Budapesti Műszaki és Gazdaságtudományi Egyetem
Villamosmérnöki és Informatikai Kar

Tóth Zsolt

DIMENZIÓCSÖKKENTÉSI ELJÁRÁSOK ELOSZTOTT MEGVALÓSÍTÁSA HADOOP PLATFORMON

KONZULENS

Prekopcsák Zoltán

BUDAPEST, 2013

Kivonat

Az elmúlt évtizedben az elektronikus adattároló eszközök tárkapacitás tekintetében hatalmas fejlődésen mentek keresztül, miközben áruk egyre csökkent, így egyre több és több adat kerül tárolásra a vállalatok mindennapi tevékenysége során. Ahhoz, hogy ezen adattömegből kézzelfogható és valóban hasznos információhoz jussunk, meg kell látni a bennük rejlő rejtett összefüggéseket. Erre a hagyományos adatfeldolgozó és adatelemző módszerek csak korlátozottan vagy egyáltalán nem alkalmasak, hiszen – ha képes is eredményt produkálni az adott eljárás – a számítás olyan sok időt vehet igénybe, hogy az eredmények sokszor már nem aktuálisak.

Az adatgyűjtés felgyorsulásának közvetlen következménye, hogy sokszor kevés új információt tartalmazó adatok is tárolásra kerülnek. A statisztikában, képfeldolgozásban vagy a gépi tanulás területén régóta használnak olyan módszereket melyek automatikusan, a szakterület ismerete nélkül képesek elkülöníteni a hasznos dimenziókat a kevésbé hasznosaktól. Ilyen például a főkomponens-analízis technikája és az attribútumok közötti hasznossági sorrend felállítására hívatott úgynevezett *feature ranking* eljárások is.

Dolgozatomban ezen dimenziócsökkentési algoritmusokat fejlesztettem tovább elosztott módon úgy, hogy azok hatékonyan használhatóak legyenek Big Data környezetben, azaz olyankor, amikor a dimenziók száma nem teszi már lehetővé a hagyományos eszközök használatát. A feladat megoldásához az igen népszerű és széles körben használt Hadoop platformot, illetve az erre épülő adattárház réteget, az Apache Hive-ot választottam. Ez utóbbi rendszer lehetőséget kínál felhasználó által definiált eljárások Java nyelvű implementációjára. Ezt a funkciót használtam fel a PCA-hoz szükséges korrelációs mátrix előállítására és egy igen elterjedt feature ranking mutató, az információtartalom (information gain) kiszámításához szükséges entrópia és feltételes entrópia számolására.

Mint azt a dolgozat második felében bemutatásra kerülő mérési eredmények is igazolják, a fent említett dimenziócsökkentési problémákra az eddig használt módszerek és a Hive-ban elérhető beépített eljárások már néhány száz dimenzió esetén is a gyakorlatban használhatatlannak bizonyultak. Az általam tervezett és implementált algoritmusok azonban képesek ezen feladatokat akár több ezer attribútumból álló adathalmazon is hatékonyan elvégezni.

Abstract

In the last decade the electronic data storage devices went through a major improvement considering the devices' capacity while their price has been lessening continuously and significantly. Thus, more and more data is stored in the everyday business life. In order to retrieve concrete and useful information, the hidden connections within the plain data need to be discerned. For this purpose the traditional data processing and data analyzing techniques are not - or not completely - capable. It is more than possible that even if the applied process is able to produce the desired results, the processing may take so much time that the results will not be usable anymore.

The immediate effect of the increased data collection is that in many cases a part of the stored data does not contain a significant amount of new information. In the fields of statistics, image processing and machine learning some methods have been used for a long time to distinguish the useful dimensions from the less useful without knowing anything about the domain of the data. Such techniques are for example the methods of feature ranking and principal component analysis (PCA).

In this paper I improved these dimensionality reduction algorithms in a distributed way so that they can be used efficiently when the number of dimensions does not allow the usage of the traditional methods. To implement these techniques, I used the well-known Hadoop platform extended with a data warehouse infrastructure, the Apache Hive. I applied Hive's option to create user defined functions in Java to produce the correlation matrix required for the principal component analysis and to count the entropy and conditional entropy required for producing the widely used feature ranking index, the information gain.

Proven by the measurement results presented in the second part of the paper, the traditional methods and the Hive built-in functions are practically useless when the input data has more than a few hundred dimensions. My solutions however can execute these tasks efficiently on data sets consisting of thousands of attributes.

Tartalomjegyzék

Tartalomjegyzék ... **4**

1 Bevezetés ... **6**

2 Elosztott adatkezelés és adatfeldolgozás ... **7**

2.1 Apache Hadoop ... 7

2.1.1 A HDFS architektúra ... 8

2.1.2 A MapReduce paradigma ... 9

2.2 Apache Hive ... 12

2.2.1 A HiveQL nyelv ... 13

2.2.2 Felhasználó által definiált függvények ... 14

3 Dimenziócsökkentés nagyméretű adatokon ... **15**

3.1 Informatikai alkalmazások ... 15

3.2 Feature selection ... 16

3.2.1 Triviális és optimális feature selection algoritmusok ... 17

3.2.2 Feature selection Pearson korrelációval ... 18

3.2.3 Információtartalom számítás ... 18

3.3 A főkomponens-analízis technikája ... 19

3.3.1 PCA számítása szinguláris érték szerinti felbontás segítségével ... 19

3.3.2 Korrelációs mátrix ... 20

3.4 Az implementálandó algoritmusok kiválasztása ... 21

3.4.1 Feature selection algoritmusok ... 22

3.4.2 Főkomponens-analízis ... 23

4 Tervezés és implementáció ... **25**

4.1 A Hive UDAF felépítése ... 25

4.2 Az információtartalom számítása ... 26

4.2.1 Entrópia ... 26

4.2.2 Feltételes entrópia ... 27

4.3 A korrelációs mátrix számítása ... 30

4.4 Az eredmények validálása ... 31

5 Mérések ... **33**

5.1 Mérési környezet ... 33

5.1.1 Hardvertulajdonságok és konfigurációs adatok 33
5.1.2 A mérésekhez használt mintaadatok 34
5.1.3 A mérések automatizálása 35
5.2 Mérési eredmények 35
5.2.1 Összehasonlítás más implementációkkal 36
5.2.2 Skálázódási mérések 38
5.2.3 A mérések összegzése 43
6 Továbbfejlesztési lehetőségek, jövőbeli tervek 45
7 Összefoglalás 46
Irodalomjegyzék 47

1 Bevezetés

Vitathatatlan tény, hogy napjainkban mind az üzleti, mind a tudományos életben a statisztika és az adatbányászat kulcsfontosságú szerepet játszik. A tárolt adatmennyiség exponenciális növekedésével azonban a hagyományos eszközök már egyre kevésbé alkalmasak arra, hogy megfelelő időn belül eredményre jussanak. Erre a problémára nyújt megoldást a 2. fejezetben bemutatásra kerülő Hadoop platform, mely mára tulajdonképpen egyeduralkodóvá vált a különösen nagyméretű adathalmazok (Big Data) kezelésének területén. Az erre épített adattárház-réteg, a szintén egyre szélesebb körben használt Hive pedig egy kényelmes és egyszerűen használható felületet biztosít elsősorban az aggregációs jellegű lekérdezések definiálására és futtatására.

A dolgozat célja, hogy néhány, az adatbányászatban gyakran használt dimenziócsökkentési eljárás végrehajtására olyan implementációt adjon, amely képes áthidalni a fenti problémát, és így használható alternatívaként lépjen fel a nagy adatok feldolgozásának színterén. Az implementálandó algoritmusok kiválasztása nem volt triviális feladat, hiszen első látásra nem egyértelmű, hogy mely esetben van szükség valóban Big Data eszközökre és melyekre alkalmazhatóak a – nyilvánvalóan gyorsabban eredményre jutó – memóriában dolgozó módszerek. A szóba kerülő és a végül kiválasztott eljárásokat mutatja be a 3. fejezet, míg a tervezés és az implementáció folyamata a 4. fejezetben olvasható.

Természetesen önmagában egy helyes eredményt visszaadó függvény megvalósítása még nem jelenti azt, hogy az adott eljárás valóban a gyakorlatban használható alternatívát biztosít. Ez akkor lenne elmondható, ha az adatsorok számának növekedésével lineárisan vagy még kedvezőbben nőne a végrehajtási idő. Ennek bizonyítására, illetve a már létező implementációk teljesítményével való összevetés céljából készült részletes mérések olvashatóak az 5. fejezetben.

A mérések elvégzése után már tisztán láthatóvá váltak a Hive UDAF alapú megvalósítás előnyei, érdemes tehát átgondolni, hogy mely kiegészítő műveletek megvalósításával lennének a leghatékonyabban használhatóak az implementált dimenziócsökkentési eljárások a kapcsolódó tudományos és ipari területeken. Ezeket a lehetőségeket tekinti át a 6. fejezet.

2 Elosztott adatkezelés és adatfeldolgozás

Az informatika fejlődése során mindig kiemelt figyelmet kaptak az elosztott és a párhuzamos rendszerek. Egyrészt ilyen eszközök használatával a rendszer szolgáltatásbiztonságát és robosztusságát nagymértékben lehet növelni például redundáns adattárolással, másrészt a párhuzamos végrehajtásból adódóan a futási idő radikálisan csökkenthető. Ez utóbbi szempont különösen fontos egyes nagy számításigényű területeken, mint a gépi tanulás, a képfeldolgozás vagy az adatbányászat. Az alapkoncepció itt triviális: mivel a számítógépek ár-teljesítmény függvénye közel sem lineáris, ezért költséghatékonysági szempontból érdemesebb használni három darab P teljesítményű szervert, mint egy önálló 3P teljesítményűt. Természetesen a párhuzamosításnak ára van: a megszokott egy számítógépre épülő architektúrát le kell cserélni egy bonyolultabb, elosztott rendszerre. Ebben figyelni kell többek között a csomópontok közötti kommunikáció lebonyolítására, a hálózati- és szerverhibákra, és a párhuzamosan kiszámolt részeredmények integrálására is.

Az elosztott architektúrák terjedésével piacra kerültek olyan keretrendszerek, melyek a fent említett adminisztrációs feladatokat a felhasználó számára transzparens módon hajtják végre, így a fejlesztőnek nem kell foglalkoznia azzal, hogy az adott adatszegmens vagy erőforrás pontosan hol található, vagy hogy az adott csomópont például milyen titkosítást használva kommunikál a többi résztvevővel.

2.1 Apache Hadoop

Adatintenzív alkalmazások esetén a szokásos kommunikációs és szolgáltatásbiztonsági feladatok mellett külön figyelmet kell fordítani arra, hogy a keretrendszer minél elfogadhatóbb kompromisszumot találjon a csomópontok közötti terheléselosztás és a hálózati forgalom minimalizálása között. A legnépszerűbb és talán a legkiforrottabb ilyen keretrendszer a nyílt forráskódú Apache Hadoop[1].

A Hadoop fejlesztése a Google mérnökei által 2003-ban a Google File System-ről (GFS), majd 2004-ben a MapReduce technikáról publikált tanulmány alapján

[1] http://hadoop.apache.org

kezdődött. Mivel hatalmas igény mutatkozott egy több terrabájtos adatmennyiséget hatékonyan feldolgozni képes keretrendszerre, a Hadoop 2008-ra már az Apache egyik felső szintű projektjévé nőtte ki magát. Azóta több, a Hadoop szárnya alatt született rendszer is elért erre a szintre, ilyenek például a HBase, a Hive, a Pig és a Zookeeper. Manapság olyan jól ismert vállalatok használják, mint a Last.fm, a New York Times és a Facebook[2].

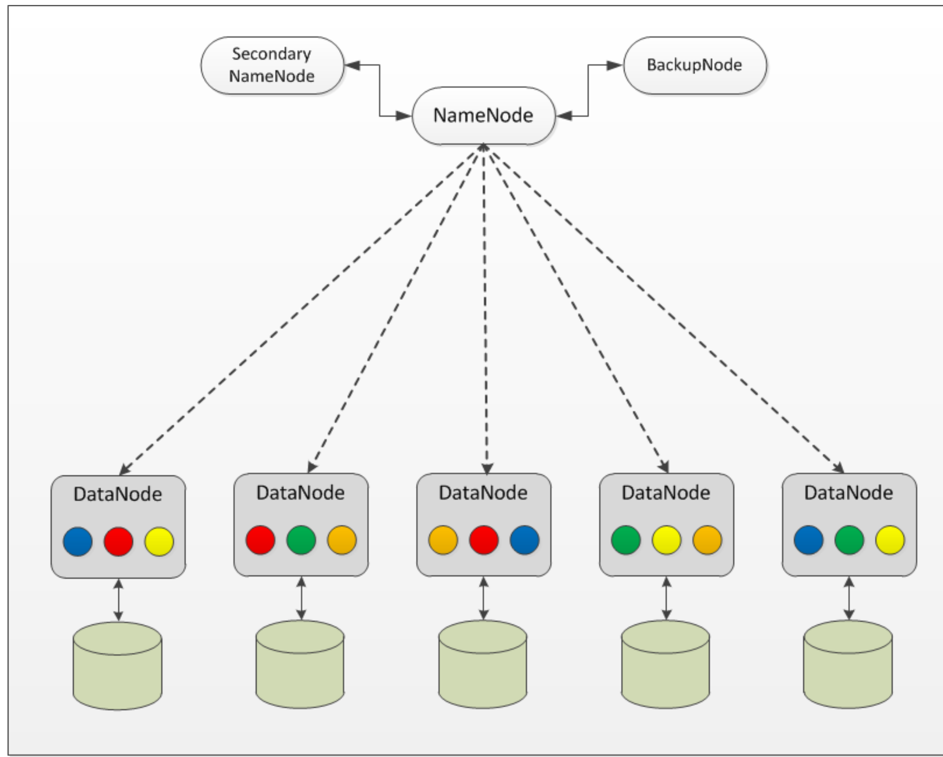
2.1.1 A HDFS architektúra

A hagyományos fájlrendszerek általában nem teljesítenek kielégítően elosztott környezetben, hiszen sok adminisztratív jellegű funkciót (kommunikáció módja, szükség esetén replikátumok létrehozása) a fejlesztőnek kell implementálnia. Kézenfekvő tervezői döntés tehát, hogy egy elosztott alkalmazásokra optimalizált keretrendszerhez saját fájlrendszer is tartozzon. A Hadoop esetében ez a GFS alapjain kifejlesztett HDFS (Hadoop Distributed File System).

A HDFS egy blokk alapú fájlrendszer, általában 64-256 MB közötti blokkmérettel (tipikusan 128 MB). Ahogy az elvárható, beépítetten kezeli a szolgáltatásbiztonság növelését szolgáló másodpéldányok létrehozását és tárolását, valamint automatikusan képes meghatározni az adatblokkok optimális tárolási helyét az egyes csomópontokon. Egy HDFS fürt master/slave alapon épül fel, melyben a master szerepét a Name Node, a slave-ekét a Data Node-ok játszák [1]. Az egyedüli Name Node felelős a fájlrendszer adminisztrációjáért, így alapvetően metaadatokat tárol az egyes tárolt fájlok helyéről, melyet kérésre a kliens rendelkezésére bocsát. A Data Node-ok felelősek a konkrét fájl tárolásáért és az írási/olvasási kérések kiszolgálásáért, valamint a Name Node utasításai alapján az egyes adatblokkok létrehozásáért, módosításáért és törléséért. Tipikusan a klaszter egy másodlagos (slave) csomópontján egy darab Data Node található, mely több adatblokkot is tartalmaz – egy nagyobb fájl tehát az elosztott és redundáns tárolás miatt egyszerre több csomóponton, tehát több Data Node részeként is megjelenik. A fent említetteken felül a fájlrendszer tartalmaz egy úgynevezett Secondary Name Node-ot, mely a Name Node-ról időnként pillanatképeket készít és tárol abból a célból, hogy egy esetleges hiba esetén a Name Node helyreállítása minél zökkenőmentesebben történhessen. Megjegyzendő, hogy a

[2] A teljes lista: http://wiki.apache.org/hadoop/PoweredBy

HDFS-ben a Name Node egyedi meghibásodási pont (Single Point Of Failure, SPOF). Ennek enyhítésére újabban aktív illetve passzív példány is található belőle a fájlrendszerben (1. ábra).



1. ábra HDFS architektúra[3]

2.1.2 A MapReduce paradigma

A MapReduce egy programozási paradigma, mely kifejezetten nagy adatmennyiségek hatékony feldolgozására lett kifejlesztve. Elsőként 2004-ben a Google mérnökei publikálták az elméletét [2]. A MapReduce ezután hamarosan világszerte ismert és használt módszerré nőtte ki magát, manapság ez a Big Data feldolgozásának elsődleges eszköze.

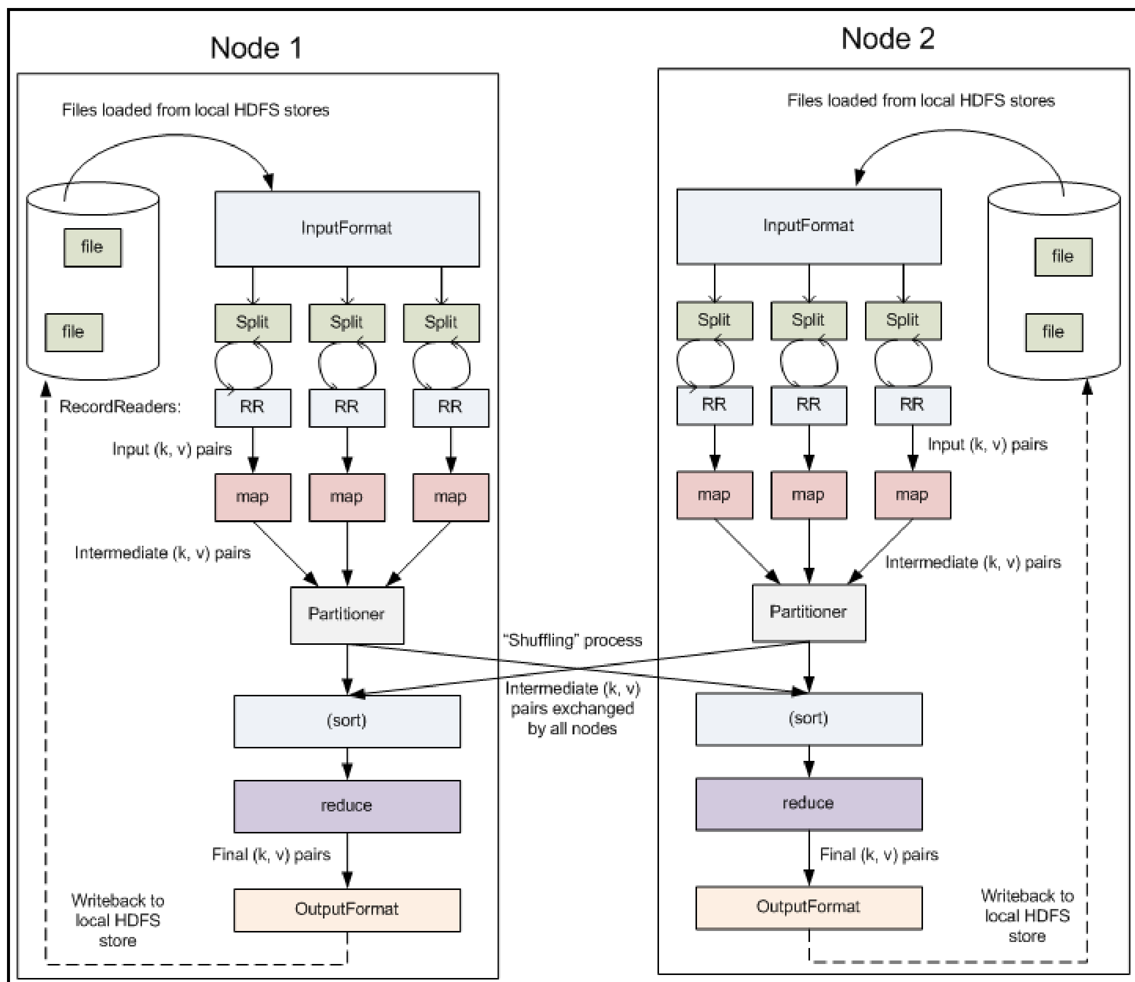
A technika lényege, hogy az adatfeldolgozást két fő lépésre bontja: Map és Reduce. A Map során a bemeneti kulcs-érték párokat a megfelelő programrészlet (mapper) átalakítja egy másik, szűrt és rendezett kulcs-érték párrá, melynek formátuma már általában közvetlenül feldolgozható a Reduce lépést végző komponens (reducer) által. A Reduce bemenete tehát a Map által kiadott kulcs-érték pár, ezen a

[3] http://simranjindal.com/wp-content/uploads/2011/10/bigdata_8.png alapján.

feldolgozóegység általában valamilyen aggregációs lépést hajt végre. Egyenlet alakban ez a következőképpen írható le:

$$Mapper: < K1, V1 > \rightarrow \; list(< K2, V2 >)$$
$$Reducer: < K2, list(V2) > \rightarrow \; list(< K3, V3 >) \tag{1}$$

Egy MapReduce feladat futtatása során minden bemeneti kulcs-érték pár mindössze egyszer van beolvasva és párhuzamosan, egymástól független részletekben kerülhet feldolgozásra (2. ábra). A Map és a Reduce lépések között opcionálisan további két funkció is felüldefiniálható, ilyen a Partitioner, mely a mapperek kimenetét rendeli össze egyenletesen a rendelkezésre álló reducerekkel, valamint a Hadoop rendszer esetén a Combiner, mellyel a Reduce lépés előtt egyfajta előfeldolgozást lehet végrehajtani. A Map és Reduce lépéseken felüli metódusokat azonban a modern rendszerekben úgy implementálják, hogy azok a legtöbb adathalmazt egyenletesen rendelik a reducerekhez, így nincs szükség ezek felülírására [3][5].

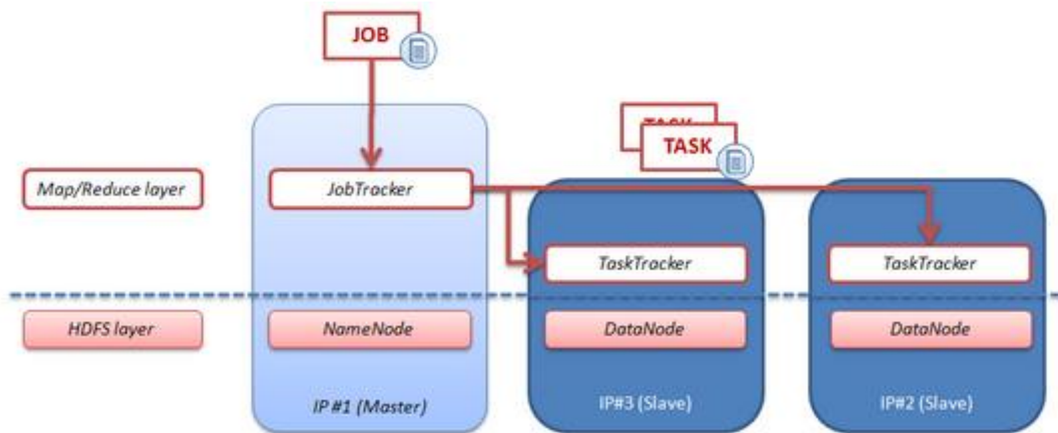


2. ábra MapReduce adatfolyam[4]

A Hadoop rendszerben a HDFS fölött egy MapReduce motor működik, mely hatékonyan alkalmazza a gyakorlatban a MapReduce-nál látott elveket és technikákat. A motor végrehajtóegységeinek két fő típusa a JobTracker és a TaskTracker. Egy Hadoop klaszterben általában egy JobTracker (olykor a magas rendelkezésre állás érdekében létrehozhatnak másodpéldányokat is) és több TaskTracker található, melyek master/slave rendszerben működnek együtt, hasonlóan a Name Node/Data Node pároshoz. Egy MapReduce feladat futtatása a következő lépésekből áll: a kliens megszólítja a JobTrackert és értesíti a kívánt feladatról, miközben felmásolja a futtatandó kódot a fájlrendszerre. A JobTracker ezután szétosztja az alfeladatokat az egyes TaskTrackerek között úgy, hogy a kommunikációs overhead csökkentése érdekében a munka lehetőleg az adathoz legközelebbi TaskTrackerhez kerüljön [3]. Ez

[4] http://farm3.static.flickr.com/2275/3529146683_c8247ff6db_o.png

végrehajtja a feladatot egy saját Java virtuális gépen, miközben időnként úgynevezett heartbeat üzeneteket küld a JobTrackernek, így értesítve azt a csomópont működésének tényéről és állapotáról. Ha néhány ilyen üzenet elmarad, a JobTracker egyszerűen egy másik TaskTrackerhez utalja ki a megszakadt munkát (3. ábra).



3. ábra Hadoop klaszter architektúrája[5]

2.2 Apache Hive

A Hadoop-ra való közvetlen fejlesztés, azaz a Java nyelvű MapReduce eljárások írása már viszonylag egyszerű feladatok esetén is redundáns és fejlesztői szempontból kevésbé hatékony tud lenni. Ennek a problémának az orvoslására elkezdtek fejleszteni olyan eszközöket, melyek magasabb szintű bemeneti parancsok alapján képesek a Hadoop által futtatható MapReduce feladatokat generálni. A legismertebb ilyen rendszerek a Pig[6] és a Hive[7]. A fejezetben ez utóbbi kerül részletesebben bemutatásra.

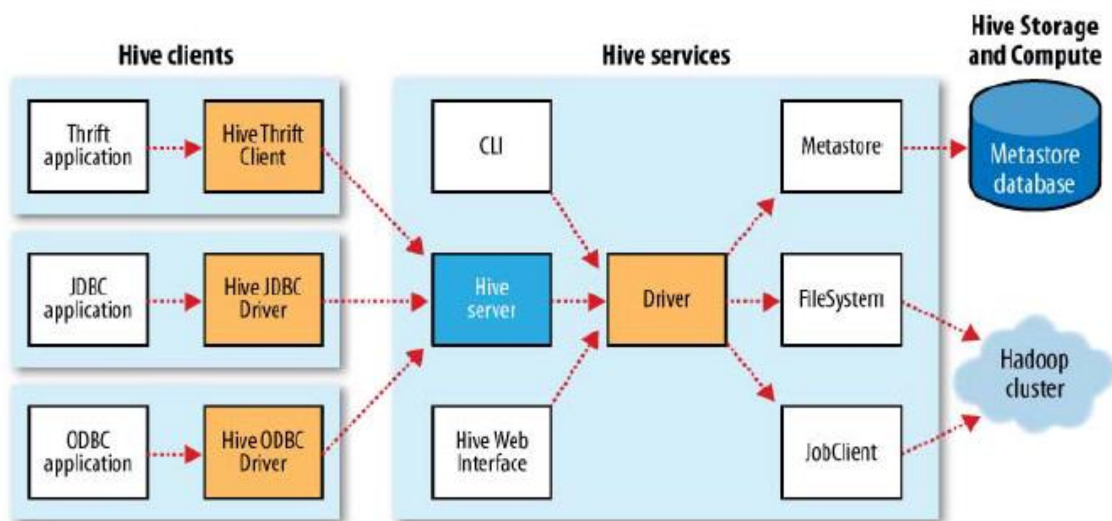
A Hive egy Hadoop alapokon futó, arra ráépülő adattárház réteg. Elsősorban arra lett tervezve, hogy az adattárház-kezelésben megszokott aggregációs feladatokat gyorsan és hatékonyan legyen képes elvégezni Big Data környezetben. A Hive népszerűségét mutatja, hogy olyan cégek fejlesztik és használják, mint a Facebook, a Last.fm és a Netflix.

[5] http://blog.octo.com/wp-content/uploads/2010/10/ScreenShot224.png

[6] http://pig.apache.org/

[7] http://hive.apache.org/

Az alábbi ábrán a Hive architektúrája látható. A felső rétegben találhatóak az ODBC és JDBC interfészek, melyek Java kódból való elérést tesznek lehetővé, valamint a Thrift szerver, mely a több programozási nyelvben (C++, PHP, Python) is használható Thrift kliensen keresztüli elérést biztosítja. Lehet kommunikálni továbbá egyszerű parancssoros felületen (Command Line Interface, CLI), valamint webes interfészen keresztül is. A munka „lényegi” részét a Driver végzi, mely a Metastore-ból kinyert információkat felhasználva MapReduce feladatokat generál a HiveQL nyelven megadott lekérdezésekből, majd végrehajtja azokat [5] (4. ábra).



4. ábra Hive architektúra [5]

2.2.1 A HiveQL nyelv

A Hive egyik nagy előnye, hogy egy SQL-hez hasonlatos nyelv, az úgynevezett HiveQL segítségével lehet megfogalmazni a futtatni kívánt lekérdezéseket, amikből a rendszer automatikusan Hadoop MapReduce feladatokat generál.

Bár a HiveQL azzal a céllal készült, hogy az SQL nyelvet ismerő fejlesztők azonnal képesek legyenek használni, több különbség is van a két nyelv között. Nem tartalmaz például utasításokat tranzakciók és materializált nézetek kezelésére, de olyan alapvető elemek is hiányoznak belőle, mint az egyszerű *„insert into ... values”* parancs, az *update* jellegű műveletek vagy az allekérdezések (*select...where...in/extsts*) támogatása. Vannak azonban olyan parancsok, melyek a HiveQL szerves részét képezik, azonban az SQL-ben nem találhatóak meg. Ilyen például az *„insert into… select”* parancs, mellyel egy lekérdezés kimenetét automatikusan beszúrhatjuk egy adattáblába. További különbség, hogy a HiveQL jelenlegi verziója nem támogat egyes

SQL adattípusokat, mint a *char* és a *varchar*, elérhetőek viszont egyes kollekció típusok, mint a *Map* és a *List* [4].

Bár mindez jelentős megszorításnak tűnhet, a Hive által elsődlegesen megcélzott aggregációs jellegű feladatok természetesen elvégezhetőek a nyelv jelenlegi állapotában is. Ugyanakkor a Hive jövőbeli verziói várhatóan egyre inkább le fogják fedni az SQL-92 szabványt [5][6].

2.2.2 Felhasználó által definiált függvények

A Hive tervezése során nagy hangsúlyt fektettek a testreszabhatóságra, ezért a rendszer kiegészíthető további, általában Java nyelven írt függvényekkel. Az egyik ilyen lehetőség az úgynevezett *serde* (Serializer/Deserializer) használata. Ez gyakorlatilag egy Java osztály, mely a HDFS-en tárolt komplex bemeneti formátum (pl. JSON) átalakítására szolgál. A Deserialize lépés bináris vagy karakteres bemenetet alakít át egy, a Hive által kezelhető Java objektummá, míg a Serialize épp az ellenkezőjét csinálja, azaz a Java objektumot alakítja vissza olyan formátumba, amit a fájlrendszer hatékonyan tárolni tud. Előbbi általában select típusú lekérdezéseknél, míg utóbbi leginkább insert-select parancsok végrehajtásánál használatos [7].

A másik gyakran használt kiegészítő lehetőség a felhasználó által definiált függvények (User Defined Function, UDF) használata. Sokszor a tárolt táblán elvégezni kívánt művelet nem hajtható végre egyszerűen HiveQL parancsok segítségével, ezért a rendszer beépítetten támogatja a saját metódusok Java nyelvű létrehozását és használatát. Természetesen a leggyakrabban használt függvények beépítetten elérhetőek a Hive-ban, ám ezek az esetek egy részében messze nem elegendőek a kívánt eredmény előállítására.

Az UDF-eknek három típusát lehet megkülönböztetni. A *User Defined Function*-t a legegyszerűbb megvalósítani; ez csak a Map oldalon fut, soronként olvas és általában egyszerű soronkénti műveleteket valósít meg. A második típus a *User Defined Table-Generating Funtion* (UDTF), mely még mindig soronkénti végrehajtásra használatos, ám ez már –ahogy a neve is mutatja – nem egy egyszerű értéket, hanem egy teljes adattáblát képes visszaadni. A harmadik típus, a *User Defined Aggregating Function* (UDAF) már elsősorban aggregációs jellegű funkciók megvalósítására használatos. Az előzőekkel ellentétben az UDAF Map és Reduce oldali végrehajtást is eredményezhet a metódus jellegétől függően [8].

3 Dimenziócsökkentés nagyméretű adatokon

A Big Data-val, vagyis a nagyméretű adatállományokkal kapcsolatos problémák és lehetőségek az elmúlt évtizedben váltak láthatóvá az informatika világában. Ennek oka, hogy csak akkorra ért el az „egységnyi" adat tárolásának költsége elegendően alacsony szintet ahhoz, hogy a részletesebb információk kinyerésébe és elraktározásába fektetett pénz egy idő után megtérüljön a vállalatok számára. Természetesen önmagában az adattömeg jelenlétéből még nem lehet profitálni – szükség van olyan eszközökre, melyek hatékonyan képesek felfedni az adott vállalat számára fontos, rejtett összefüggéseket.

Elsőre talán nem egyértelmű, hogy mely területeken jelenthet segítséget a nagyméretű adathalmazok hatékony feldolgozása. A legalapvetőbb eset az informatikai vállalatok által gyűjtött felhasználói adatok, például web logok (internetes naplófájlok), a felhasználói szokások vagy a felhasználók közötti kapcsolatok elemzése. Ezen felül azonban sok más szakterületen is megjelennek hasonló feladatok. A biológia területén belül a genomikában (a gének kölcsönhatásait vizsgáló tudomány területén), az informatikában például a biztonságtechnika, az adatbányászat és a gépi tanulás területén, továbbá a telekommunikációs és banki ajánlattevő rendszerekben és nagyvállalati környezetben a vezetői döntéstámogatás egyik formájaként (pl. adattárházakon különböző analitikus jellegű lekérdezések elvégzésénél) is lehet szükség Big Data módszerekre [9][10].

Az első alfejezetben a Big Data néhány, a feladat gyakorlati részéhez kapcsolódó használati esetéről esik szó, majd a továbbiakban ezek közül részletesebben bemutatásra kerülnek azok a mutatók, melyek konkrét előállításáról a dolgozat második felében szó esik.

3.1 Informatikai alkalmazások

Ahhoz, hogy az implementált elosztott algoritmusok jelentőségét megértsük, érdemes megvizsgálni közelebbről a fent említett területek közül néhányat. A gépi tanulás, a statisztika és az adatbányászat egymás között nagymértékű átfedéseket tartalmaznak, így egyes mutatók és algoritmusok akár mindhárom szakterületen is hasznosíthatóak.

A gépi tanulás és az adatbányászat egyik célja lehet, hogy a rendelkezésre álló tanulóhalmazban olyan szabályokat keressünk, melyek minél jobban meghatározzák az adott célváltozó értékét. Tipikus példa erre egy olyan adathalmaz, melyben az attribútumok egy kép pixeleinek értékei, a célváltozó pedig a képen szereplő valamilyen betűtípussal írt szám. Ebben az esetben tehát a tanulási folyamat során olyan pixeleket keresünk, melyek értéke nagy valószínűséggel meghatározzák a számjegyet.

A statisztika területe szorosan kapcsolódik a fenti két tudományhoz. Míg az adatbányászat és a gépi tanulás csak az utóbbi néhány évtizedben, az informatika térhódításával kezdett el fejlődni, statisztikai számításokat már sokkal régebb óta használnak. Könnyen belátható, hogy a fenti két területen használt algoritmusok egy része tulajdonképpen tisztán statisztikai számításokat igényel. Visszagondolva a fenti számjegyes példához érezhető, hogy ha a pixelek értékéről és a hozzátartozó számjegyekről képesek vagyunk valamilyen statisztikát készíteni, közelebb kerülünk a probléma megoldásához.

A továbbiakban néhány olyan konkrét mutató kerül ismertetésre, mely mindhárom fenti tudományterület alapvető számítási módszerei közé tartozik.

3.2 Feature selection

A gépi tanulás során a célváltozót általában sok különböző jellemző írja le. Ezek természetesen nem hordoznak azonos mennyiségű hasznos információt. Ha valamilyen oknál fogva a futási idő és az erőforrás-használat mértéke nem lényeges, akkor egész egyszerűen az összes attribútum-részhalmaz megvizsgálható relevancia szempontjából és kiválasztható közülük a legkedvezőbb. A gyakorlati esetekben azonban erre nincs elég idő, így olyan módszereket kell keresni, melyek segítségével kiválasztható a leghasznosabb jellemzők halmaza (feature selection, variable subset selection), illetve sorrendezhetők hasznosság szerint (feature ranking) [13].

Közelebbről nézve a feature selection legfőbb előnyei a következők [11].

- Az irreleváns jellemzők nagyban megnövelhetik a számítási (tanulási) időt. Mint később a méréseknél látható lesz, a futási idő általában exponenciálisan nő az attribútumok számának növekedésével. Nagyon fontos tehát, hogy a haszontalan jellemzőket minél korábban kiszűrjük.

- Az irreleváns jellemzők használata a tanulási folyamat során úgynevezett túltanulást (high variance problem, overfitting) eredményezhet. Például, ha egy orvosi diagnosztikai rendszerben a tanulóhalmaz tartalmazza a beteg személyi számát is, előfordulhat, hogy a tanuló program valamiféle párhuzamot vél felfedezni az azonosító és a betegség között – ami nyilvánvalóan téves feltételezés [11].
- Ha nem használ az algoritmus egy bizonyos határértéknél kevésbé hasznos jellemzőket, akkor a kész előrejelző modell mérete kisebbé, így könnyebben értelmezhetővé válik.

A következő alfejezetekben néhány konkrét feature selection módszerről esik szó, az egészen triviálisaktól kezdve a bonyolultabb, nagyobb számításigényű algoritmusokig.

3.2.1 Triviális és optimális feature selection algoritmusok

Jelölje X az eredeti, minden jellemzőt tartalmazó, n méretű attribútum halmazt. Ebből kell kiválasztani a leghasznosabb k méretű Y részhalmazt. Ezt a legegyszerűbb módon a már említett kimerítő kereséssel lehet megtalálni. Ebben az esetben $\binom{n}{k}$ esetet kellene megvizsgálni, ami már egy közepes méretű adathalmaz esetén is a valóságban kivitelezhetetlen. A gyakorlati esetek többségében a kimerítő keresésnél gyorsabban ad optimális megoldást az úgynevezett branch and bound algoritmus, amely azonban csak bizonyos speciális esetekre alkalmazható [12]. Általánosságban elmondható, hogy az optimális részhalmaz kiválasztásának problémája NP-nehéz, tehát nagy adathalmazokon csak közelítő algoritmusokkal juthatunk garantáltan belátható időn belül eredményre.

Ha nem egy attribútum halmaz kiválasztása a cél, hanem egyszerűen egy változóról kell eldönteni, hogy önmagában mennyire hasznos, használható a statisztikából ismert szórásnégyzet, mint triviális mutató. Ha a változó szórása 0, vagyis az értéke az összes adatsorban azonos, akkor egyértelmű, hogy nem tartalmaz hasznos információt, tehát eldobható. Természetesen a másik véglet is felhasználható: ha minden helyen a változó értéke különböző (például a fenti betegazonosító példában), akkor az attribútum szintén nem hordoz információt a célváltozóra nézve.

3.2.2 Feature selection Pearson korrelációval

A feature ranking, azaz a jellemzők hasznosság szerinti sorrendezése történhet a változók korrelációja szerint is. Ennek lényegét M. Hall disszertációjának tételmondata így fogalmazza meg: „… a jellemzőknek egy kedvező halmaza olyan attribútumokat tartalmaz, melyek az osztályozandó változóval nagymértékben, egymással azonban nem korrelálnak."[8] [13]. Az optimálishoz közelítő halmaz megtalálására szolgál a CFS (Correlation-based feature selection) algoritmus, melynek középpontjában a következő egyenlet áll:

$$M_s = \frac{k\overline{r_{cf}}}{\sqrt{k + k(k-1)\overline{r_{ff}}}} \tag{2}$$

A fenti képletben M_s jelöli a kiválasztott k elemű S attribútum halmaz hasznosságát, $\overline{r_{cf}}$ az osztály és a jellemző, $\overline{r_{ff}}$ pedig két jellemző közti átlagos korrelációt. Az egyenlet segítségével tehát két halmazt már össze lehet hasonlítani, de az előzőekhez hasonlóan továbbra is $\binom{n}{k}$ különböző halmazról beszélhetünk. A problémát a gyakorlati esetek többségére megoldja az M. Hall által ajánlott heurisztikus algoritmus [13], mely itt nem kerül részletezésre.

3.2.3 Információtartalom számítás

Az információelmélet témakörébe tartozó mutatók hatékonyan használhatóak feature ranking célzattal, így például az adatbányászat területén közvetlenül is alkalmazhatóak. A leggyakrabban előforduló ilyen mérőszám az információtartalom (information gain, IG).

Az információtartalom egyik ismert alkalmazási területe az adatbányászatban a döntési fa építése. Ha sok jellemzővel dolgozunk, akkor nem mindegy, hogy egy döntési fának melyik szintjén melyik változó alapján vágunk, azaz hogyan osztjuk tovább a célváltozót. Nyilvánvalóan olyan jellemzőket érdemes használni, amelyek segítségével a lehető legjobban szétválaszthatóak a különböző osztályok. Ezt az intuitív jellegű megfogalmazást konkretizálja az információtartalom mérőszáma. Egy adott változó esetében minél nagyobb ez az érték, annál inkább érdemes ez alapján építeni a

[8] A szerző fordítása

döntési fát. Egy attribútum (X) információtartalma egy adott osztályozandó célváltozó (Y) esetén a következő [14]:

$$IG(Y|X) = H(Y) - H(Y|X) \quad (3)$$

$H(Y)$ a célváltozó entrópiáját, $H(Y|X)$ pedig a feltételes entrópiát jelöli. A teljesség kedvéért álljon itt ezeknek a jól ismert mutatóknak is a számítási módja.

$$H(Y) = -\sum_{j=1}^{m} p_j \, log_2 p_j \quad (4)$$

A fenti egyenletben p_j annak a valószínűsége, hogy az Y változó az m darab lehetséges érték közül a j-ediket veszi fel. A feltételes entrópia pedig:

$$H(Y|X) = -\sum_{j=1}^{k} P(X = v_j) \, H(Y|X = v_j) \quad (5)$$

Itt $H(Y|X = v_j)$ az Y entrópiáját jelenti azokon a helyeken, ahol az X attribútum a $v_j$értéket veszi fel. A bemutatott entrópia és feltételes entrópia alkalmazhatósága igen széleskörű, hiszen nem csak az adatbányászathoz kapcsolódó területeken, hanem például a kódolástechnika legalapvetőbb számításaihoz is használható.

3.3 A főkomponens-analízis technikája

Az utolsóként bemutatott dimenziócsökkentési eljárás a főkomponens-analízis (Principal Component Analysis, PCA), mely többek között a képfeldolgozás, mintafelismerés, adattömörítés, és az osztályozás területén is hasznosítható [16]. Ennek lényege, hogy az eredeti változóhalmazt (X) tartalmazó mátrix bázisát lecseréljük egy kevesebb attribútumot tartalmazóra úgy, hogy az adathalmazról a lehető legkevesebb információt veszítsük el. A PCA matematikai kiszámításához szükség van az n sort és d oszlopot tartalmazó **X** mátrixhoz tartozó $\mathbf{C_x}$ kovariancia vagy az $\mathbf{R_x}$ korrelációs mátrixra. A $\mathbf{C_x}$ mátrix sajátérték-egyenletének megoldása után a kapott sajátvektorokat a hozzájuk tartozó sajátértékek szerinti csökkenő sorrendbe rendezve, majd ezek közül az első k elemet kiválasztva megkapjuk a transzformáció alapjául szolgáló bázist [15].

3.3.1 PCA számítása szinguláris érték szerinti felbontás segítségével

A következőkben a C. Ordonez által ajánlott PCA számítási módszer [16] főbb lépései kerülnek bemutatásra, melynek előnye a fenti módszerrel szemben, hogy olyan

adatmennyiségre is hatékonyan végrehajtható, amelynek mérete már nem teszi lehetővé a teljes adathalmaz memóriába való betöltését. Mivel jelen dolgozatnak nem célja a szinguláris érték szerinti felbontás algoritmusának, ezen belül pedig a Householder-transzformációnak és a QR-faktorizációnak a részletes bemutatása, ezért ezek elméleti háttere itt nem kerül közlésre ([16]-ban megtekinthető).

A bemutatott módszer a **C** kovarianca mátrix helyett az **R** korrelációs mátrixból indul ki. Ennek fő előnye, hogy itt az előzőekkel ellentétben a [-1; 1] tartományra normalizált értékekkel számolhatunk. Alapvetően a cél az, hogy megtaláljuk azt a transzformációt, amellyel az **X** mátrixon a PCA által előírt báziscsere végrehajtható. Ezt például szinguláris értékek szerinti felbontással (Singular Value Decomposition, SVD) tehetjük meg. Az ennek elvégzésére ajánlott algoritmus lépései a következők:

1. Az L és a Q összegmátrixok kiszámítása
2. Ezek alapján a korrelációs mátrix kiszámítása
3. Householder-transzformáció végrehajtása
4. QR-algoritmus végrehajtása

Az első két lépést a 3.3.2-es alfejezet részletezi.

3.3.2 Korrelációs mátrix

Mint ahogy a fejezet második felében látható lesz (3.4.2), a PCA kiszámításának lépései közül egyedül a korrelációs mátrix előállítása igényel Big Data módszereket, így itt csak ez a lépés kerül részletezésre. Az algoritmus bemeneti paraméterként egy n sort és d oszlopot tartalmazó mátrixot kap. Az **R** korrelációs mátrix a következő lépésekkel állítható elő [16][17]:

1. Az L sorvektor kiszámítása:

$$L = \sum_{i=1}^{n} X_i \quad (6)$$

   A fenti egyenletben $X_i = (x_{i,1}, x_{i,2}, \dots, x_{i,d})$ sorvektor. Szemléletesen tehát a mátrix minden oszlopát összegezni kell, így egy d méretű sorvektor áll elő.

2. **Q** összegmátrix kiszámítása:

$$Q = \sum_{i=1}^{n} X_i{}^T X_i \qquad (7)$$

Vagyis az **X** sorainak transzponálásával kapott oszlopvektorokat megszorozzuk az eredeti sorvektorral és az így kapott mátrixokat összegezzük.[9]

3. Ezek alapján az **R** mátrix minden $\rho_{i,j}$ elemére:

$$\rho_{i,j} = \frac{nQ_{ij} - L_i L_j}{\sqrt{nQ_{ii} - L_i^2}\sqrt{nQ_{jj} - L_j^2}} \qquad (8)$$

A további érvelések alátámasztásaképpen érdemes megjegyezni, hogy a kapott **R** korrelációs mátrix $d \times d$ méretű, tehát dimenziószáma független a bemenet sorainak számától.

3.4 Az implementálandó algoritmusok kiválasztása

Az elméleti háttér megismerése után a megvalósítandó algoritmusok kiválasztásának nemtriviális feladata következik. Ehhez több tényezőt is figyelembe kell venni és csak ezek mérlegelése után dönthető el, hogy mi és hogyan kerüljön implementálásra. Ezek a következők:

1. Van-e gyakorlati haszna a megvalósításnak?
2. Létezik-e már olyan implementáció, amely hasonló mennyiségű adatra képes ugyanezeket a számításokat elvégezni?
3. Ha már létezik, lehetséges-e az algoritmus javításával bizonyos esetekre kisebb futási idővel eredményre jutni?
4. Nem lesz-e bizonyos adatmennyiség felett elfogadhatatlanul hosszú a futási idő még direkt Big Data feldolgozására tervezett platformon is?

Ebben a fejezetben a fenti kérdések mentén kerülnek kiértékelésre az előzőekben bemutatott algoritmusok.

---

[9] [16]-ban valószínűleg egy elírás miatt a fenti egyenletben $X_i{}^T$ helyett a nyilvánvalóan helytelen $X_j{}^T$ áll.

3.4.1 Feature selection algoritmusok

Az alábbiakban a 3.2-ben bemutatott algoritmusok kerülnek elemzésre a fenti 4 szempont alapján. A konkrét esetekre való kitérés előtt elmondható, hogy az 1. pontnak mindegyik algoritmus megfelel, hiszen gyakorlati haszna az összes eljárásnak van, elsősorban a 3.1-ben bemutatott területeken.

Optimális és triviális algoritmusok

A feature selection algoritmusok közül az optimálisak (kimerítő keresés, branch and bound) rögtön „elbuknak" a fenti 4. ponton: NP-nehéz feladatról lévén szó, egy bizonyos adatmennyiség felett nincs olyan eszköz, amely hatékonyan képes lenne végrehajtani őket. Természetesen közepes mennyiségű adatra használható lehetne egy Hadoop alapú implementáció, ám az exponenciális időigény miatt ez az intervallum elég szűk, így ezekkel jelen dolgozat nem foglalkozik.

A triviális algoritmusok közül a variancia számítására a Big Data eszközök minden további nélkül alkalmasak. Éppen ezért előkerül a 2. pontban felvázolt eset: már létezik rá Hive beépített UDF, a *var(X)* függvény, mely a bemeneti attribútum szórásnégyzetét számolja ki. Ezt a függvényt lefuttatva a bemeneti mátrix minden oszlopára, majd ezekből kiszűrve a 0-hoz közeli értékeket megkapható az irreleváns jellemzők egy halmaza. Egy másik triviális módszer annak vizsgálata, hogy egy attribútum minden értéke különböző-e. Ez egy egyszerű SQL lekérdezés segítségével eldönthető.

Feature selection Pearson korrelációval

Mint az 3.2.2-ben bemutatásra került, a korreláció alapú jellemzőkinyerésnek két fő lépése van: a korrelációs együtthatók kiszámítása, valamint a heurisztika implementálása. Előbbire létezik a Hive-ban beépített függvény, a *corr(X,Y)*, mely két bemeneti változó korrelációját adja meg. Ahhoz azonban, hogy a bemeneti mátrix egy k elemű részhalmazán belül bármely attribútum pár korrelációját megkapjuk, $\binom{k}{2}$ darab ilyen függvényt kell futtatni. Ez érezhetően egy olyan pont, amin egy olyan eljárással, amely egyszerre megkapja bemenetként a teljes kiinduló mátrixot, esetleg javítani lehet. Ezért (valamint a PCA számításánál játszott szerepe miatt) a korrelációs mátrix előállítása az egyik olyan algoritmus, amely kiválasztásra került. Érdemes lenne továbbá megvizsgálni a heurisztikus algoritmus Hadoop platformra való megvalósításának lehetőségét is, de ez már nem fért be a dolgozat keretei közé.

Információtartalom-számítás

Az információtartalom számításának a 3.2.3-ben bemutatott módon két fő lépése van: a célváltozó saját entrópiájának, valamint a jellemzőkkel vett feltételes entrópiájának az előállítása. Előbbi nem feltétlenül igényel Big Data módszereket, hiszen csak egy bemeneti paramétere van, ám ha a tanulóhalmaz elegendően sok mintát tartalmaz, a hagyományos módszerek már itt sem teljesítenek megfelelően. A jellemzőkkel vett feltételes entrópiák számítása esetében azonban már a teljes bemeneti mátrixot fel kell dolgozni, itt tehát már egy közepes méretű adathalmazra is gyorsabb lehet egy Hadoop alapú implementáció. Továbbá, nagyon kevés a hatékonyan használható megvalósítás mind az entrópia, mind az információtartalom számítására. Érdekes módon sem a Matlab, sem a Hive nem tartalmaz ilyen beépített funkciót[10]. Van azonban információtartalom-számítást végző elem a RapidMiner nyílt forráskódú adatbányászati célú szoftverben[11], ám ez a teljes input mátrixot memóriában kezeli, így nagyobb adathalmazok kezelésére alkalmatlan.

3.4.2 Főkomponens-analízis

A bemutatott dimenziócsökkentési eljárások közül talán a PCA az, amely a legtöbb területen hasznosítható, a biológiától a csillagászaton át az adatbányászatig. A fenti lista 1. pontja tehát egyértelműen teljesül. Az is biztos, hogy lehet és érdemes is rá Big Data módszereket alkalmazni, hiszen 3.3-ban látható volt, hogy a bemeneti **X** mátrix mérete megfelelően nagy lehet. A széleskörű alkalmazhatóság miatt az elmúlt évek során több különböző implementáció is készült különböző platformokra, ezért a fenti lista 3. pontjából kiindulva különös figyelmet kell fordítani a megvalósított algoritmus teljesítményének mérésére (5.2.1).

Mint ahogy az 3.4.2-ben látható volt, a PCA számításának három fő elvi lépése van: a korrelációs mátrix előállítása, a Householder-transzformáció és a QR-faktorizáció. A kezdeti bemenet az **X** $n \times d$ dimenziójú mátrix. Ebből áll elő az egyes lépések során először a $d \times d$ méretű **R** korrelációs mátrix, a Householder-transzformáció után a $\mathbf{B_0}$ tridiagonális, végül a QR-felbontás után a $\mathbf{C_s}$ ortogonális és az átlóban a

[10] A Matlab tartalmaz egy beépített *entropy* függvényt, ez azonban nem klasszikus értelemben vett entrópiát számol: http://www.mathworks.com/help/images/ref/entropy.html

[11] http://www.rapidminer.com

keresett sajátértékeket tartalmazó $\mathbf{B_s}$ mátrix, melyek mérete megegyezik a korrelációs mátrix dimenzióival. Egy valós adathalmazon $d \ll n$, azaz a dimenziószám sokkal kisebb a megfigyelt minták számánál. Mivel az **R** kiszámítását követő lépések már egy jóval kisebb bemeneti mátrixon dolgoznak, így egészen addig, amíg a dimenziószám szinte elképzelhetetlen méreteket nem ölt (legalább 10^5 nagyságrend), a Householder-transzformáció és a QR-felbontás implementálására alkalmazhatóak a klasszikus, memóriában dolgozó módszerek.

Az előzőekhez hasonlóan meg kell még vizsgálni, hogy létezik-e már a problémára beépített vagy széles körben ismert implementáció Big Data környezetben. A válasz természetesen igen, hiszen – az előző fejezetben látottakkal analóg módon – a szimmetrikus és a főátlóban csupa 1-est tartalmazó korrelációs mátrixot elő lehet állítani $\frac{d^2}{2} - d$ darab kétváltozós *corr(X,Y)* függvény futtatásával is. A két implementáció közötti teljesítménybeli különbség az 5. fejezetben ismertetett méréseknél lesz részletezve.

4 Tervezés és implementáció

Az előzőekben meghatározásra kerültek a bemutatottak közül azok az algoritmusok, amelyek Hive platformon való megvalósítása gyakorlati haszonnal is kecsegtet. Ezek a következők:

- Az információtartalom számítása
- A korrelációs mátrix számítása

Ebben a fejezetben ezek konkrét implementációjának tervezése és kivitelezése kerül ismertetésre, kitérve a megvalósítás során felmerülő döntési helyzetekre, nehézségekre és az elkövetett hibákra is.

4.1 A Hive UDAF felépítése

A kitűzött célok között szerepelt, hogy Hive platformon legyenek implementálva a kiválasztott algoritmusok. Ahogy arról már a korábbiakban szó esett, a Hive rendszerben háromfajta felhasználó által definiált függvény érhető el: az UDF, az UDAF és az UDTF (2.2.2). Mivel mindkét algoritmus aggregációs jellegű műveleteket igényel, kézenfekvő választás az UDAF-ok használata.

Egy Hive UDAF alapvetően három jól elkülönülő részből áll: az *iterate*, a *merge* és a *terminate* függvényekből. A kezdeti inicializálásért felelős *init* függvény lefutása után az *iterate* végzi a lekérdezésben szereplő tábla soronkénti beolvasását, ez tulajdonképpen a MapReduce folyamat Map részének felel meg. Az elosztott végrehajtás miatt egyszerre több párhuzamos Map indulhat, így egyszerre több *iterate* függvény is futhat. Fontos megjegyezni, hogy nem tudható előre, hogy az *iterate* egy hívásakor mekkora adatmennyiség kerül feldolgozásra, és hogy mely adatsorokat melyik mapper függvénye dolgozza fel. Egy *iterate* végrehajtása után kerül meghívásra a *terminatePartial* függvény, mely egyfajta utófeldolgozást hajthat végre, majd – az összes sor beolvasását követően - a *merge*, aminek célja az *iterate*-ek által visszaadott lokális eredmények összeolvasztása egy globális állapotváltozóba. Végül a Hive a *terminate* eljárást hívja meg, amely már bemenetként kapja a tábla teljes feldolgozása után előálló állapotot és ezen végzi el a kívánt műveleteket. A *merge* és a *terminate* már a MapReduce logikájának megfelelően Reduce oldalon fut.

4.2 Az információtartalom számítása

Mint az elméleti részben (3.2.3) látható volt, az információtartalom-számítás egy adott attribútum és egy célváltozó esetén két fő lépésből áll: a célváltozó (label, azaz címke) saját entrópiájának, valamint a jellemzővel vett feltételes entrópiájának előállításából. Ezek ismeretében egy egyszerű kivonás után megkapható az információtartalom. A két mutató kiszámítása külön alfejezetekben kerül ismertetésre, de nyilvánvalóan egy darab UDAF részeként vannak implementálva.

4.2.1 Entrópia

Az entrópia számítását volt az implementáltak közül a legegyszerűbb megvalósítani. A cél az, hogy létrejöjjön egy lista, mely a célváltozóban (Y) minden értékhez eltárolja az előfordulásának valószínűségét. Erre közvetlenül lehet alkalmazni az entrópia képletét (4).

Állapot

Az állapot tárolásához szükség van egy int típusra a beolvasott sorok számához, valamint egy Java Map<String, Integer> típusú objektumra, mely a változó értékeit és előfordulásuk számát tárolja.

Iterate

A függvény bemenete a célváltozót tartalmazó oszlop egy értéke. Amennyiben az érték már szerepel kulcsként a Map-ben, megkeressük ezt a bejegyzést és megnöveljük az értékét eggyel. Ha még nem szerepel, beszúrunk egy új bejegyzést a kapott kulccsal és eggyel, mint értékkel. Növeljük továbbá eggyel minden iterációban a sorok számolására szolgáló változó értékét. A *terminatePartial* függvényre nincs szükség, egyszerűen továbbadja a már meglévő állapotot a *merge*-nek.

Merge

A *merge* bemenetként minden esetben a *terminatePartial* visszatérési értékét kapja, vagyis az adott *iterate* által előállított és feltöltött állapotváltozót. Ebben az esetben a *merge* egészen triviális feladatot végez: végigolvassa a Map kulcs-érték párjait, és ha már szerepel a globális állapotban az adott kulcs, akkor a hozzátartozó értéket megnöveli a lokálisan számolt értékkel, ha még nem, akkor beszúrja új bejegyzésként a kapott párost. Ezen felül a sorok számát egyszerűen meg kell növelni a bemenetként kapott értékkel.

Terminate

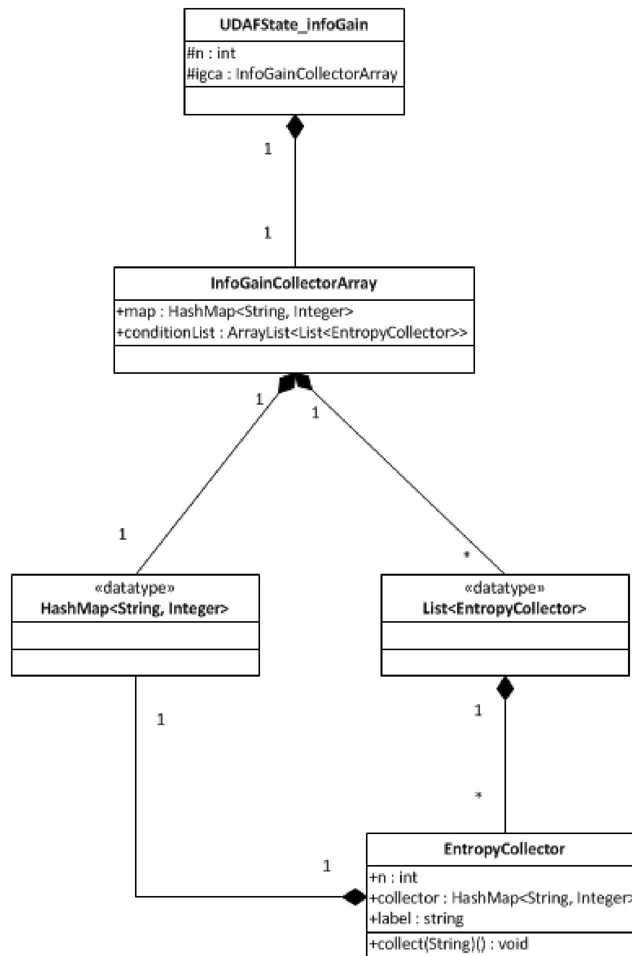
Végül elő kell állítani az egyes értékek előfordulásának valószínűségét. Ez legegyszerűbben úgy kivitelezhető, ha végigiterálunk a Map-en és egy Double értékeket tartalmazó listába beszúrjuk az adott bejegyzés értékének és a sorok számát tartalmazó változó hányadosát. Végül végig kell iterálni a valószínűségeket tartalmazó listán, minden értékre kiszámolva a $p \log_2 p$ szorzatot, a kapott értékeket összeadni, majd a képletnek megfelelően az összeg ellentettjét venni.

4.2.2 Feltételes entrópia

A feltételes entrópia implementálása már nagyobb kihívásnak bizonyult, hiszen itt külön-külön kell gyűjteni a célváltozó értékeinek előfordulását a feltétel minden különböző értékére, majd ezekre (5) szerint entrópiát és előfordulási valószínűséget kell számolni.

Állapot

Az algoritmus implementálásakor az első nem triviális feladat az állapotot tartalmazó osztály struktúrájának a megtervezése volt (5. ábra). A legfelső szinten található állapotosztály egyszerre szolgál az entrópiához és a feltételes entrópiához tartozó részeredmények gyűjtésére; itt most csak az utóbbi kerül bemutatásra. Az InfoGainCollectorArray osztály által tartalmazott conditionList kétszintű listát és az entrópiához szükséges HashMap-et logikailag nem kellett volna külön osztályba tenni, ám ha mindkettő a legfelső állapot része, a Hive futási idejű kivételt dob, aminek oka, hogy nem tud kétféle kollekciót kezelni az állapoton belül. A conditionList-re azért van szükség, mert a program minden, az eljárás bemeneteként megadott feltételre kiszámolja az információtartalmat. Ezért gyakorlatilag a teljes algoritmust ciklikusan végre kell hajtani az összes bemeneti feltételváltozóra. Egy változó esetén – tehát ha csak a bemeneti feltételváltozó-tömb egyik elemét vizsgáljuk – szükség van egy listára, amely a feltételben előforduló összes érték számára tartalmaz egy EntropyCollector objektumot. Ennek három tagváltozója van: az n az érték előfordulásainak számát, a label magát az értéket tartalmazza, míg a HashMap itt is az entrópia számolásának eszköze, vagyis gyűjti a célváltozó különböző értékeit és az előfordulásuk számát ott, ahol a feltételben a label által tárolt érték szerepel.



5. ábra Az információtartalom-számítás osztálydiagramja

Iterate

A függvény bemenete a célváltozó, valamint egy meghatározatlan hosszúságú feltételváltozó-tömb (Java-ban „*String… conditions*” szintaxis). Egy sor beolvasásakor végigiterálunk a bemeneti feltételeket tartalmazó tömbön. Amennyiben az aktuális mapperen ez az *iterate* első futása (tehát a sorszámláló változó értéke 0), akkor létrehozunk a feltételváltozónak egy EntropyCollector listát. Egyébként ha már létezik, a listában megkeressük az aktuális feltétel bemenetként kapott értékét, mint EntropyCollector label-t. Ha nincs még létrehozva ilyen értékkel EntropyCollector, akkor ezt megtesszük, beszúrva a célváltozó adott értékét (y_i), mint az első Map bejegyzést. Ha már van, akkor egy *collect* függvény beszúrja a Map-be y_i-t, vagy ha már létezik, inkrementálja a számlálóját.

Merge

A *merge* megvalósítása elméletben nem tűnik bonyolultnak, ám a függvény és a Hive kapcsolatának pontos dokumentációja nélkül a beágyazott listák és HashMap-ek megfelelő kezelése nem bizonyult egyszerű feladatnak. Elsőként itt is egy ciklust kell indítani, mely a feltételváltozókon iterál végig. Minden feltétel esetén egy belső ciklusban végig kell járni az egyes EntropyCollector-okat és megvizsgálni, hogy a globális állapotot tartalmazó objektumban létezik-e ugyanahhoz a feltételhez ugyanaz a label. Ha igen, végig kell iterálni a HashMap bejegyzésein és az entrópiánál leírtakhoz hasonlóan egybeolvasztani a kulcs-érték párokat valamint meg kell növelni a sorszámlálót. Ha még nem létezik a címke a globális EntropyCollector-ok között, akkor ezt létre kell hozni és egyenként át kell másolni a lokális objektum mezőit a globális állapot megfelelő részébe. A függvény írásakor elkövettem azt a hibát, hogy referencia szerint másoltam a globális listába a lokális EntropyCollectort. Ez persze hibás eredményt adott, hiszen a Hive a *merge* lefutása után a lokális állapot már nem tartja meg, így azt begyűjtheti a Garbage Collector.

Terminate

A *terminate* indulásakor tehát össze van gyűjtve az összes feltételre, hogy melyik értékből mennyi szerepelt, illetve hogy ezeken a helyeken a célváltozónak milyen értékei voltak. Itt is végig kell iterálni az egyes feltételeken és azon belül az oszlop egyes EntropyCollector objektumain. Egy listában minden ilyenhez ki kell számolni a HashMap alapján a célváltozó entrópiáját (a képlet $H(Y|X = v_j)$ része), majd ezt meg kell szorozni a tárolt adatok alapján egyszerű osztással számítható $P(X = v_j)$ valószínűséggel. Ezek az eredmények egy listába kerülnek, majd az elemeknek a feltétel összes különböző értékére történő összegezése után az entrópia felhasználásával kiszámításra kerül minden feltételre az információtartalom. Végül a végrehajtás az ezeket tartalmazó listával visszatér a Hive-hoz.

Eredetileg tervbe volt véve, hogy nagyság szerint legyen rendezve a visszaadott lista, ám a Hive nem támogatja az attribútum nevek átadását az UDF bemenetére, így nem lehetne azonosítani, hogy melyik kapott érték melyik jellemzőhöz tartozik. Természetesen a visszaadott értékek sorrendje egyezik a bemenetre adott feltételek sorrendjével, így kevés jellemzőre akár szemmel, sokra pedig például egy egyszerű táblázatkezelő segítségével sorrendezhetőek az eredmények.

4.3 A korrelációs mátrix számítása

A korrelációs mátrix számítása három részből áll: az előfeldolgozás során az **L** és a **Q** mátrixokat kell előállítani, majd ezeket felhasználva (8) alapján ki kell számolni **R** elemeit. Ez az algoritmus „kompatibilis” a MapReduce technikával, így a Hive UDAF felépítésével is: a Map oldalon kerül előállításra a két segédmátrix, melyek már elférnek a memóriában, így a Reduce oldalon történhetnek a további számítások.

Állapot

Az állapotváltozónak azokat az objektumokat kell tartalmaznia, amelyek az előfeldolgozás során állnak elő, vagyis az **L**-et, a **Q**-t és a sorokat számláló n-et. Ezek közül n primitív int, **L** beépített Java ArrayList<Double> típusú (hiszen ez tulajdonképpen egy sorvektor), míg **Q** számára létre kellett hozni egy SquareMatrix osztályt, mely egy négyzetes mátrixot reprezentál (6. ábra).

SquareMatrix
-mx : ArrayList<List<Double>> -size : int
+initMatrix(int) : void +get(int,int) : double +set(int,int,double) : void +add(int,int,double) : void +getSize() : int +addPartial(Double[]) : void +getMx() : ArrayList<List<Double>> +addMatrix(SquareMatrix) : void

6. ábra A SquareMatrix osztály

Az ábra nagyrészt önmagyarázó: a *get* és *set* a megadott indexű elemet adják vissza; az *initMatrix* egy megadott méretű kétdimenziós listát hoz létre; az *add* hozzáadja a bemenetként kapott értéket a meghatározott elemhez és ugyanezt a feladatot végzi az *addMatrix* egy teljes mátrixra. Az *addPartial* függvény egy bemeneti oszlopvektort szoroz össze egy beágyazott ciklus segítségével a saját transzponáltjával és az eredményt az *add* függvény meghívásával hozzáadja a mátrixhoz (ha a bemeneti oszlopvektor mérete nem egyezik a mátrix méretével, kivétel dobódik).

Iterate

Mint az feljebb már említésre került, a mapperek – tehát az *iterate* – feladata az **L** és **Q** mátrixok előállítása. A függvény bemenete egy meghatározatlan méretű Double tömb, ez tartalmazza az **X** mátrix adott sorát. A végrehajtás során először az *initMatrix* függvény hívódik meg a **Q** tárolására készített, kezdetben csupa nullát tartalmazó

SquareMatrix-on (ezt a lépést nem lehet az *iterate* előtt futó *init* függvényben elvégezni, hiszen akkor még nem lehet tudni, hogy mekkora mátrixra lesz szükség). Ezután az **L**-t tároló listához elemenként hozzáadjuk a bemeneti tömb elemeit, majd meghívjuk a fent bemutatott *addPartial* függvényt, ami a **Q** mátrixot frissíti a kapott sorral. Végül az n változót egyszerűen inkrementáljuk minden iterációban.

Merge

A *merge* feladata ebben az esetben nem bonyolult, tulajdonképpen hasonló műveleteket végez, mint az *iterate*. A lokális **L** mátrixon végigiterálva hozzáadja az elemeket a globális **L**-t tartalmazó listához. A lokális **Q**-t pedig egyszerűen átadjuk paraméterként a globális **Q** *addMatrix* függvényének.

Terminate

A *terminate* már dolgozhat a teljes **X** mátrix feldolgozásával előállt segédmátrixokkal. Az **R** korrelációs mátrix számára létrehozunk egy újabb, a **Q**-hoz hasonlóan d×d méretű SquareMatrix példányt. Ezután egy beágyazott ciklussal **R** minden elemére kiszámoljuk (8) alapján $\rho_{i,j}$ értékét (ez nem okoz különösebb nehézséget, mivel a SquareMatrix *get* és *set* függvényei alkalmazhatóak). Az így feltöltött kétdimenziós listával a végrehajtás visszatér a Hive-hoz.

Az eljárás első verziójában még valóban minden elemre ki lett számolva a korrelációs együttható értéke, azonban – a főátlóban csupa egyest tartalmazó szimmetrikus mátrixról lévén szó – ezt elég csak az $i<j$ elemekre elvégezni, majd egyszerűen $\rho_{i,j}$ értékét beszúrni $\rho_{j,i}$ helyére is.

4.4 Az eredmények validálása

Az információtartalom-számítás eredményének validálására alkalmasnak bizonyult a már említett RapidMiner adatbányász program. Ebben a különböző számításokat végző folyamatok grafikus módon, építőelemekből rakhatóak össze. A validálást végző folyamat a következőképpen működik: először egy „*Read CSV*” komponens beolvassa az adatokat tartalmazó szövegfájlt, majd a „*Set Role*” beállítja kiválasztott attribútumot célváltozónak. Végül a „*Weight by Information Gain*” elem kiszámítja az információtartalmat. Ahhoz, hogy a *merge* függvény helyességét is tesztelni lehessen, akkora bemeneti adatfájlt kellett használni, amire a Hadoop már legalább két map taszkot indít (megjegyzés: a speciálisan Big Data-t feldolgozó

adatbányászati eszközök létjogosultságát az is mutatja, hogy a RapidMiner a célnak megfelelő méretű, 600MB-os CSV fájl esetén 3 GB memóriát használva éppen le tudott futni). A kétféle kapott eredmény az implementáció első verziója esetén még különbözött, a *merge* függvény módosítása után azonban 3 tizedes jegyig megegyezett.

A korrelációs mátrix számítására több eszköz is rendelkezésre áll. A Matlab beépített *corrcoef(X)* függvénye egy mátrixot vár bemenetként és a korrelációs mátrix-szal tér vissza[12]; a Hive-ban elérhető *corr(X,Y)* függvény pedig két változóra számol korrelációt. A fejlesztés közben utóbbi volt hasznosabb, hiszen egyszerűen lekérdezhető volt a mátrix egy-egy eleme, amikor pedig ez már helyes eredményt adott, a teljes eredményhalmaz Matlab-bal történő validálására is sor került.

[12] Matlab corrcoef eljárás: http://www.mathworks.com/help/matlab/ref/corrcoef.html

5 Mérések

Az előző fejezetekben bemutatott mérőszámokat már több évtizede használják a kapcsolódó szakterületeken. A készített implementációknak tehát akkor lehet gyakorlati hozadéka, ha a meglévő módszereknél a megcélzott adatmennyiségre számottevően hatékonyabbak, azaz rövidebb végrehajtási idővel rendelkeznek. Éppen ezért a dolgozatban kiemelt figyelmet kap az algoritmusok futásidejének részletekbe menő mérése. Az ezzel kapcsolatos módszerek és eredmények kerülnek ismertetésre a fejezetben.

5.1 Mérési környezet

A fejlesztés kezdeti szakaszában az algoritmusok futtatása a Cloudera által ingyenesen elérhetővé tett virtuális gépen[13] történt. Ez már előre konfiguráltan tartalmazza többek között a Hadoop és Hive rendszereket, így kezdetben kényelmes választás volt a felületek megismerésére, a Java kódban előforduló hibák keresésére illetve az eredmények validálására. Mivel azonban itt a „fürt” csak egyetlen csomópontot tartalmaz, releváns mérések végzésére alkalmatlan. A fejezetben közölt futási idők és grafikonok az egyetemen belüli klaszteren való végrehajtás alapján születtek. Ezt a rendszert mutatja be az első alfejezet, míg a továbbiakban a mérések paraméterezése és automatizálása kerül ismertetésre.

5.1.1 Hardvertulajdonságok és konfigurációs adatok

A pontos hardverjellemzők és konfigurációs adatok megismerése kulcsfontosságú szerepet játszik a mérési eredmények elemzésekor és a mások által végzett mérésekkel való összehasonlításkor.

A használt klasztert a TMIT tanszék adatbányászat csoportja bocsátotta rendelkezésemre, melyen a Hadoop és Hive rendszerek telepítve és konfigurálva vannak. A fürtben 6 szerver található, mindegyik azonos tulajdonságokkal. Az 1.

[13]Cloudera virtuális gép: http://www.cloudera.com/content/cloudera-content/cloudera-docs/DemoVMs/Cloudera-QuickStart-VM/cloudera_quickstart_vm.html

táblázat a klaszterben található egy csomópont jellemzőit, a 2. táblázat pedig a Hadoop és a Hive konfigurációs beállításait foglalja össze.

Processzorok száma	2
Egy processzorban lévő magok száma	4
Processzor frekvencia	1.80 GHz
Fizikai memória (RAM)	48 GB
Operációs rendszer	Ubuntu 12.04

1. táblázat A klaszterben található szerverek jellemzői

Tárhely összesen	42.47 TB
Data Node-ok száma	6 db
Blokkok átlagos száma	2761 db/Data Node
Map taszk kapacitás	8 db/TaskTracker
Reduce taszk kapacitás	4 db/TaskTracker
Blokkméret	128 MB
Replikátumok	3 db/blokk

2. táblázat Hadoop konfigurációs adatok

5.1.2 A mérésekhez használt mintaadatok

A méréseket egy ingyenesen elérhető adathalmazon végeztem, mely különböző kézzel írt számjegyeket ábrázoló, 28×28 képpont méretű képek pixeleinek értékét (0-255) tartalmazza[14]. A bemeneti CSV fájl a következő felépítésű:

label	pixel0	…	$pixel_i$	$pixel_{i+1}$	…	pixel783
2	0	…	123	96	…	0
7	0	…	56	231	…	0
…	…	…	…	…	...	…

3. táblázat A használt adathalmaz struktúrája

[14] Mintaadatok: http://www.kaggle.com/c/digit-recognizer/data

Látható, hogy az adattábla egy címkét és további 784 jellemzőt tartalmaz (d=784), amely elegendően sok ahhoz, hogy kitűnhessenek az UDAF implementáció előnyei. A sorok száma a fenti tanulóhalmazban 42000 (n=42000), ami így 73.2 MB-os fájlt eredményez. Ez természetesen még bőven kezelhető memóriában dolgozó módszerekkel, így az egyes mérésekhez az adattábla sorainak a számát folyamatosan növelve egyre nagyobb halmazon futtattam a lekérdezéseket. A tábla fokozatosan a kezdeti méret 700-szorosáig, azaz nagyjából 50 GB méretig lett növelve az eredeti adatsorok újbóli beszúrásával

5.1.3 A mérések automatizálása

A két UDAF és a konkurens függvények futási idejének méréséhez a dimenziók és a sorok számának folyamatos növelése miatt nagyszámú lekérdezésre volt szükség. Mivel a klaszter parancssoron keresztül férhető hozzá, kézenfekvő választás volt a Unix shell szkriptek használata. Ezeket egy Java-ban írt rövid program generálta, melynek paraméterül meg lehet adni a futtatni kívánt függvényeket tartalmazó JAR fájl és a visszaadott eredmények tárolására szolgáló állomány elérési útját, az attribútumszám-intervallumot és lépésközt, valamint a használni kívánt adattáblákat (a táblák elnevezési konvenciója azon alapul, hogy hányszorosa az adott tábla sorainak a száma a kezdeti 42000 sorhoz képest - így egyszerűen lehetett a táblák nevét is generálni).

A szkript az aktuális mérés paramétereinek naplózása után a „*hive -e*” parancsot használva (mellyel Unix parancssorból adhatunk a Hive számára SQL utasításokat) betölti a megfelelő JAR fájlt és létrehozza ideiglenes függvényként a mérendő UDAF-ot. Ezután egy lekérdezésben meghívja a kívánt függvényt, végül a Hive által adott kimeneti adatokból a „*grep*” paranccsal kiszűri a futási időt és beszúrja egy szövegfájlba.

5.2 Mérési eredmények

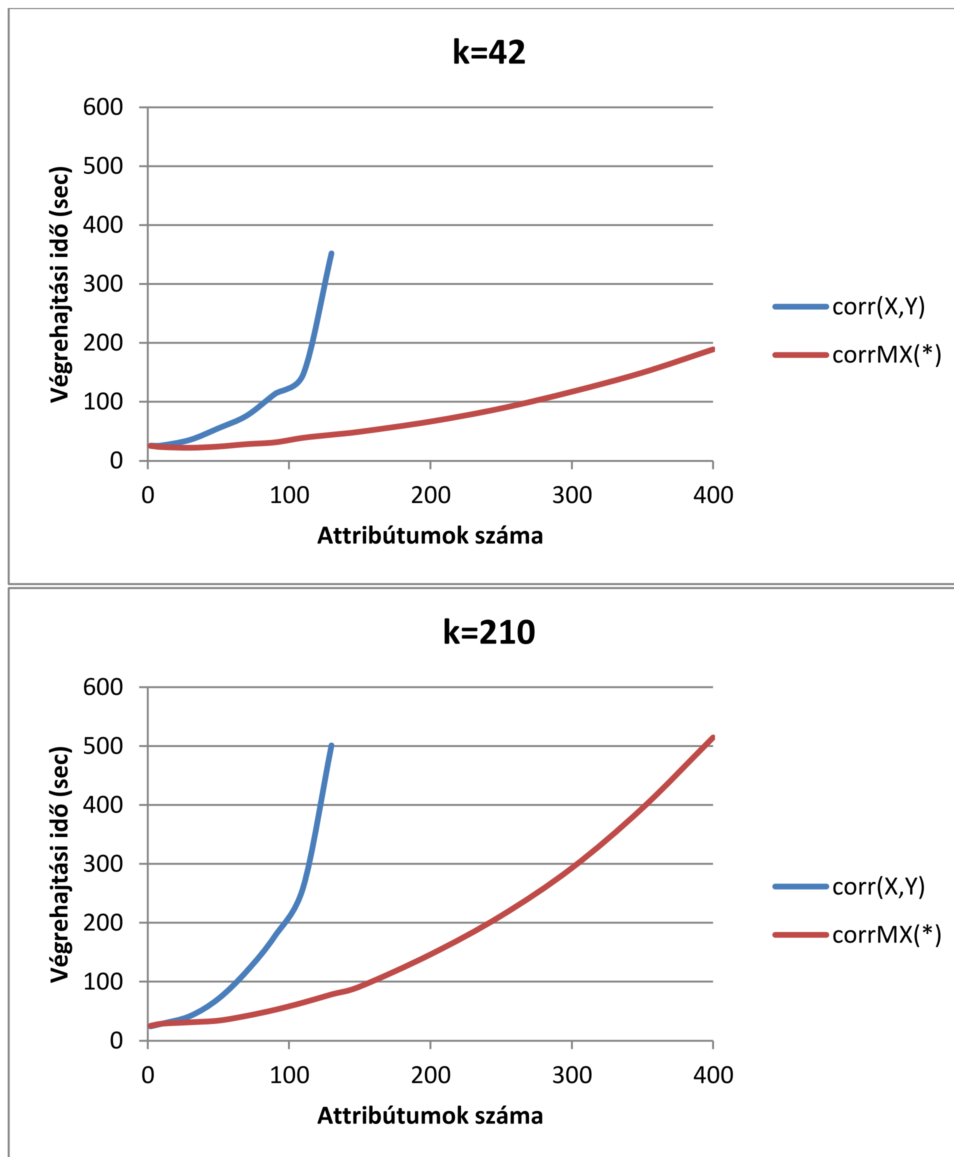
A fenti generátor megvalósítása után a paraméterezés feladata következett. Kétféle mérést volt érdemes elvégezni: az egyik csoport célja a futási idő összehasonlítása volt egy hasonló célú implementációéval, ennek eredményei az 5.2.1-es fejezetben kerülnek bemutatásra; a másik csoport az implementációk skálázódását volt hívatott tesztelni, ez olvasható az 5.2.2-ben. A grafikonok értelmezését

megkönnyítendő néhol szerepel a k változó, mely a mért adattáblában tárolt sorok ezredrészét jelöli.

5.2.1 Összehasonlítás más implementációkkal

Mint az a 3.4.1-es fejezetben látható volt, az információtartalom-számításhoz nem érhető el olyan nyílt implementáció, mely hatékonyan alkalmazható Big Data környezetben. Éppen ezért csak a korrelációs mátrix számításához készültek ilyen mérések.

Amivel közvetlenül összehasonlítható az általam implementált algoritmus, az a Hive beépített *corr(X,Y)* függvénye. A 7. ábrán látható a két függvény futásideje azonos méretű táblák esetén. Körülbelül 140 oszlopú korrelációs mátrixot tudott még kiszámolni a Hive könyvtári függvény (természetesen csak a szimmetrikus mátrix egyik felének lekérdezése történt meg), ennél több attribútumra a Java OutOfMemory kivételt dobott. Az implementált UDAF (*corrMX(*)* függvény) az összes bemenetként kapott adattáblán lefutott. Ezen felül azoknál a lekérdezéseknél, ahol tudott kimenetet produkálni a *corr(X,Y)*, tulajdonképpen minden esetben a *corrMX(*)* futott le gyorsabban. Érdekes eredmény, hogy két bemeneti változó esetén nagyjából azonosan 25 másodperc alatt jutottak eredményre.



7. ábra A korrelációt számoló Hive függvények összehasonlítása

A főkomponens-analízist számos hagyományos platformra implementálták. A JAMA nevű Java csomag[15] mátrixműveleteket valósít meg Java nyelven, természetesen memóriában való futtatással. Elérhető továbbá PCA implementáció az R nyelvhez, mely kifejezetten statisztikai mutatók számítására lett fejlesztve. Végül érdemes szót ejteni a C. Ordonez által [16]-ban bemutatott SQL és UDF alapú implementációról, melyben nagy hangsúly van fektetve a memóriahasználat alacsonyan tartására és a futtatás sebességének optimalizálására. Bár az implementált corrMX(*) függvényt a memóriában dolgozó algoritmusokkal nincs értelme összehasonlítani, néhány

[15] http://math.nist.gov/javanumerics/jama/

tanulságos eredményt érdemes megemlíteni [16]-ból. Az alábbi táblázat egy tisztán MySQL UDF alapú és a szerző által javasolt SQL/UDF hibrid megoldás futási idejét hasonlítja össze. Sajnos saját méréseket az SQL/UDF függvény hiányában nem tudtam végezni, ezért a szerző által végzett mérésekre hagyatkozom. A használt szerver egy kétmagos, 2.6GHz-es processzorral és 4GB memóriával rendelkezett.

n×1000	d	SQL/UDF[16]	UDF
10	30	2	3
10	50	21	16
10	70	59	34
100	10	1	2
100	20	7	7
100	30	16	16
100	50	53	47
1000	20	33	72
1000	40	154	261

4. táblázat Egyéb módszerek korrelációs mátrix számítására [16]

Bár a lekérdezéseket futtató számítógépek közötti különbség nem elhanyagolható, nagyságrendileg összevethető az implementált *corrMx(*)* a táblázatban található értékekkel. Annak ellenére, hogy ekkora adatmennyiségnél még nincs szükség Big Data módszerekre, a Hive UDAF egymillió rekord és 40 attribútum esetén (a táblázat utolsó sora) kb. 30 másodperc alatt hozott eredményt, tehát – a szerverek közti különbségeket is belekalkulálva – nem marad el az implementált algoritmus a fenti eredmények mögött.

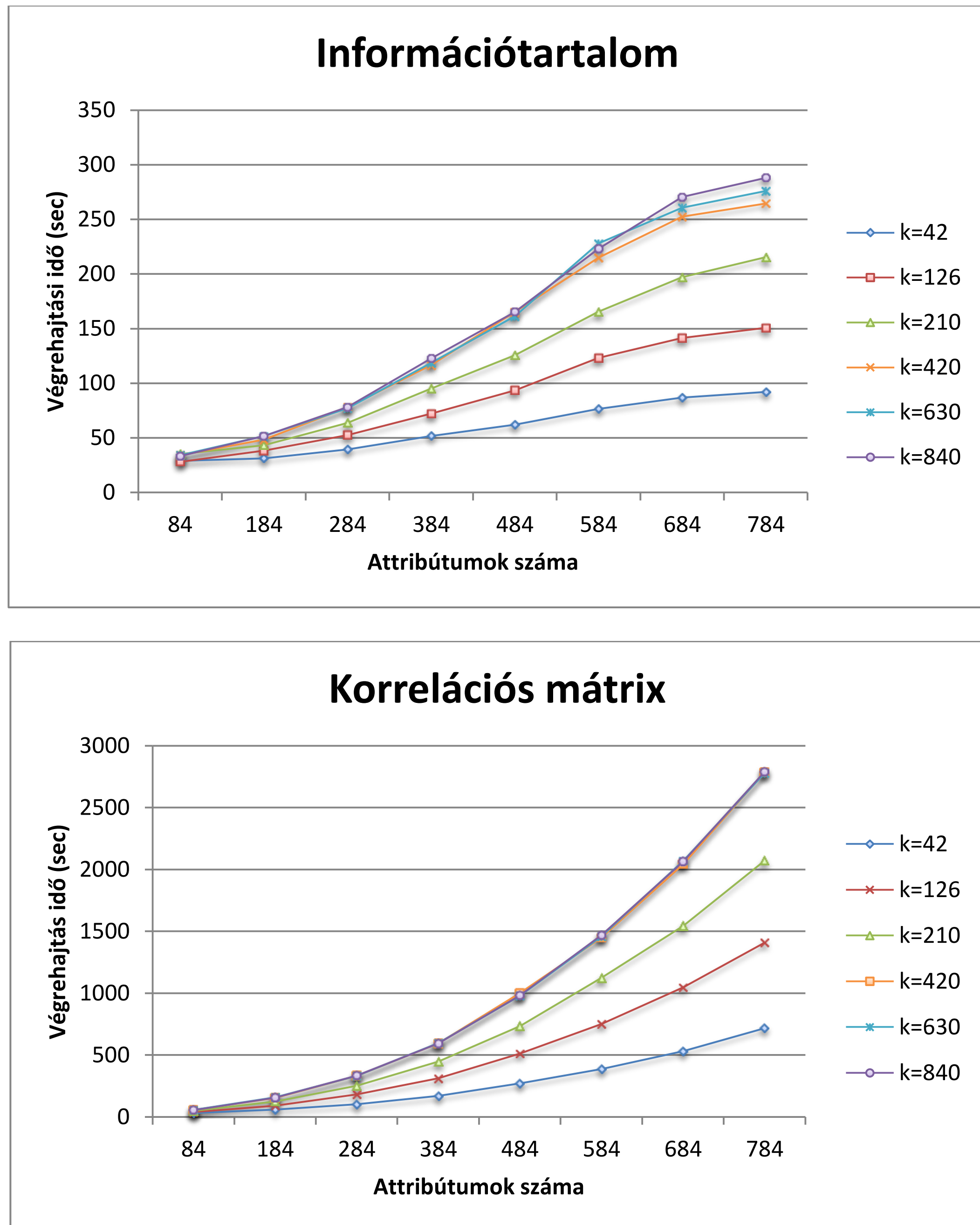
5.2.2 Skálázódási mérések

A következő feladat az implementációk skálázhatóságának mérése volt. Mivel az eljárások kifejezetten nagy adatmennyiségre lettek kifejlesztve, nagyon fontos tudni, hogy hogyan növekszik a futási idő az attribútumok és sorok számának növelésekor.

Az oszlopok száma szerinti skálázódás mérésére az eljárások egyre növekvő attribútum számmal kerültek meghívásra, d=84-től kezdve százas lépésközzel d=784-ig. Hogy az oszlopok száma szerinti skálázódás és a sorok száma közötti összefüggésre is

[16] A kétféle mutatott módszerből (horizontális és vertikális aggregáció) a gyorsabb horizontális aggregáció eredménye van megadva. A végrehajtási idők másodpercben értendőek.

fény derüljön, a mérést több különböző nagyságú táblán is lefuttattam. Ennek eredményei láthatóak az alábbi ábrán.

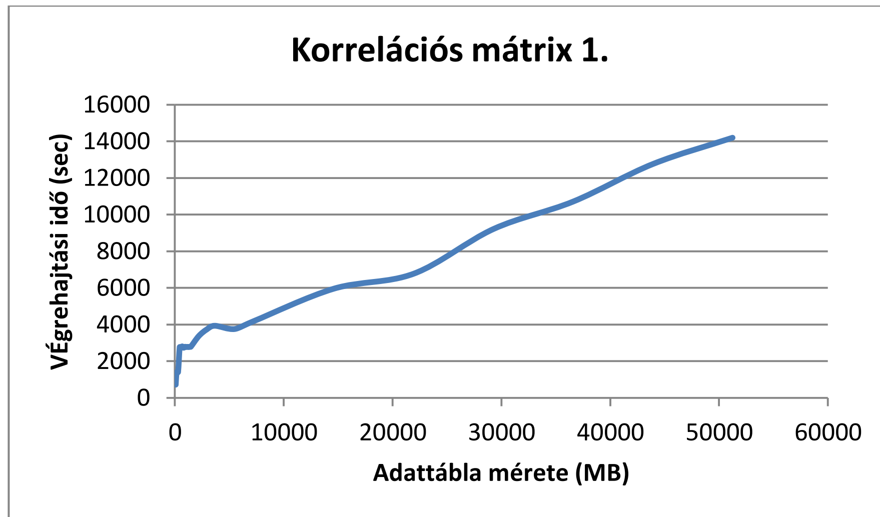


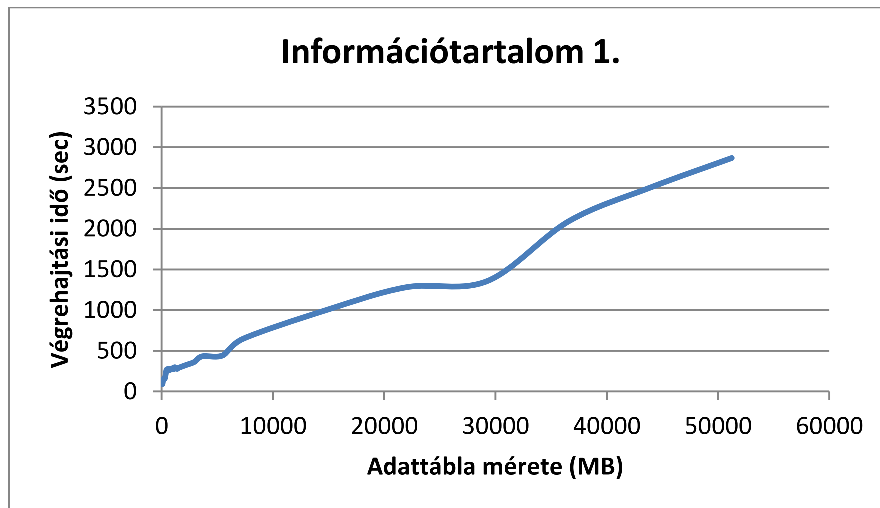
8. ábra Az eljárások skálázódása a bemeneti dimenziószám függvényében

Az első diagramon látható, hogy információtartalom-számítást végző UDAF esetében az attribútum szám növelésével középen nagyjából lineárisan, míg a görbe egyre a mért intervallum elején és végén kisebb meredekséggel növekszik. A lineáris növekedés megegyezik az elvárt viselkedéssel, hiszen egy attribútum hozzáadásával a Map és a Reduce oldalon is eggyel többször fut le a külső ciklus. A széleken

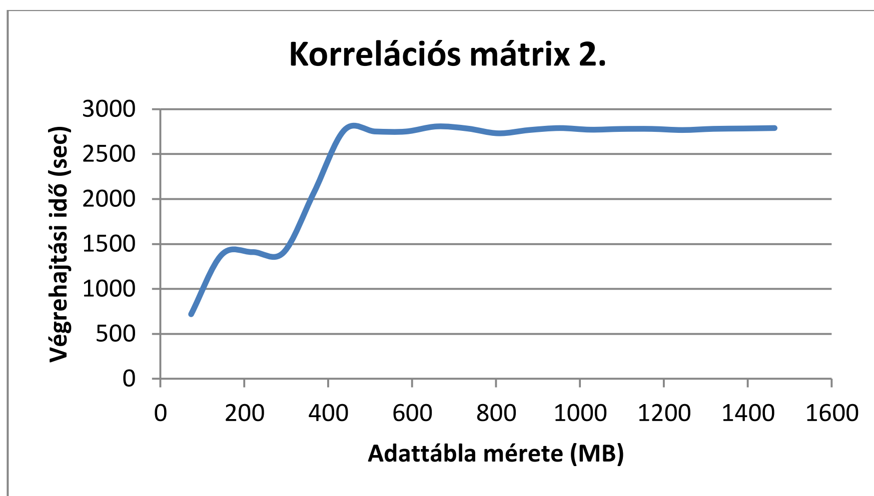
tapasztalható kisebb meredekség valószínűleg annak köszönhető, hogy a mintaadat első néhány és utolsó néhány attribútuma azonosan 0, vagy csak nagyon kevés különböző értéket tartalmaz, így a Map és a Reduce között áramló adatmennyiség jóval kisebb. Kisebb hálózati forgalom pedig rövidebb végrehajtási időt eredményez. A korrelációs mátrix számításának futásideje nagyjából exponenciálisan nő a dimenziószám növelésével. Ez várható volt, hiszen egy négyzetes mátrix mérete az oszlopok számával nyilvánvalóan négyzetesen nő. Megfigyelhető továbbá, hogy a k paraméter növelésével egyre kevésbé térnek el egymástól az eredmények. Az alsó diagramon már nem is tapasztalható jelentős eltérés a futásidőben, hiába nőtt a sorok száma 420 ezerről 840 ezerre. Ennek oka alább kerül ismertetésre.

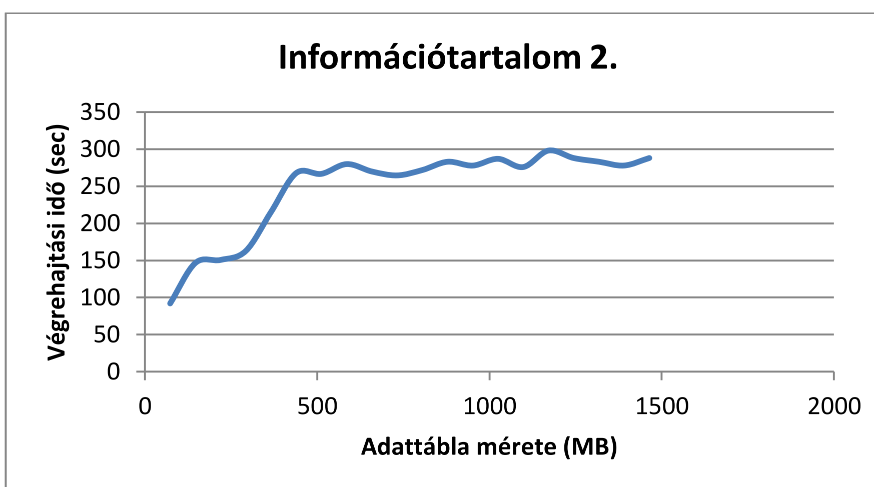
A sorok számának növeléséhez kapcsolódó mérések eredményei a 9. és a 10. ábrákon láthatóak. A 9. ábra diagramjai a teljes lemért intervallumon kapott eredményeket mutatják. Látható, hogy hosszú távon a sorok növelésével egyenes arányosan nő a végrehajtási idő mindkét eljárás esetében. A növekedés egyértelműen szublineáris, a függvény meredeksége nagyjából $\frac{3}{5}$.





9. ábra Az eljárások skálázódása a sorok számának függvényében a teljes intervallumon





10. ábra Az eljárások skálázódása a sorok számának függvényében kis táblákra

A 10. ábra diagramjai a kisebb táblákon mért eredményeket nagyítják fel. A kezdetben tapasztalt lépcsőzetesség és az utána következő nagyjából konstans végrehajtási idő a MapReduce technika sajátossága. A Hadoop a táblamérettől függően valahány mapperhez rendeli hozzá az adott feladatot. A könnyebb érthetőség kedvéért az alábbi táblázat a különböző méretű táblákhoz használt Map slotok számát tartalmazza.

méret (MB)	Szükséges mapperek
73	1
146	1
220	2
293	2
366	2
439	2
512	3

5. táblázat Az egyes táblákhoz szükséges Map slotok száma

Látható, hogy az első és a második táblán futtatott lekérdezéseket is 1 mapper hajtja végre. A meredek növekedés a grafikon elején azért van, mert egy egységnek kétszer annyi sort kell szekvenciálisan beolvasnia a második tábla esetén, a végrehajtási idő tehát bizonyosan nőni fog. A második és harmadik tábla esetén a futási idő nagyjából megegyezik. Ennek oka, hogy a kettő közti különbséget egy második mapper hajtja végre párhuzamosan, így – mivel az ilyenkor meghívott *merge* függvény nem fut számottevő ideig – a „kritikus út" továbbra is a kétszer annyi adatot feldolgozó első mapper futási ideje. A következő lépésben a két mapper már azonos terhelést kap, a végrehajtási idő nem változik, utána az egyik ismét többet, tehát megint ugrást tapasztalunk. A következő pontban – az 5. és a 6. tábla esetén – azonos végrehajtási időt várnánk, hisz ismét mindkét mapper azonos adatmennyiséget kaphatna. Ennek ellenére a Hadoop az egyik mappernek kétszer annyi adatot ad, mint a másiknak, ezért tapasztalunk itt is ugrást. Ez valószínűleg azért van, mert a Hadoop elsődleges célja az adatblokkok és mapperek összerendelésekor, hogy minél kisebb hálózati forgalmat generáljon. A 6. tábla adatblokkjai 4:2 arányban vannak eltárolva a két mappert futtató csomópontokon, így a hálózati adatmozgatás helyett a Hadoop a „kisebb rosszat" választja, és inkább aránytalanul terheli le a mappereket.

A használt klaszterben 6 szerver van és a konfigurációnak megfelelően mindegyikben 8 Map slot található, így összesen 48 hellyel tud gazdálkodni a Hadoop. Előzetesen helyénvalónak tűnt az a feltételezés, hogy az összes hely betelte után valamiféle ugrás fog következni a végrehajtási időben, hiszen néhány feladatnak meg kell várni egy előző befejeződését. Ez valóban igaz, ám mivel nem egyszerre fejeződik be az összes futtatás, ezért a Hadoop általában talál az új feladatoknak üres slot-ot. A végrehajtási idő növekedése tehát nem ugrásszerű.

5.2.3 A mérések összegzése

Az összehasonlító mérések egyértelműen azt mutatják, hogy a korrelációs mátrix számítására olyan eljárást sikerült létrehozni, amely a Hive beépített függvényét gyakorlatilag bármely bemenetre felülmúlja. Ezen felül az eddigi, hagyományos adatbázisokon implementált UDF-ekkel is felveszi a versenyt már olyan adatokra is, amelyek nem esnek bele a Big Data fogalmába. A skálázhatósági mérések során tapasztalt tendencia pedig bebizonyította mindkét függvényre, hogy a megcélzott

adatmennyiséget hatékonyan, a sorok számával egyenesen arányos futásidő-növekedéssel képes feldolgozni.

6 Továbbfejlesztési lehetőségek, jövőbeli tervek

Az előző fejezetben látható volt, hogy az implementált algoritmusok elegendően hatékonyak ahhoz, hogy megoldják a nagy adatokon való statisztikai mutatók előállításának problémáját. Önmagában azonban ezek futtatása csak kevés esetben állítja elő a végső eredményt, hiszen mind a korrelációs mátrix, mind az információtartalom számítása egy nagyobb folyamat építőeleme. Hogy valóban a hétköznapi feladatok során is alkalmazni lehessen őket, érdemes egy olyan eszközt megvalósítani, mely – akár memóriában futó módszerekkel – képes automatikusan felhasználni az előállított eredményeket a származtatott értékek előállítására. Az információtartalom esetén a kapott értékeket például fel lehet használni automatikus osztályozásra egy döntési fa alapján. A korrelációs mátrix számítását pedig érdemes lenne egyrészt a teljes PCA elvégzésére, másrészt a 3.2.2-ben bemutatott feature ranking mutató kiszámítására automatikusan felhasználni. Ez utóbbinál az említett heurisztika implementálása jelenthet további feladatot. Szintén a feature ranking területén maradva érdemes lenne implementálni az említett optimális algoritmusokat (pl. branch and bound) és lemérni, hogy van-e létjogosultsága egy Hadoop alapú megvalósításnak.

Mivel a Hive elsősorban egy adattárház-réteg, érdemes lenne eltávolodni a jelen dolgozatban megcélzott statisztika területétől és a szokásos analitikus (OLAP) funkciók megvalósítását is átgondolni, mint a lefúrás (drill down), szeletelés (slicing) vagy forgatás (pivoting). Ehhez természetesen ellenőrizni kell, hogy mely műveletek kerültek már hatékonyan megvalósításra Big Data környezetben, de teljes biztonsággal állítható, hogy az üzleti intelligencia területe tartogat még kiaknázatlan lehetőségeket.

7 Összefoglalás

A dolgozat első felében a Hadoop és Hive környezetek bemutatására került sor. A MapReduce paradigma köré épített Hadoop platform a várakozásoknak megfelelően tökéletesen alkalmasnak bizonyult nagyméretű adathalmazok kezelésére. Az egyetlen hátránya a nehezen tanulható és kevésbé intuitív fejlesztői felülete. Erre ad megoldást a Hive rendszer, melynek segítségével a megszokott módon kezelhetőek a Hadoop-on futtatott lekérdezések. A két rendszer leírását tartalmazta a 2. fejezet. A dolgozat következő részében arra kerestem a választ, hogy melyek azok a dimenziócsökkentési eljárások, amelyeket meg lehet és meg érdemes valósítani Big Data környezetben. A választás az információtartalom és a korrelációs mátrix kiszámításának feladatára esett. Ennek elméleti alátámasztása és az algoritmusok felhasználási területei olvashatóak a 3. fejezetben. Az implementáció Hive UDAF-ok segítségével történt. A megvalósításnak a rendszer elosztottságából és a MapReduce technika használatából adódó specialitásai, valamint az implementáció során felmerült nehézségek kerültek bemutatásra a 4. fejezetben. A dolgozat végén, a mérések leírásánál pedig bebizonyosodott, hogy az eddig elérhető hasonló célú megoldásoknál egyértelműen jobban szerepelnek az adott implementációk. A mérések bizonyították továbbá, hogy a sorok számának növelésével hosszú távon a lineárisnál kedvezőbb mértékben nő a végrehajtási idő.

Összegzésként elmondható tehát, hogy a Hive rendszer és a megvalósított algoritmusok beváltották a bevezetőben említett elvárásokat, hiszen életképes alternatívát biztosítanak a nagyméretű adattáblák esetén felmerülő dimenziócsökkentés mindezidáig megoldatlan problémájára.

Irodalomjegyzék

[1] HDFS Architekture, 2013: http://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-hdfs/HdfsDesign.html (2013 október)

[2] Jeffrey Dean, Sanjay Ghemawat: *MapReduce: Simplified Data Processing on Large Clusters* at OSDI'04: Sixth Symposium on Operating System Design and Implementation, San Fransisco, CA, December 2004. http://research.google.com/archive/mapreduce-osdi04.pdf (2013. október)

[3] Hadoop tutorial from Yahoo!: http://developer.yahoo.com/hadoop/tutorial/module4.html (2013. október)

[4] Jason Rutherglen, Dean Wampler, Edward Capriolo: *Programming Hive*; O'Reilly, 2012.

[5] Tom White: *Hadoop: The Definitive Guide*; O'Reilly, 2011.

[6] Marc Holmes: *Simple Hive 'Cheat Sheet' for SQL Users*, 20. Aug. 2013. http://hortonworks.com/blog/hive-cheat-sheet-for-sql-users/ (2013. október)

[7] Jon Natkins: *How to use a SerDe in Apache Hive*; 21. Dec. 2012. http://blog.cloudera.com/blog/2012/12/how-to-use-a-serde-in-apache-hive/ (2013. október)

[8] Daniel McClary: *User Defined Functions in Hive*, 02. Apr. 2013. https://blogs.oracle.com/datawarehousing/entry/user_defined_functions_in_hive (2013. október)

[9] The 5 game changing big data use cases: http://www-01.ibm.com/software/data/bigdata/use-cases.html (2013. október)

[10] Machine learning and Big Data analytics: the perfect marriage: http://www.ngdata.com/machine-learning-and-big-data-analytics-the-perfect-marriage/ (2013. október)

[11] Kan Deng: *OMEGA: on-line memory-based general purpose system classifier*; PhD thesis, 1998. http://www.cs.cmu.edu/~kdeng/thesis/feature.pdf (2013. október)

[12] Anil Jain, Dougles Zongker: *Feature selection: Evaluation, Application and Small Sample Performance*, at IEEE Transactions on Pattern Analysis and Machine Intelligence, February 1997. http://www.cs.msu.edu/~cse802/Papers/JainZongker_FeatureSelectionSmallSamplePerformance.pdf (2013. október)

[13] Mark A. Hall: *Correlation-based Feature Selection for Machine Learning*, PhD thesis, 1999.
http://www.cs.waikato.ac.nz/~mhall/thesis.pdf (2013. október)

[14] Andrew G. Moore: *Information Gain*, oktatási segédanyag, 2003.
http://www.autonlab.org/tutorials/infogain11.pdf (2013. október)

[15] Veres Péter: *Gamma felvillanások spektrális elemzése*, diplomamunka, 2006.
A. Függelék: Főkomponens-analízis
http://www.konkoly.hu/~veres/grb/d1.pdf (2013. október)

[16] Carlos Ordonez, Mario Navas: *Efficient Computation of PCA with SQL* at 2nd Workshop of Data Mining using Matrices and Tensors, 2009.

[17] Carlos Ordonez: *Building statistical models and scoring with UDFs* at ACM SIGMOD International Conference on Management of data, 2007.