Budapesti Műszaki és Gazdaságtudományi Egyetem
Villamosmérnöki és Informatikai Kar

Hadoop alapú megoldások vizsgálata gyakori elemhalmazok meghatározására

Illés János

2012

KONZULENS

Kovács Ferenc

Automatizálási és Alkalmazott Informatikai Tanszék

Tartalomjegyzék

1 Bevezetés

Adatbányászati algoritmusok segítségével nem magától értendő, hasznos információkat nyerhetünk ki különösen nagy méretű, esetenként zajos adatbázisokból. Ilyen információ az adatbázisban lévő asszociációs szabályok is. Ezek felfedezése alapvető kérdés és folyamatosan kutatott terület az adatbányászatban.

Napjainkban az adatbázisok mérete egyre nagyobb méreteket ölt. Ekkora méretekben egy-processzoros rendszerek erőforrásai nem mindig elegendőek ahhoz, hogy hatékonyan kutassunk összefüggéseket bennük.

Az elosztott rendszerek fejlődésével természetes igény adódik arra, hogy asszociációs szabályok kereséséhez felhasználjuk a *grid* és *felhő* rendszerekben rendelkezésre álló számítási kapacitást.

Asszociációs szabályok kinyerésére egy klasszikus algoritmus az Apriori. Az Apriori algoritmus tranzakciókat (például vásárlói kosarak tartalmát) tartalmazó adatbázisokból képes kinyerni a gyakran előforduló elemeket és ezek segítségével asszociációs szabályokat előállítani.

Az Apriori algoritmus tervezésekor nem volt szempont az elosztott működés, így módosítás nélkül nem használható több processzoros környezetben. Munkám során implementálom az Apriori algoritmus egy olyan változatát amely elosztott környezetben is képes működni, lehetőleg minél jobban kihasználva az elosztottságból származó előnyöket. Ehhez az Apache Hadoop szoftver keretrendszert használom fel. A Hadoop széleskörűen használt, nyílt forrású szoftver, amely a *Google*nél kifejlesztett MapReduce programozási modell legelterjedtebb nyílt forráskódú implementációja.

Célom az algoritmus viselkedésének vizsgálata különböző bemenő paraméterek és felhasznált processzorok mennyiségének függvényében.

2 Elosztott számítási modellek

Elosztott számítások olyan feladatok elvégzése amiket több számítógépből álló, hálózaton keresztül kommunikáló elosztott rendszerek végeznek. Az elosztott rendszerek programozására több megközelítés is született. Közvetlenül elosztott környezetben programozva választhatunk elterjedt elosztott memóriás vagy üzenetküldéses párhuzamos algoritmusok közül [9]. Ezek a megoldások azonban komoly tervezést és bonyolult megvalósításokat eredményeznek.

Egy elosztott számítások elvégzésének megkönnyítésére magasabb szintű modellek születtek, úgy mint az elosztott adatbázisok [7] vagy a MapReduce [2]. Ezek a rendszerek teljesítménycsökkenés nélkül képesek elrejteni az elosztott rendszerek programozásának implementációs részleteit [8].

Dolgozatomban a MapReduce modellt vizsgálom meg közelebbről.

2.1 MapReduce

A MapReduce [2] egy nagy adathalmazok feldolgozására és létrehozására tervezett programozási modell. A programozási modellt – és egy azonos nevű implementációt – a Google mérnökei fejlesztették ki, azzal a céllal, hogy általánosítsák az általuk összegyűjtött nagyméretű adatokon történő munkavégzést. A nagy adatokon végzett feladatok jellemzően önmagukban egyszerűek (például log állományokból bizonyos kulcsszavak kikeresése) ám az adathalmaz mérete lassítja, vagy egyenesen lehetetlenné teszi az egygépes feldolgozás lehetőségét. Több számítógépet felhasználva szükséges a feldolgozást végző programot az összes, a folyamatban részt vevő gépre eljuttatni, a folyamat párhuzamos lefutását felügyelni, majd feladat végeztével az eredményeket összegezni.

A párhuzamosítás hatékonysága, a hibakezelés, valamint az adatok és kódok gépek közti egyenletes elosztása és mozgatása bonyolult mérnöki feladat, természetes igény volt, hogy ne legyen szükséges ezen problémák megoldásával újra és újra foglalkozni minden egyes elosztott feladat megoldása közben.

A MapReduce modell célja, hogy az elosztottsággal járó komplexitást elrejtse a programozó elől, így ő csak a megoldandó feladatra koncentrálhasson. A modell alapgondolata az elosztott számítást két részre bontó absztrakció bevezetése. A funkcionális programozásban már régóta ismert és használt map és reduce (egyes programozási nyelvekben a reduce helyett fold néven található) műveleteket emeli át elosztott környezetbe. Funkcionális programozásban a map művelet egy függvényt hajt végre egy lista minden egyes elemén, a reduce művelet pedig rekurzív adatszerkezetek bejárására használható, legtöbb esetben adatokat aggregál.

A MapReduce környezet felhasználójának pusztán egy Map és egy Reduce függvényt kell írnia, hogy számítását végrehajthassa elosztott környezetben. A számítás – és így egyben a Map függvény – bemenete a kulcs-érték párokat tartalmazó adathalmaz. A Map függvény feladata, hogy ezekből a bementi párokból átmeneti kulcs-érték párokat állítson elő amiket a MapReduce rendszer a Reduce függvénynek továbbít. A Reduce függvény a Map kimenetét kulcsok szerint csoportosítva kapja meg majd ebből tetszőleges kimenetet állít elő. A Reduce függvények kombinált kimenete egyben az egész MapReduce számítás kimenete is.

A MapReduce rendszer a Map és a Reduce függvényt is párhuzamosan, több gépre elosztva futtatja. A Map függvényt futtató folyamatokat mappereknek, míg a Reduce függvényt futtató folyamatokat reducereknek hívjuk. A mapperek és reducerek száma a bemenet méretétől és a rendszerben rendelkezésre álló számítógépek számától függ. A MapReduce környezethez hozzá tartozik egy elosztott fájlrendszer is. Adatfeldolgozás közben a mapperek az elosztott adatokon dolgoznak, úgy, hogy a lehető legkevesebb adatot kellejen mozgatni az elosztott fájlrendszert alkotó gépek között. A nagy méretű bemeneti fájlok gyakran több darabban, több különböző gépen tárolódnak az elosztott fájlrendszerben. Ilyenkor a mapper folyamat az adott darabot tartalmazó gépen kerül futtatásra.

A bementi fájlt egészében egy mapper folyamatnak sem szükséges látnia így számuk a bemenet méretével együtt növelhető. A reducer folyamatok is csupán annyi

garanciát kapnak, hogy az egy átmeneti kulcshoz tartozó értékeket egy reducer folyamat fogja megkapni. A sem a mapper sem a reducer folyamatok nem kommunikálnak egymással, az esetlegesen kieső folyamatok könnyedén pótolhatók. Látszik, hogy a modell hibatűrése nagy, mivel egy leálló mapper vagy reducer könnyen újraindítható. Ez a modell jól skálázódik, viszont speciálisan tervezett algoritmusokra van szükség a hatékony adatfeldolgozáshoz [6].

MapReduce modellt használva nem kell foglalkoznunk az elosztott keretrendszer működéséhez tartozó implementációs kérdésekkel.

3 Hadoop

A *Google*nél kifejlesztett MapReduce nevű szoftver a külvilág számára ugyan nem elérhető, de a témában készült, szabadon elérhető publikációk alapján többen többféle programozási nyelven is implementáltak MapReduce programozási modellen alapuló környezeteket és keretrendszereket. Ezek közül messze legsikeresebb a Hadoop névre hallgató, *Java* programozási nyelven készült implementáció. A Hadoop az Apache alapítvány felügyelete alá tartozó, sikeres nyílt forrású projekt ami *de facto* MapReduce implementációvá vált mind ipari mind kutatási célokra [1].

3.1 A Hadoop felépítése

Egy Hadoop klaszter egy mester gépből és tetszőleges számú szolga gépből áll. A Hadoop szoftvercsomag több részből tevődik össze, a klaszterben lévő gépeken több folyamat is fut, ezek felelősek az adatok tárolásáért, illetve a különböző munkák futtatásáért. A folyamatok egymással TCP/IP felett, SSH kapcsolaton keresztül kommunikálnak.

A Hadoop klaszter működéséhez elengedhetetlen a HDFS fájlrendszer. A HDFS egy elosztott, virtuális fájlrendszer, aminek mérete a klaszterben részt vevő gépek tárolási kapacitásának összege. A Hadoop környezetben futó programok számára a HDFS fájlrendszer egy nagy egészként jelenik meg. A fájlrendszerbe „feltöltött”

fájlokat a Hadoop blokkokra darabolja, majd ezek a blokkok a klaszterben lévő gépeken redundánsan kerülnek szétosztásra. A Hadoop szoftvercsomagban biztosított segédprogramok segítségével a fájlrendszerbe tetszőleges fájlt elhelyezhetünk, illetve „letölthetünk" onnan helyi adathordozóra. A HDFS fájlrendszer nem érthető el az operációs rendszer beépített eszközeivel és a rajta lévő fájlok manipulálásra a Hadoop Java API-t biztosít.

A HDFS működéséért a mester gépen futó *NameNode* és a szolga gépeken futó *DataNode* folyamatok a felelősek. A NameNode feladata a fájlrendszerben tárolt blokkok helyének nyilvántartása míg a DataNode azért felel, hogy a kapott blokkokat a helyi fájlrendszeren eltárolja és a fizikai írás és olvasás műveleteket végrehajtsa.

A MapReduce munkák futtatásáért a mester számítógépen futó *JobTracker* és a szolga gépeken futó *TaskTracker* folyamatok felelősek. Egy munka indításakor a felhasználó által írt programot a JobTracker kapja meg. A JobTracker munkák indítása előtt egyeztet NameNode folyamattal, és a MapReduce programot azokra a gépre küldi tovább, amelyek legközelebb helyezkednek el az adathoz amin dolgozni fognak.

3.2 Hadoop Streaming

A Hadoop keretrendszer egy hasznos szolgáltatása a Streaming munkák futtatása. Segítségével tetszőleges futtatható állományt használhatunk a mapper és reducer folyamatnak. Ha ezt a futtatási módot választjuk, akkor a Hadoop két futtatható fájlt vár tőlünk. Az egyik a mapper a másik a reducer szerepét fogja ellátni. A munka elindításakor a bemeneti fájlt (fájlokat) szeletekre bontva megkapja a mapper folyamat a szabványos bemenetén (standard input) a kimenetként generált átmeneti kulcs/érték párokat pedig a szabványos kimenetre (standard output) kell írnia. Miután a mapper folyamatok végeztek, a kimenetüket a Hadoop kulcs szerint sorba rendezi, majd a rendezett átmeneti kulcs/érték párokat a reducerek számának megfelelően szeleteli és a reducer folyamatok szabványos bemenetére

küldi. A munka kimenete a reducer folyamatok szabványos kimenetei lesznek, összefűzve.

Alapértelmezetten a Hadoop Streaming szöveges protokollt használ a lépések közti kommunikációra. A bementi fájl egyszerű szöveges fájl, amiben a kulcsok és értékek között egyetlen tabulátor karakter található. Ez az elválasztó karakter egyébként tetszőlegesen változtathatjuk egy parancssori kapcsoló segítségével.

Streaming segítségével tetszőleges szkript-nyelveken is írhatunk MapReduce munkákat, a szkriptnek annyi feladata van, hogy szabványos bemeneten várja az adatokat és szabványos kimenetre írja ki az eredményeket. Ez nagyban meggyorsítja a fejlesztést és könnyűvé teszi a kísérletezést, feltételezve, hogy a bementi adatállományunk szöveges. Szerencsére sok gyakori probléma esetében természetesen adódik a szöveges reprezentáció vagy az adatbázisban lévő rekordok könnyen sorosíthatóak szöveges formátummá. Az átalakítás automatizálására a Hadoop olyan eszközöket is biztosít amelyekkel akár menet közben is képes szöveges bemenet előállítására bináris adatokból.

Hadoop Streaming környezetben tetszőleges nyelven készíthetünk programokat, a dokumentációban a leggyakrabban példaként használt programozási nyelv a Python. A mérések során én is ezt a nyelvet választottam a MapReduce programok elkészítésére, mivel rendelkeztem a nyelvben szerzett korábbi tapasztalatokkal és az átlátható szintaxis, illetve a kiterjedt osztálykönyvtár nagyban segíti a gyors, kísérletezés-orientált fejlesztést.

4 Asszociációs szabályok

Az asszociációs szabályok segítenek kapcsolatokat felfedezni látszólag nem kapcsolódó elemek között különböző adatbázisokban [11]. Felfedezésük alapvető kérdés az adatbányászatban.

Asszociációs szabályokat tranzakciókat tartalmazó $T = \{t_1, t_2, ..., t_n\}$ adatbázisokban keresünk, ahol a tranzakciók elemhalmazokból állnak.

Gyakorlatban az adatbázisban lévő tranzakciók jellemzően együtt vásárolt termékek halmaza, más szóval a vásárlói kosár.

Asszociációs szabályoknak $I_1 \rightarrow I_2$ szabályokat nevezünk, ahol I_1 és I_2 is elemhalmazok és $I_1 \cap I_2 = \emptyset$ teljesül. Egy ilyen szabály jelentése, hogy ha egy tranzakcióban (kosárban) megtaláljuk I_1-et akkor valószínűleg megtaláljuk I_2-t is. A szabály bekövetkezésének valószínűségét a szabály biztossága (*confidence*) határozza meg. Egy szabály támogatottsága (*support*) megadja a szabály előfordulásának gyakoriságát, azt, hogy összesen hány tranzakcióban fordulnak elő együtt a szabályban lévő elemhalmazok. A gyakorlatban a minél nagyobb bizonyosságú szabályokat igazán hasznosak, az ilyen szabályok támogatottsága is nagy. Egy szabály biztosságát egyszerűen kiszámolhatjuk a támogatásából az 1. képleten látható módon [3].

$$conf = \frac{sup(I_1 \cup I_2)}{sup(I_1)}$$

1. képlet: biztosság megállapítása támogatásból

Például a $\{burgonya, paprika\} \rightarrow \{kolbász\}$ szabály jelentése, hogy aki burgonyát és paprikát vásárol, az valószínűleg kolbászt is venni fog, *biztosság* szerinti valószínűséggel.

Látható, hogy az asszociációs szabályok ismerete a gyakorlati életben nagyon hasznos. Vásárlói kosarak elemzésével szabályok állíthatóak fel a vásárlási szokásokra, és előrejelezéseket tehet jövőbeni vásárlói viselkedésre. Ez az információ pedig fontos szerepet játszik többek közt áruház elrendezések kialakításában és termék-katalógusok összeállításában.

4.1 Gyakori elemhalmazok

Az asszociációs szabályok felfedezéséhez gyakori elemhalmazokat kell találnunk az adatbázisban. A szabályokat elő tudjuk állítani gyakori elemhalmazokból a minimális biztosság (*minconf*) bemenő paraméter függvényében. Gyakori elemhal-

maznak azokat az halmazokat nevezzük, amelyek támogatottsága meghaladja egy meghatározott minimális támogatás (*minsup*) értékét. A támogatottságot aszerint vizsgáljuk, hogy az elemhalmaz az adatbázis hány rekordjában (hány vásárlói kosárban) fordul elő. A kosárban lévő elemek sorrendje nem számít, ahogyan az sem, hogy milyen sorrendben követik egymást a rekordok az adatbázisban. Nem foglalkozunk azzal az esettel sem, ha egy kosárban egy elemből több is előfordul, számunkra csak az az információ érdekes, hogy az adott elem szerepel az adott kosárban.

Gyakori elemhalmazok keresésére több algoritmus is ismert, ezeknek az algoritmusoknak bemenő paramétere a kívánt minimális támogatás értéke. Ilyen algoritmus az Apriori és az FP-fa építő algoritmusok. Az Apriori az egyik legismertebb gyakori elemhalmaz kereső algoritmus, az FP-fát használó algoritmusok memóriaigénye lényegesen nagyobb [5].

5 Apriori algoritmus

Az Apriori egy tradicionális algoritmus gyakori elemhalmazok megkeresésére. Az algoritmust kifejezetten tranzakciókat tartalmazó adatbázisokon való működésre tervezték. Olyan adatbázisokra ahol a rekordok vásárlói kosarakat jelölnek, a rekord tartalma pedig a vásárló által kosárba helyezett termékek listája.

Az algoritmus bemenete egy tranzakciókat tároló D adatbázis és a minimális támogatás értéke (*minsup*). A feladat D adatbázisban megtalálni olyan, tovább már nem bővíthető, elemhalmazokat amelyek elérik a felhasználó által meghatározott minimum támogatást. A támogatás egy számérték, hogy hány tranzakció tartalmazza az adott elemhalmazt. Azokat az elemhalmazokat amelyek teljesítik a minimális támogatást, *nagy elemhalmaz*oknak nevezzük. Minden egyéb elemhalmazt kis elemhalmaznak hívunk. A tranzakciókat egyedi azonosító, *TID* jelöli.

Az algoritmus fő ötlete a jelöltek generálása az apriori tulajdonságot használva. Eszerint minden gyakori elemhalmaz összes részhalmaza is gyakori elemhalmaz

kell legyen. Ezzel nagymértékben vágható a gyakori elemhalmazoknak keresési tere, így az algoritmus memória igénye jobban kézben tartható. Ennek az az ára, hogy az nagy elemhalmazok előállításához az adatbázis többszöri végigolvasására van szükség [3].

5.1 Az algoritmus működése

Az algoritmus működését lépésről lépésre az alábbiak szerint tudjuk leírni

1. gyakori 1-méretű elemhalmazok megkeresése (L_1)
2. 2-hosszú jelöltek generálása a gyakori elemekből ($L_1 \Rightarrow C_2$)
3. Jelölt elemek vizsgálata, nem gyakori jelöltek kiszórása ($C_2 \Rightarrow L_2$)
4. A k-hosszú gyakori elemekből k+1 hosszúságú jelöltek előállítása ($L_k \Rightarrow C_{k+1}$)
5. Jelölt elemek vizsgálata, nem gyakori jelöltek kiszórása ($C_k \Rightarrow L_k$)
6. A 4. és 5. pontok ismétlése amíg nem készíthető nagyobb elemhalmaz (L_{k+1} üres)

Az első lépésben végigiterálunk az adatbázison és megszámoljuk az összes elem előfordulását, ezzel megállapítjuk a nagy 1-elemhalmazokat. A nagy elemhalmazok listájába tehát azok az elemek kerülnek amelyek a minimum támogatásként megadottnál nagyobb számú tranzakcióban fordulnak elő. Az egy elemet tartalmazó nagy elemhalmazokat együttesen L_1-nek nevezzük, a k. lépés nagy elemhalmazait pedig L_k-nak.

Innentől az algoritmus egy két fázisra bontható lépést ismétel. Az első fázisban létrehozzuk L_{k-1} alapján a jelölteket tartalmazó C_k listát. A második fázisban kiszórjuk a jelöltek közül azokat amelyek nem nagy elemhalmazok. Az így kapott L_k listával folytatódik az algoritmus. A futás akkor fejeződik be amikor nem sikerül bővíteni az L_k listát, más szavakkal, amikor a második fázisban minden elem kiszóródik.

5.2 Jelöltek generálása

A C_k jelölt elemek generálása úgy történik, hogy az előző kör nagy elemhalmazaiból összefűzés segítségével előállítjuk az összes egyel nagyobb méretű halmazt. A k. lépésben tehát a nagy elemhalmazok $k-1$ elemből állnak. Az új, k elemből álló jelölt elemhalmazok készítéséhez az összes olyan elemhalmazpárt amely csak egy elemben tér el egymástól bővítjük az eltérő elemmel. Formálisan leírva:

insert into C_k
select $\mathrm{p.item}_1, p.item_2, ..., p.item_{k-1}, q.item_{k-1}$
from $\mathrm{L}_{k-1}\ p, L_{k-1}\ q$
where $\mathrm{p.item}_1 = q.item_1, ..., p.item_{k-2} = q.item_{k-2}, p.item_{k-1} < q.item_{k-1}$

A jelöltek kiszórásakor a C_k listából eltávolítjuk azokat az elemhalmazokat amelyeknek van olyan $k-1$ méretű részhalmaza ami nem része L_{k-1}-nek.

Az apriori-algoritmus lényegi ötlete, hogy a jelölteket mindig az előző lépésben létrehozott nagy elemhalmazokból állítja elő. Ez a folyamat jóval kevesebb jelölt elemet állít elő mint a korábbi, jelölt-generálást alkalmazó algoritmusok [3].

6 Gyakori elemhalmazok keresése Hadoop környezetben

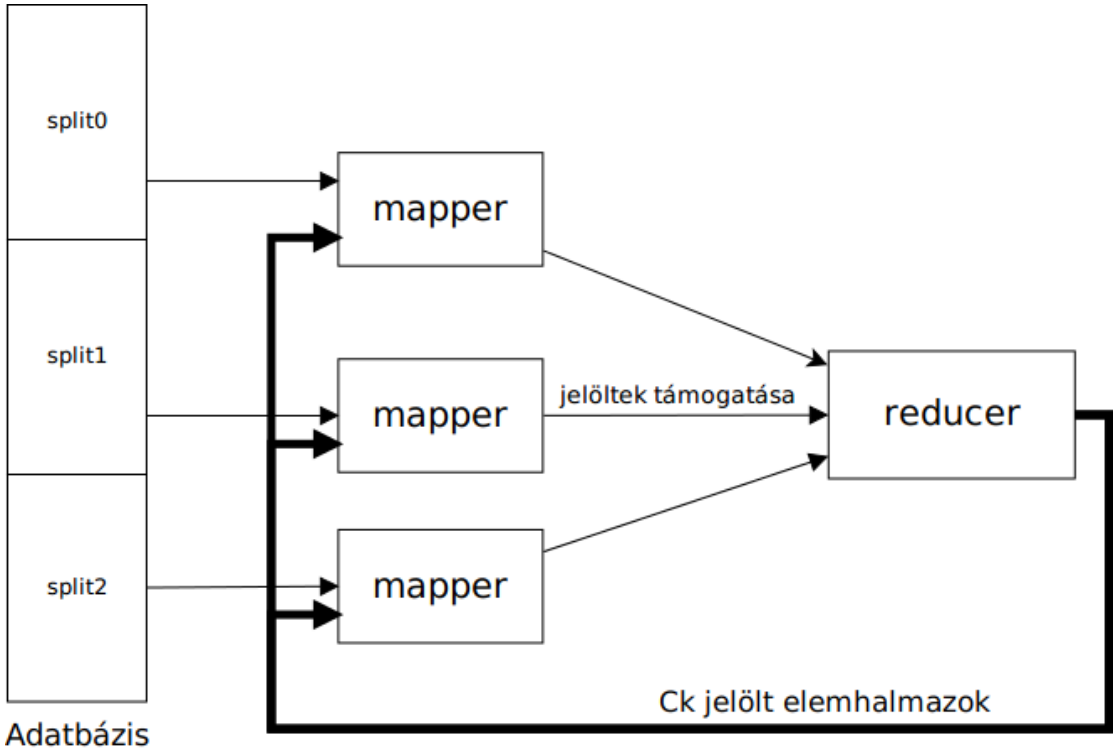
6.1 Az Apriori algoritmus átalakítása Hadoop környezetre

Az Apriori algoritmus megalkotásakor soros futás volt az elsődleges cél, nincs felkészítve elosztott, párhuzamos működésre. Ahhoz, hogy MapReduce munkaként futtassam az algoritmust, szükség volt annak átalakítására.

A Hadoop környezetben történő futáshoz két részre, egy map és egy reduce szakaszra, kell bontani az algoritmust. MapReduce környezetben működő Apriori algoritmuson korábban is végeztek méréseket [10]. Ebben publikációban említett felbontásból indultam ki a saját implementációmnál, mivel a jelölt generálást MapReduce folyamaton kívül oldották meg.

Mivel az algoritmus futása rekurzív és a mapper folyamatok nem látják az egész adatbázist, csupán szeleteit, ezért a MapReduce munkát többször kell végrehajtani

egymás után, hogy megkaphassuk a végeredménynek várt gyakori elemhalmazokat. A MapReduce környezetben történő működés szemléletesen az 1. ábrán látható.



1. ábra: Apriori algoritmus működése MapReduce környezetben

A map szakaszban minden mapper folyamat megkapja a bemeneti adatbázis egy szeletét, majd összeszámolja a jelölt elemek támogatottságát az adott szeletben. Minden mapper folyamat hozzáfér az összes jelölthöz.

Az elsőnek futó map folyamat abban különbözik a többitől, hogy még nem rendelkezik a jelölt elemekkel, amelyek támogatását meg kell számolnia az adatbázisban. Ezért ez a folyamat az összes 1-méretű elemhalmaz támogatását számolja meg.

mapper:

```
if first_run:
    foreach transaction in database_split:
        foreach item in transaction:
            print "item <TAB> 1"
else:
    candidate_tree = read_from_file(candidates)
    foreach transaction in database_split:
        foreach subset in transaction:
            if subset is in candidate_tree:
                increase_count

    print serialize(candidate_tree)
```

reducer:

```
foreach itemset in input.key:
    foreach support for input.value:
        sum(support)
    if support > MINSUP
        largesets.append(itemset)

next_candidates = apriori_gen(largesets)
if next_candidates is empty:
   print "RESULT"
   print largesets
else:
   print next_candidates
```

1. algoritmus: Apriori algoritmus MapReduce környezetben

A reducer folyamat minden iterációban ugyanazt a műveletet végzi el. Bemenetként k hosszú jelölt elemeket és támogatásuk értékét kapja. Mivel a mapper folyamatok az adatbázisnak csak egy-egy szeletén számolják meg a jelölt elemhalmazok támogatását ezért a reducer folyamatnak először összegeznie kell az azonos elemhalmazhoz tartozó értékeket. Miután az összegzés megtörtént a reducer kiszórja azokat a jelölteket amik nem teljesítik a minimális támogatottság követelményét, ezzel előállítva az L_k listát. A reducer feladata ebben még nem merül ki, az L_k listát felhasználva az Apriori algoritmus ismertetésénél tárgyalt módon létrehozza a C_{k+1} jelölt halmazt. A jelöltek halmazát felsorolva a futása véget ér,

amennyiben készültek új jelöltek akkor indulhat a következő map lépés ami megszámolja a C_{k+1} jelöltek támogatottságát.

Látható, hogy a reducer folyamatnak az C_{k+1} lista előállításához szüksége van arra, hogy az egész L_k a rendelkezésére álljon. Éppen ezért a helyes működés érdekében egy futási ciklusban csak egy reducer folyamat futhat.

A mapper indulásakor a jelölteket tartalmazó fájl egészét be kell olvassa és a teljes futás alatt a memóriában tartania, hogy a tranzakciókon végigiterálva meg tudja számolni az egyes jelöltek támogatottságát. A jelöltek száma, különösen az első néhány lefutás során még igen magas, ezért olyan adatszerkezetben kell tárolni őket, ami az elemek gyors hozzáférhetőségét biztosítja.

A jelölteket egy fában tárolom aminek mélysége megegyezik a jelöltek aktuális k hosszával. A csomópontok a jelölt elemek vannak a levelekben pedig az adott levélhez vezető elemekből álló jelölt támogatása.

6.2 Hadoopon futó Apriori algoritmus módosítása

Az Apriori algoritmus első ciklusának futása során nagyon sok jelölt generálódik. Az idő nagy része az első lépésekben a sok, kis elemszámú jelölt generálásával és támogatásuk megszámolásával telik el [12]. Megpróbáltam tehát az első két lépés számításigényes feladatait hatékonyabban, egy MapReduce lépésben elvégezni.

Az előző részben vázolt implementáció a C_3 jelölt elemeket a második lefutási ciklust záró reduce függvény állítja elő az $Adatbázis \Rightarrow L_1 \Rightarrow C_2 \Rightarrow L_2 \Rightarrow C_3$ lépéseken keresztül. Ezen a folyamaton rövidítek az alábbi módszerrel: Az első map függvényben az adatbázisban lévő tranzakciók összes két elemű részhalmazát és a hozzá tartozó támogatást előállítom, így kapok egy C_2-nél nagyobb, de azt tartalmazó halmazrendszert. A reduce függvényben L_2 előállítása triviális és gyors, ki kell dobni azokat az halmazokat amiknek a támogatása nem éri el a minimum szintet. Az előző algoritmus reduce függvénye pontosan ugyanezt teszi, így módo-

sítás nélkül felhasználható. A reducer kimenete az L_2 halmazrendszerből a már megismert módszerrel elkészíthető C_3. Tehát egy map és egy reduce függvény segítségével előállt pontosan az a kimenet amihez az előző algoritmusnak két map és két reduce függvényre, és ezzel az adatbázis két teljes átolvasására volt szüksége.

A 2. algoritmusban látható a mapper függvény módosított része. Működése csak a legelső futáskor módosul. A mapper és a reducer a többi lépésben pontosan úgy viselkedik mint az előző pontban leírt változat.

```
if first_run:
    c2_matrix = new array[1000][1000]
    foreach transaction in database_split:
        foreach item1, item2 in every_2_combinations(transaction):
            c2_matrix[item1][item2] += 1
    print serialize(c2_matrix)
```

2. algoritmus: módosított Apriori mapper részlete

A map függvényben az összes két elemből álló halmaz támogatásának megszámolásához kihasználtam, hogy a tranzakcióban szereplő egyedi elemek száma véges. Az általam használt, 7.2. fejezetben tárgyalt, tesztadatokban például 1000 különböző elem fordul elő. Ennyi elempár előfordulását hatékonyan tárolom egy 1000×1000 méretű háromszög mátrixban (két dimenziós tömbben).

6.3 Gyakori elemhalmazok felkutatása a keresési tér csökkentésével

Az Apriori algoritmus 6.2 pontban vázolt módosításával az első lépésben az összes két elemből álló halmazt is előállítjuk és megszámoljuk, köztük rengeteg olyat, amit a módosítatlan algoritmus sosem generálna le. Kihasználva viszont az elosztott környezet biztosította nyers erőt, ezzel a lépéssel megspóroltam egy teljes MapReduce futási ciklust. A lépések csökkentésére hasonló trükköket is bevezethetünk. A legkézenfekvőbb, és egyben legprimitívebb megoldás, hogy az adatbázis-szeletek minden tranzakciójából állítsuk elő az összes lehetséges kombinációt, majd ezeket összegezzük és szórjuk ki a minimumot nem elérőket. Ezzel egy

MapReduce lépésben megoldottuk a gyakori elemhalmazok keresését. Sajnos azonban ez a megoldás a gyakorlatban közel sem alkalmazható, a generált elemhalmazok mérete exponenciálisan növekszik így már viszonylag kis adatbázisok esetén is számítási kapacitás és memóriakorlátokba ütközünk.

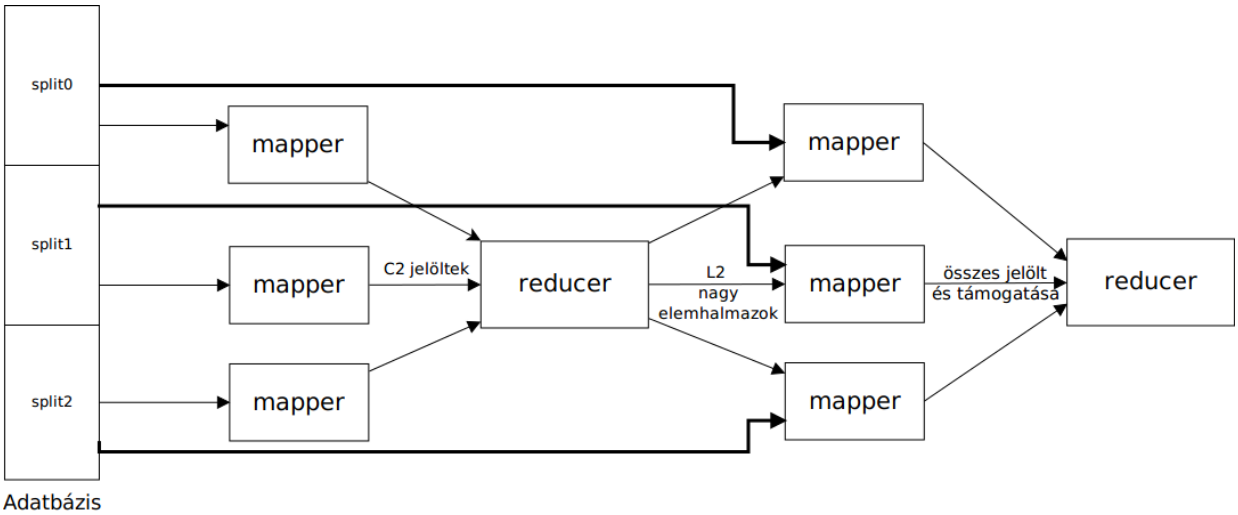
Mégsem vetettem el teljesen a nyers erő használatának ötletét, ehelyett megpróbáltam a keresési teret szűkíteni. Kezdeti heurisztikának a 6.2 pontban vázolt módosított Apriori első MapReduce lépésének kimenetét választottam, azzal az apró különbséggel, hogy egy lépésben előállított C_3 jelölt elemek helyett az L_2 nagy elemhalmazokat állítom elő, ezzel már az első lépésben számítási időt spórolva. A második lépésben történik a jelöltek generálása, kihasználva az apriori tulajdonságot, csak olyan halmazokat generálok a tranzakció elemeiből aminek minden kételemű részhalmaza része az L_2 nagy elemhalmazoknak. A nagy elemhalmazokat tartalmazó tömb $O(1)$ lépésben lekérdezhető, így a jelöltek generálásakor gyorsan megállapítható, hogy melyik elemekből biztosan nem lehet nagy elemhalmaz. Az utolsó reduce lépésnek a mapper folyamatokban generált jelöltek támogatását kell összegezni és kiszórnia a minimum szintet nem elérő jelölteket.

Az első mapper pontosan megegyezik a 6.2. fejezetben módosított függvénnyel, az első reducer pedig csak annyiban különbözik, hogy az összegzés végeztével nem állítja elő a C_3 halmazrendszert. Az érdemi változás a másodjára futó mapper függvényben van, ennek működése a 3. algoritmust leíró pszeudokódban látható.

```
candidate_tree = read_from_file(l2largesets)
l2_matrix = new array[1000][1000]
l2_matrix = read_from_file(l2largesets)
foreach transaction in database_split:
    foreach combination in combinations(transaction):
        if every 2-subset of combination in l2_matrix:
            candidate_tree(combination) += 1
print serialize(candidate_tree)
```

3. algoritmus: Nyers erővel dolgozó algoritmus második mapper lépése

Ez az algoritmus két MapReduce lépésben implementálható. A lépések a 2. ábrán látható módon követik egymást.



2. ábra: Két menetben történő gyakori elemhalmaz keresés, nyers erővel

7 Implementáció

7.1 Implementációs akadályok

A Hadoop Streaming környezetben egyetlen MapReduce művelet futtatására után, a munka végeztével leáll. A fentebb vázolt elosztott Apriori algoritmus viszont rekurzívan fut saját kimenetét felhasználva egészen addig amíg nem készül el a végső halmazok előállításával. Itt rögtön két probléma is adódik. Az egyik, hogy dinamikusan kell újraindítani a klaszteren a munkát. A másik pedig, hogy a befejeződött munka kimenetét át kell adni az éppen induló új munkának. A Hadoop ugyan biztosít lehetőséget munkák egyszerű láncolására, de ehhez pontosan ismerni kell, hogy hány munka fog egymás után következni. Ezt az információt sajnos az algoritmusfutásának végéig nem tudhatjuk. Ha tudnánk is előre, hogy hány egymást követő ciklusban fog lefutni a program sem tudunk Hadoop láncolást alkalmazni, ugyanis a bemenet visszacsatolása önmagában nem elég, szükség van az eredeti adatbázisra is, a láncolás viszont automatikusan felülírja a bemenetet.

Az munka kimeneti fájljának átadása manuálisan viszonylag egyszerűen megoldható, a Hadoop két módot is biztosít egy fájl olvashatóvá tételére. Az egyikkel tet-

szőleges helyi fájlt adhatunk át a klaszternak, ez esetünkben nem hatékony mivel a kimeneti fájlt a helyi gépre kell másolni előbb, csak azért hogy aztán újra az elosztott fájlrendszerre kerüljön. A másik kapcsoló viszont pont azt teszi amire szükségem volt, egy paraméterként megadott HDFS-en tárolt fájlt elérhetővé tesz a klaszteren futó összes folyamatnak. Mivel a kimeneti fájl neve ismert, ezért új munka kezdetekor könnyedén át tudom adni a következő indítandó munka számára elérhető fájlként.

A mapperként futó program ellenőrzi a jelölteket tartalmazó fájl jelenlétét. Amennyiben nem találja azt, akkor biztos lehet benne, hogy ő fut elsőként így feladata az elemek megszámolása, hogy abból elkészülhessen az L_1 halmaz. Ha fájl létezik akkor beolvassa azt a fentebb tárgylat adatszerkezetbe majd megkezdi a jelöltek megszámolását.

A Hadoop által futtatott folyamatok egy elszigetelt, sandbox szerű környezetben futnak a szolga gépeken, a külvilággal nem tudnak kommunikálni. A mérésekhez pedig változó minimális támogatás értékek átadására volt szükség, amit nem lehetett konstansként tárolni a reducer program forrásában. Parancssori paraméterek átadására nincs lehetőség, viszont a Hadoop megkérhető, hogy bizonyos környezeti változókat biztosítson a futó folyamat számára. Az elosztott apriori-algoritmus bemeneti paramétere a minimális támogatottság értéke, ezt egy MINSUP nevű környezeti változóban helyeztem el, amit a Hadoop minden szolga gépen futó folyamatnak biztosított. A reducer szkriptnek csupán annyi dolga van, hogy induláskor ennek a változónak értékét kiolvassa, majd eszerint folytassa futását és a jelöltek generálását.

A MapReduce munka folyamatos újraindításának problémájára azonban nincs Hadoop által biztosított módszer, megoldására egy egyszerű Python programot készítettem. A program két dologért felelős: képes a következő MapReduce folyamat elindítására, valamint ellenőrzi a véget ért munkák eredményét. Emellett számon tartja, hogy hanyadik lépésnél tart az algoritmus futása és ennek megfelelően

állítja elő a jelölteket tartalmazó fájlhoz tartozó útvonalat, amit a soron következő munka indításához használ fel.

Az algoritmus futásának végét a jelölteket tartalmazó fájl első sorában megjelenő RESULT karaktersorozat jelzi. Az általam írt keretprogram minden lépés végeztével ellenőrzi a kimenet első sorát és csak akkor indítja el ismét a munkát ha nem találja ott ezeket a karaktereket.

7.2 Tesztadatok

A tesztadatokat egy szintetikus adat generátorral állítottam elő. Két különböző méretű adatbázist hoztam létre. Az egyik ötszázezer tranzakciót, a másik kétmilliót tartalmaz. A tranzakciókban megtalálható egyedi elemek száma 1000.

A tesztadat generátor kimeneti formátuma bőbeszédű, és az egy tranzakcióhoz tartozó elemeket több soron keresztül sorolja fel. A Hadoop Streamingen keresztüli működéshez szükséges, hogy minden egyes tranzakció egy sorba kerüljön. Az adatok konvertálására egy rövid Python szkriptet készítettem.

Az eredményül kapott tesztfájlokat feltöltöttem a klaszteren elhelyezkedő HDFS fájlrendszerbe, hogy Hadoop alól elérhetőek legyenek. Feltöltés előtt a fájlokat több részre daraboltam a 8.2. fejezetben taglalt okokból.

8 A mérési környezet bemutatása

A méréseket a BME I épület 206-os laborjában lévő számítógépeken végeztem. Ezekben a gépekben 2.1Ghz órajelű Intel Core2 típusú processzorok és 4 gigabyte RAM található. A rajtuk futó operációs rendszer Windows XP. A gépeket 100 megabit/sec sebességű switchelt ethernet hálózat köti össze.

8.1 A mérési környezet elkészítése

Az Apache Hadoop futtatásához rendszergazdai jogosultságra és Linux operációs rendszerre van szükség. Bár a Windowson történő futtatás lehetséges ha telepítjük a *cygwin* által biztosított Linux-kompatibilitási réteget, ám ez a megoldás ke-

vésbé dokumentált és a laborbon lévő gépeken futó rendszerekkel sem akartam kísérletezni. Kézenfekvően adódott, hogy a rendszerre előre telepített, 4-es verziójú *VMWare Player* virtualizációs szoftvert használjam fel és a méréseket virtuális gépeken futó Linux környezet alatt végezzem.

A virtuális gépeken futó operáció rendszernek az *Ubuntu Linux 12.04.1*-es, szerverekre szánt verzióját választottam. A Hadoop számára nincs preferált disztribúció, egyedüli követelmény az, hogy legyen elérhető Java futtatókörnyezet a rendszeren. Így szabad kezem volt a rendszer kiválasztásában, ezért választottam egy széleskörűben használt, jól dokumentált disztribúciót.

A Hadoop klaszter felállításához két különböző virtuális gépre volt szükségem. Az egyik rendszerből a klaszter működéséért felelős mester gép lett, míg a másik a szolga gépekre került. Mivel a laborban lévő gépek egyformák, egyet tetszőlegesen választva mester gépnek neveztem ki, és azon futtattam a virtuális gépet a Hadoop mester konfigurációt futtató Linux rendszerrel. Másik kilenc számítógépre a szolga virtuális gép került. A szolga gépeken futó virtualizált Linux rendszerekben csak minimális módosításra van szükség, a hálózati azonosítót kell egyedi értékre változtatni, hogy a mester gép meg tudja különböztetni őket. Miután az összes virtuális gép elindult az egész Hadoop klaszter felállítását és vezérlését a mester végzi. Az összes gépen futó *DataNode* és *JobTracker* folyamatok hálózaton keresztüli elindításáért és leállításáért is a mester gép felelős.

Munkák futtatása közben a mester gép is dolgozik, tehát a mérések tíz gépből álló Hadoop klaszteren történtek.

8.2 Mérési konfigurációk

A mérés egyik célja annak vizsgálata, hogy minként viselkednek a futtatott munkák ha változik a klaszter mérete, mekkora sebességkülönbséget tapasztalunk ha kevés csomópont dolgozik. Hogy a mérés automatizálható tehető legyen, a klaszter megoldást kellett találnom a klaszter méretének dinamikus módosításra.

A klaszter méretének állítására a Hadoop nem biztosít egyértelmű és determinisztikus módszert. A beállítások között rögzíthetjük, hogy maximum hány *mapper* és *reducer* folyamat futhat egy időben, azonban az egy munkához rendelt gépek számát a munka indításkor egy belső heurisztika dönti el. A méréseimhez viszont pontosan meg szeretném határozni, hogy hány gép dolgozzon a klaszterből egyszerre. Szerencsére sikerült felismerni a Hadoop heurisztika működésének egy „kiskapuját". A Hadoop Streaming a munka bemeneteként megadott fájlokat nem fűzi össze, hanem minden fájlhoz külön folyamatot indít. Tehát, ha például öt fájlt adunk meg bemenetként, akkor legalább öt mapper folyamat fog elindulni. Innentől elég volt gépenként egyre korlátozni az egyszerre futtatott mapper folyamatok számát és korlátozni a globális maximumot, hogy pontosan annyi gép dolgozzon a klaszterben amennyit épp szeretnék.

Ahhoz, hogy a mérés teljesen automatizálhatóvá váljon, a bementi fájlokat különböző könyvtárakba egyenlő méretűre daraboltam. Az első könyvtárban 10 részre, a következőben 9-re és így tovább egészen két egyenlő darabig. A tesztelési fázisban a klaszter konzisztensen annyi géppel dolgozott ahány részre darabolva kapta meg a bemeneti adatfájlt.

8.3 Mérés automatizálása

A méréseket tehát a fent bemutatott 10 számítógépből álló, Hadoop környezetet futtató klaszteren végeztem. Egy bemenetei adatbázisban több különböző minimális támogatottság bemenő paraméter szerint kerestem gyakori elemhalmazokat, minden mérést elvégezve 9 különböző klaszterérettel.

Hogy az implementált algoritmusok viselkedését vizsgálni tudjam, sok mérés elvégzésére volt szükségem. Ahhoz, hogy ne kelljen minden egyes konfigurációt és mérést kézzel elindítani, létrehoztam egy egyszerű szkriptet, ami az előre beállított sorrendben indítja el a klaszteren a munkákat, az indítási paramétereket és futási időket pedig egy állományba naplózza.

A mérést végző program működése a 4. algoritmusban vázolt módon működik.

```
for nodes in [2,3,4,5,6,7,8,9,10] {
    for minsup in [0.005, 0.007, 0.01, 0.012, 0.015, 0.02] {
      measure(nodes, minsup)
    }
}
```

4. algoritmus: a mérés automatizálását végző program

Ez összesen 54 mérést jelent egy bemeneti adatbázison. Mivel két különböző bemeneti adatbázissal dolgoztam, ezért a mérést végző program egy menetben összesen 108 mérést futtatott a klaszteren.

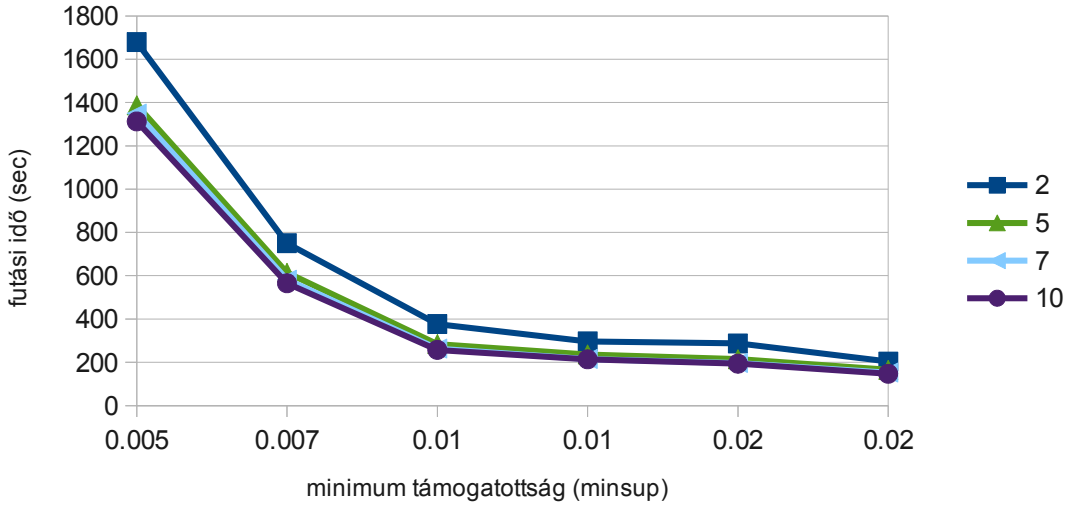
9 Mérési eredmények

A mérések lefutása után összegyűjtöttem a napló állományokat és tartalmukból grafikonokat készítve elemzem és összehasonlítom az implementált algoritmusokat.

9.1 Elosztott Apriori algoritmus

A 6.1. fejezetben tárgyalt, elosztott működésre módosított Apriori algoritmus mérési eredményei a 3. ábrán láthatóak. A mérést összesen kilenc, kettőtől tíz gépes klaszterkonfiguráció elvégeztem de a grafikon csak négy konfigurációt hagytam a könnyebb átláthatóság kedvéért. A kihagyott adatsorok nem hordoznak plusz tartalmat. Az algoritmus a vártnak megfelelően nagyobb minimum támogatás értékek mellett teljesít jól, ekkor kevesebb MapReduce ciklus alatt megtalálja a gyakori elemhalmazokat és az új elemek generálása is gyorsabban végez.

Az algoritmus skálázódása a gépek számának növekedésével mérhető, de nem látványos. Ennek oka, hogy a számítások egy jelentős részét kitevő jelöltek generálását mindig egy gépen, egy reducer folyamat végzi és ez szűk keresztmetszetet képez. Mivel a jelöltek generálásához globális állapot ismeretére van szükség, ennek a lépésnek a párhuzamosítása nem triviális.

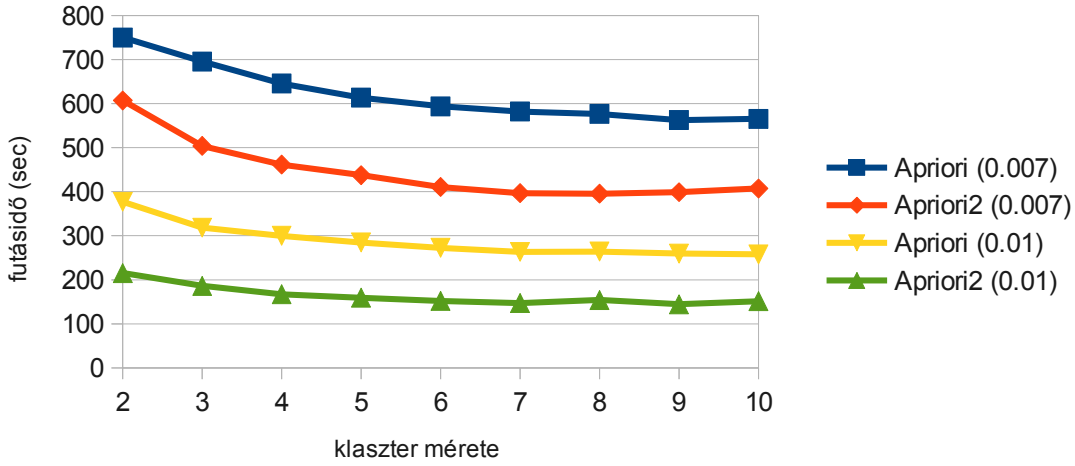


3. ábra: Elosztott Apriori algoritmus futási ideje a minimum támogatottság függvényében különböző csomópontszám esetén

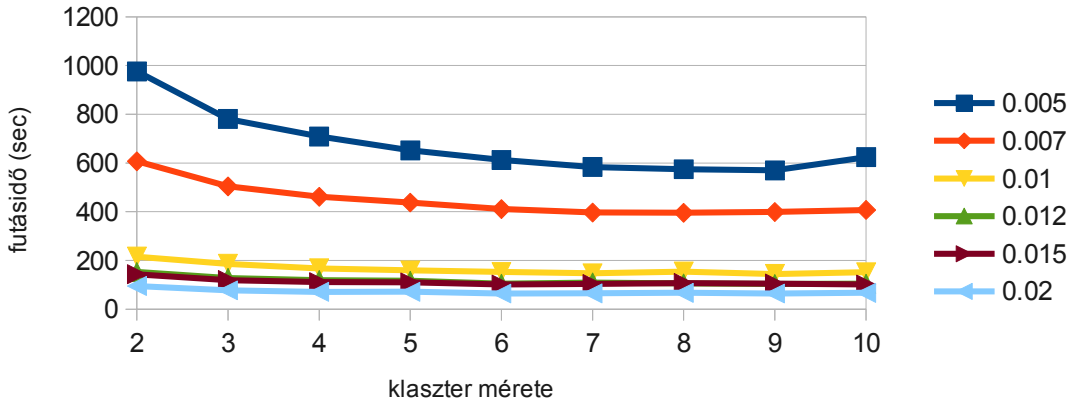
9.2 Javított elosztott Apriori algoritmus

Az 5. ábrán látható, hogy az első két lépést összevonó, 6.2. fejezetben tárgyalt, módosított Apriori algoritmus viselkedése nagyon hasonló az eredeti algoritmushoz. Magasabb támogatásoknál gyorsabb, alacsonyaknál lassabb. A reducer okozta szűk keresztmetszet a skálázódásban itt is érzékelhető.

A fő kérdés, hogy hogyan teljesít az algoritmus az eredetihez képest. A 4. ábrán lévő grafikonon együtt látható az eredeti elosztott és a módosított Apriori algoritmus két-két futási ideje azonos minimum támogatás bemenő paraméterekre. A módosított algoritmus mindkét esetben egyértelműen gyorsabban mint az eredeti párja. Megéri tehát több jelölt generálásával kiváltani egy idő és számításigényes, teljes MapReduce ciklust.



4. ábra: Elosztott Apriori és módosított változatának összehasonlítása



5. ábra: Módosított elosztott Apriori mérési eredményei

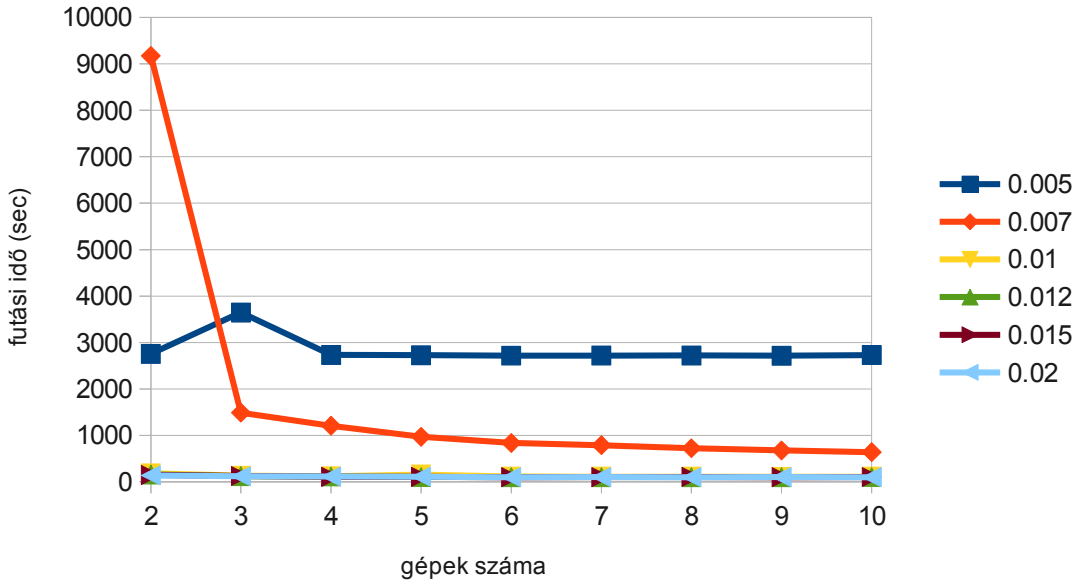
9.3 Nyers erőt használó megoldás

A 6. ábrán látható, miként viselkedik a két-elemes, heurisztikával, nyers erőt alkalmazó algoritmus. Az 1% vagy nagyobb minimum támogatási feltétellel indított mérések és az 1% alattiak között jelentős eltérés tapasztalható. Olyannyira, hogy a grafikonon az 1%-nál nagyobb mérések eredménye nem kivehető, ezért azokat külön grafikonon is megjelenítettem a 7. ábrán. Az 1% körüli nagy ugrásnál az

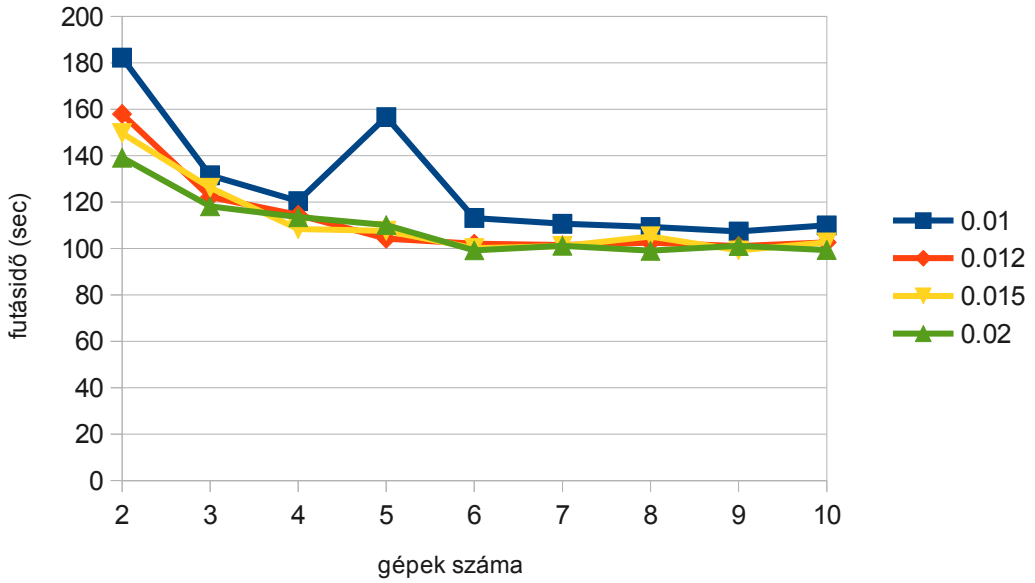
elemhalmazok mérete is túl naggyá válik, és exponenciálisan sok jelölt elem generálása rengeteg tár és számítási kapacitást igényel. A heurisztika ami a szűrést végzi, 1% fölött viszont kezelhető méretek közt tudja tartani a potenciális jelölteket így a nyers erővel történő számolás ilyen minimum támogatási értékeknél valós alternatívája lehet a rekurzív jelölt-generálást alkalmazó algoritmusoknak.

Az egy reducer folyamat okozta szűk keresztmetszetet ennél az implementációnál is tapasztaljuk.

A 7. ábrán látható kilengés az 5 gépes konfigurációnál valószínűleg egy mérési hiba, ugyanis többi bemeneti paraméternél az 5 gépes mérés nem produkált hasonló eredményt. A mérés megismétlésére nem volt lehetőségem.



6. ábra: Heurisztikus gyakori elemhalmaz kereső mérés eredményei



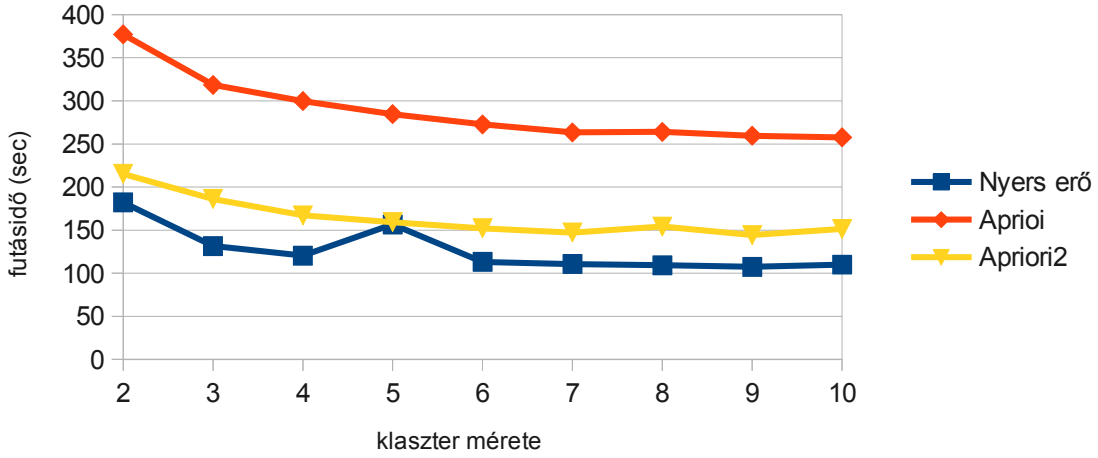
7. ábra: Heurisztikus gyakori elemhalmaz kereső mérés eredménye magasabb minimum támogatásnál

9.4 Az eredmények összehasonlítása

A 8. ábrán látható a 6. fejezetben taglalt három algoritmus egy-egy lefutása, azonos bemeneti paraméterek mellett. Érdemes megfigyelni, hogy a 6.2. fejezetben javasolt módosítással minden esetben sikerült javítani az Apriori algoritmus futási eredményein, függetlenül a számításban részt vevő gépek számától. Az eredmény javulása várható volt, mivel sikerült egyel csökkenteni az adatbázis végigolvasások számát. A gyorsulás mértéke igen jelentős majdnem eléri a kétszeres szintet, ez azt jelenti, hogy a módosítatlan algoritmus az első két MapReduce lépésben tölti el az idő jelentős részét.

Érdemes még megjegyezni, hogy az egyszerű heurisztikával működő, nyers erőt használó algoritmus milyen jó eredményeket ért el 1% fölötti minimum támogatású mérésekben. Ezekben az esetekben mindig gyorsabbnak bizonyult a módosított Apriori algoritmusnál is, igaz nem mindig látványos különbséggel.

A nyers erőt használó algoritmus az 1% alatti támogatásoknál teljesít csak rosszabbul, és ezen esetekben jelentősen rosszabbul, a másik kettőnél. Ez a bemenő paraméter gyakorlatban extrémen kicsinek tekinthető, így kijelenthetjük, hogy jó heurisztikát választva a nyers erőt használó módszer is versenyképes.



8. ábra: Három különböző algoritmus teljesítménye ugyanakkora minimum támogatásnál

10 Összefoglalás

Megismerkedtem az Apache Hadoop által biztosított szoftverplatformmal, és módosítottam egy klasszikus gyakori elemhalmazokat előállító algoritmust, hogy képes legyen elosztott környezetben történő működésre. Ezt követően előbb módosítottam az algoritmust, amivel minden esetben gyorsabb futást sikerült elérnem. Továbbá javaslatot tettem egy olyan, heurisztikán alapuló algoritmusra amivel a bemeneti paraméterek egy nagy tartományában gyorsabb működés érhető el mindkét Apriori verziónál.

10.1 Jövőbeni tervek

Az apriori algoritmus gyorsulása a klaszter méretének növekedésével nem volt arányban, jövőbeni vizsgálat tárgya lehet, hogy az algoritmus mely részét lehetne jobban párhuzamosítani.

Gyakori elemhalmazok keresésére több algoritmus is létezik, jövőbeni vizsgálatok tárgya lehet, hogy miként viselkednek ez az algoritmusok elosztott környezetben, összehasonlítva a nyers erőt használó megoldással.

Az algoritmusok elosztott futásának részletes elemzésére a Hadoop működésének mélyreható ismeretére van szükség. Érdekes kutatási irány lehet egy olyan szoftver fejlesztése amivel igazán részletes statisztikát gyűjthetünk a klaszter belső működéséről, úgy hogy azzal ne zavarjuk a futó munkákat.

11 Irodalomjegyzék

[1] **Apache Hadoop** – http://hadoop.apache.org

[2] Jeffrey Dean; Sanjay Ghemawat: **MapReduce: Simplified Data Processing on Large Clusters** (OSDI'04: Sixth Symposium on Operating System Design and Implementation, San Francisco, CA, December, 2004)

[3] R. Agrawal and R. Srikant: **Fast Algorithms for Mining Association Rules in Large Databases**. (Proceedings of the Twentieth International Conference on Very Large Databases , 1994)

[4] R. Agrawal, T. Imielinski, A. Swami: **Mining association rules between sets of items in large databases** (SIGMOD Conference on Management of Data, Washington, D.C., 1993)

[5] J. Han, J. Pei, and Y. Yin: **Mining Frequent Patterns without Candidate Generation** (SIGMOD International Conference on Management of Data, 2000)

[6] Jeffrey Dean, Sanjay Ghemawat: **MapReduce: A Flexible Data Processing Too**l (Communications of the ACM - Amir Pnueli: Ahead of His Time CACM, Volume 53 Issue 1, January 2010, p72-77)

[7] Pavlo, Andrew; Paulson, Erik; Rasin, Alexander; Abadi, Daniel J.; DeWitt, David J.;Madden, Samuel; Stonebraker, Michael: **A comparison of approaches to large-scale data analysis** (Proceedings of the 2009 ACM SIGMOD International Conference on Management of data, p165-178)

[8] Colby Ranger, Ramanan Raghuraman, Arun Penmetsa, Gary Bradski, Christos Kozyrakis: **Evaluating MapReduce for Multi-core and Multiprocessor Systems** (Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture)

[9] Gibbons, P. B.: **A more practical PRAM model** (Proceedings of the first annual ACM symposium on Parallel algorithms and architectures, 1989)

[10] Yang, Xin Yue; Liu, Zhen; Fu, Yan: **MapReduce as a programming model for association rules algorithm on Hadoop** (Information Sciences and Interaction Sciences (ICIS), 2010 3rd International Conference)

[11] Jiawei Han; Micheline Kamber; Jian Pei: **Data Mining: Concepts and Techniques,** 3rd ed., ISBN 978-0123814791

[12] Kovács Ferenc, Iváncsy Renáta, Vajk István: **Evaluation of The Serial Association Rule Mining Algorithm** (Proc. of The 22nd Iasted International Conference on Databases And Applications. Innsbruck, Ausztria, 2004.02.17-2004.02.19. ACTA Press, pp. 121-126.)