulcs-érték páronként dolgozza fel, a párokat egy speciális naplózási struktúrában tárolja a flash memórián, hogy növelje az írási teljesítményt.
Canim és szerzőtársai később szintén sikert aratva próbálkoztak egy olyan megoldással, ami az SSD - t egy második szintű write-through cache - ként használja. Itt a leggyakrabban olvasott lapokat, amiket a futási időben szerzett I/O statisztikákból lehet látni, helyezik el az SSD - n. Ha ez a lap az SSD - n van, de írás szükséges, azonnal áthelyezi a HDD - re.

Az említett példákon kívül még sok hasonló munka volt, ami megpróbálta a közös, SSD menedzserhez kapcsolódó problémákat megoldani. Ezeket a munkák a 9. pontban, a referenciáknál részletesen fel vannak sorolva.

E cikk írói nagyrészt a referenciák áttanulmányozásával, újra feldolgozásával jutottak olyan ötletekre, következtetésekre, ami alapján megszülethetett a GD2L és CAC algoritmusok. A GD2L buffer pool menedzser a M. S. Manasse és társszerzői 1990 - ben bemutatott GreedyDual algoritmusának egy korlátozott verziója. Az SSD menedzser CAC pedig az eddig megismert költségfigyelmes algoritmusok kiegészítése, ami más technika segítségét használja fel a lapokkal és felszabadult helyekkel kapcsolatos döntések meghozatalára.

8. Összegzés

A cikk két új algoritmus (GD2L és CAC) mutatott be, hogy könnyedén tudjuk menedzselni a buffer poolt és az SSD - t az adatbázis rendszerünkben. Mindkét algoritmus költségalapú és a cél a nagy terhelés mellett a teljes elérési és kiszolgálási idő minimalizálása. Az algoritmusok implementálása a MySQL InnoDB - ben történt meg, futási analizálásuk pedig TPC-C segítségével. Mindjét algoritmust már létező, ezekhez hasonló algoritmusokhoz való

összehasonlítással elemezték. Az adatbázisok méretének növekedésével egyértelműen látható volt, hogy ezek az algoritmusok teljesítményben jobb eredményt produkáltak, mint a hozzájuk hasonló, már létező megoldások. Persze nyilván ez nem minden esetben ilyen kiemelkedő, az eredmények erősen függnek a konfigurációtól, és főként az adatbázis, a HDD és az SSD méretének egyensúlyától. A tesztekben a teljesítményt olykor jelentősen csökkentette a HDD képességének felső korlátai. Azokban az esetekben, ahol a HDD nem megfelelő, vagy az SSD mérete alacsony, ott más algoritmusok (például a FaCE) jobban használhatóak a probléma megoldására.

9. Referenciák

M. Canim, G. A. Mihaila, B. Bhattacharjee, K. A. Ross, and C. A. Lang.
An object placement advisor for db2 using solid state storage.
Proc. VLDB Endow., 2:1318-1329, August 2009.

M. Canim, G. A. Mihaila, B. Bhattacharjee, K. A. Ross, and C. A. Lang.
Ssd bufferpool extensions for database systems.
Proc. VLDB Endow., 3:1435-1446, September 2010.

P. Cao and S. Irani. Cost-aware www proxy caching algorithms.
In Proc. *USENIX Symp. on Internet Technologies and Systems*, pages 193-206, 1997.

B. Debnath, S. Sengupta, and J. Li. Flashstore:
high throughput persistent key-value store. Proc. VLDB Endow., 3:1414-425, September 2010.

J. Do, D. Zhang, J. M. Patel, D. J. DeWitt, J. F. Naughton, and A. Halverson.
Turbocharging dbms buffer pool using ssds.
In Proc. SIGMOD Int'l Conf. on Management of Data, pages 1113-1124, 2011.

Facebook. Facebook: FlashCache, 2012.
http://assets.en.oreilly.com/1/event/45/Flashcache%20Presentation.pdf.

B. C. Forney, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau.
Storage-aware caching: Revisiting caching for heterogeneous storage systems.
In Proc. the 1st USENIX FAST, pages 61-74, 2002.

G. Graefe. *The five-minute rule 20 years later: and how fash memory changes the rules.*
Queue, 6:40-52, July 2008.

J. Gray and B. Fitzgerald. *Flash disk opportunity for server applications*.
Queue, 6(4):18-23, July 2008.

W.-H. Kang, S.-W. Lee, and B. Moon.
Flash-based extended cache for higher throughput and faster recovery.
Proc. VLDB Endow., 5(11):1615-1626, July 2012.

I. Koltsidas and S. D. Viglas. *Flashing up the storage layer.*
Proc. VLDB Endow., 1:514-525, August 2008.

A. Leventhal. *Flash storage memory*. Commun. ACM, 51:47-51, July 2008.

T. Luo, R. Lee, M. Mesnier, F. Chen, and X. Zhang.
hstorage-db: *heterogeneity-aware data management to exploit the full capability of hybrid storage systems.* Proc. VLDB Endow., 5(10):1076{1087, June 2012.

Y. Lv, B. Cui, B. He, and X. Chen.
Operation-aware buffer management in flash-based systems.
In Proc. ACM SIGMOD Int'l Conf. on Management of data, pages 13-24, 2011.

M. S. Manasse, L. A. McGeoch, and D. D. Sleator.
Competitive algorithms for server problems. J. Algorithms, 11:208-230, May 1990.

O. Ozmen, K. Salem, J. Schindler, and S. Daniel.
Workload-aware storage layout for database systems.
In Proc. ACM SIGMOD Int'l Conf. on Management of data, pages 939-950, 2010.

The TPC-C Benchmark. http://www.tpc.org/tpcc/.

T. M. Wong and J. Wilkes. *My cache or yours? making storage more exclusive.*
In Proc USENIX ATC, pages 161-175, 2002.

N. Young. *The k-server dual and loose competitiveness for paging.*
Algorithmica, 11:525-541, 1994.