Budapesti Műszaki és Gazdaságtudományi Egyetem
Villamosmérnöki és Informatikai Kar
Automatizálási és Alkalmazott Informatikai Tanszék

Adatfeldolgozó algoritmusok skálázódásának vizsgálata Hadoop platformon

Tudományos Diákköri Konferencia dolgozat

Konzulensek:

Dr. Dudás Ákos

Dr. Ekler Péter

Szerző:

Nagy László Bence

Budapest, 2014. október 20.

Kivonat

A mai világban nap, mint nap rengeteg új adat keletkezik. Gondoljunk például egy napi internetforgalomra, az egy nap alatt összegyűjtött időjárási adatokra vagy a banki részvények változásainak adataira. Ezt a temérdek adatot a megfelelő cégek el is tárolják, hiszen a mai adattároló eszközök ára és technológiája ezt költséghatékonyan lehetővé teszi. A cégek számára pedig értékes információkat, mint például a vásárlói szokásokat, összefüggéseket, trendeket, tartalmazzák. Éppen ezért az adatokat fel is szeretnék dolgozni, ami az adathalmaz nagy növekedésével egyre nehezebbé vált az eddigi technológiákkal.

Ma a vállalatok túlnyomó többsége az adatai nagy részét relációs adatbázisokban tárolja. Ez, a már oly rég használt technológia egy bizonyos méretig tökéletesen meg is felel az adatok feldolgozására, azonban van akkorra adatméret, mely fölött már nem lehet gyors adatfeldolgozást biztosítani a segítségével. Ekkora adathalmazok elemzésére manapság az elosztott rendszerek segítségével biztosítanak széles körben elfogadott megoldást.

A Hadoop nyílt forráskódú keretrendszer segítségével lehetővé válik nagyon nagy adathalmaz hatékony feldolgozása is. A platform a HDFS nevű fájlrendszerében hatékonyan tárolja az adatokat és a MapReduce technológia segítségével elosztott módon gyors adatfeldolgozásra képes.

A munkám során egy mások által már többféle technológia segítségével elemzett adathalmazt dolgozok fel. Ezek közt a technológiák közt eddig nem szerepelt a MapReduce technológia. A feladatom során Hadoop környezetben, tehát egy új technológia segítségével dolgozom fel az adathalmazt. Bizonyos adatfeldolgozó algoritmusok hatékonyságát vizsgálom a feldolgozandó adat méretének változtatásával. Ezek között keresek skálázódási szabályokat megfelelő metrikák segítségével és hasonlítom össze a teljesítményüket a megfelelő algoritmusok esetén. Így adom meg, hogy a Hadoop keretrendszer segítségével, milyen méretű adathalmaz, milyen algoritmussal dolgozható fel hatékonyabban.

Abstract

Nowadays, a lot of new data are generated every day. Let us take for example the daily Internet traffic, the collected weather data in a day or changes in bank shares data. This huge amount of data is stored by the respective companies, as the price and technology of today's data storage devices enables it in a cost-effective way. The companies also want to process these data, but it has been getting more difficult with the current technologies beacause of the growth of large data sets.

Today, the vast majority of companies store the large portion of their data in relational databases. This technology has been used for so long until a certain size of dataset and corresponded perfectly to the processing of data. But there is that huge amount of data that can not be processed in a short time. Nowadays, the widely adopted solution for analyzing such data sets is provided by using distributed systems.

The Hadoop open-source framework allows processing very large data sets efficiently. The platform's file system is called HDFS, which stores the data efficiently and the MapReduce technology can process the data in a fast and distributed way.

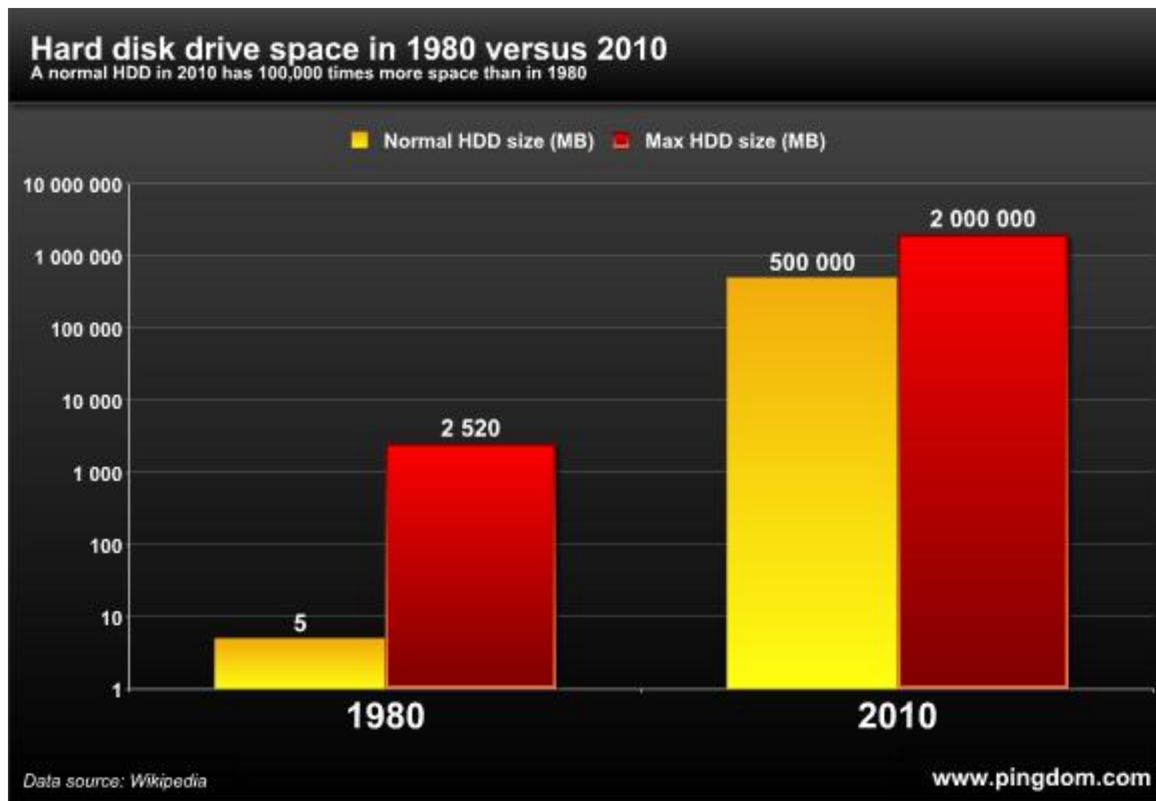
During my work, I process a data set, which has been analyzed by a variety of technologies, except the MapReduce technology. I process the data set using Hadoop environment, so I use a new technology to analyze the same data set. I examine the effectiveness of certain data-processing algorithms while changing the size of the input data. I seek scaling rules between these by using appropriate metrics and I compare their performance for the corresponding algorithms. So that is how I determine, how a certain size of the data set, with a certain algorithm can be processed more efficiently using the Hadoop framework.

Tartalomjegyzék

Kivonat 2
Abstract 3
Tartalomjegyzék 4
1. Bevezetés 5
2. Hadoop 10
2.1 HDFS 10
2.2 MapReduce 13
2.3 Kapcsolódó projektek 15
2.4 Hadoop Streaming 17
2.5 Használati módok 18
3. Feldolgozandó adathalmaz 19
4. Adatelemzés felhőben 22
4.1 C# programok 23
4.2 Algoritmusok 25
4.2.1 Maximum-keresés 25
4.2.2 Átlag-számítás 27
4.2.3 Százalék-számítás 28
4.2.4 Arány-számítás 29
4.3 Skálázódás 31
5. Optimalizáció 35
6. Összefoglalás 41
7. További tervek 42
Irodalomjegyzék 43

1. Bevezetés

Az adatok tárolási lehetőségei nagyban megváltoztak az elmúlt évtizedekben. Napjainkban körülbelül 100.000-szer annyi adatot tárol egy átlagos merevlemez, mint 30 évvel ezelőtt (1. ábra).



1. ábra Hard disk drive tárhely [1]

Ezzel egyetemben az áruk is hatalmasat zuhant. Míg 30 éve átlagosan 1 GB tárhely a 100.000 dollárt is meghaladta, manapság mindössze néhány centbe kerül [1]. Ezzel tehát az adatok nagy mennyiségben tárolására megnyíltak a lehetőségek.

Már az 1970-es években megalkották a relációs adatbázis modellt, ezután megjelentek az első relációs adatbázis kezelő rendszerek és nagymértékben el is terjedtek [2]. Alkalmasak voltak az adatok biztonságos tárolására és analízisére, így a cégek nagyon nagy százalékában használták őket. Sőt, használják ma is (2. ábra).

Mindig voltak olyan új adatbázis modellek, amelyekre azt gondolták megalkotásukkor, hogy hatékonyabbak lehetnek, mint a relációs társaik. Az 1990-es években például az objektum-orientált adatbázis modell került megalkotásra. Az objektum-orientált programnyelvekhez hasonló koncepciót követett, tehát a 4 fő programozási elvre épített [4]. Könnyű volt bennük objektumokat tárolni, ellentétben a relációs adatbázisokkal, amikben erre nincs triviális megoldás [5]. Nagymértékben elterjedni mégsem tudtak,

mert a problémák nagy részét továbbra is relációs adatbázisokkal volt kényelmes megoldani. Később megjelentek például az XML alapú rendszerek, de a relációs adatbázisokat ők sem tudták megközelíteni.

2. ábra Adatbázis kezelő rendszerek eloszlása [3]

2009-ben kezdtek megjelenni az úgynevezett NoSQL adatbázisok. A nevet főként „not only sql"-nek feleltetik meg. A definíciója, melyet 2009 óta formálnak, körülbelül, olyan új generációs adatbázisokra vonatkozik, melyek nem-relációsak, elosztottak, nyílt forráskódúak és horizontálisan skálázhatóak. Ma már legalább 150 NoSQL adatbázis rendszer létezik [6]. Ezek közül a fontosabb csoportok:

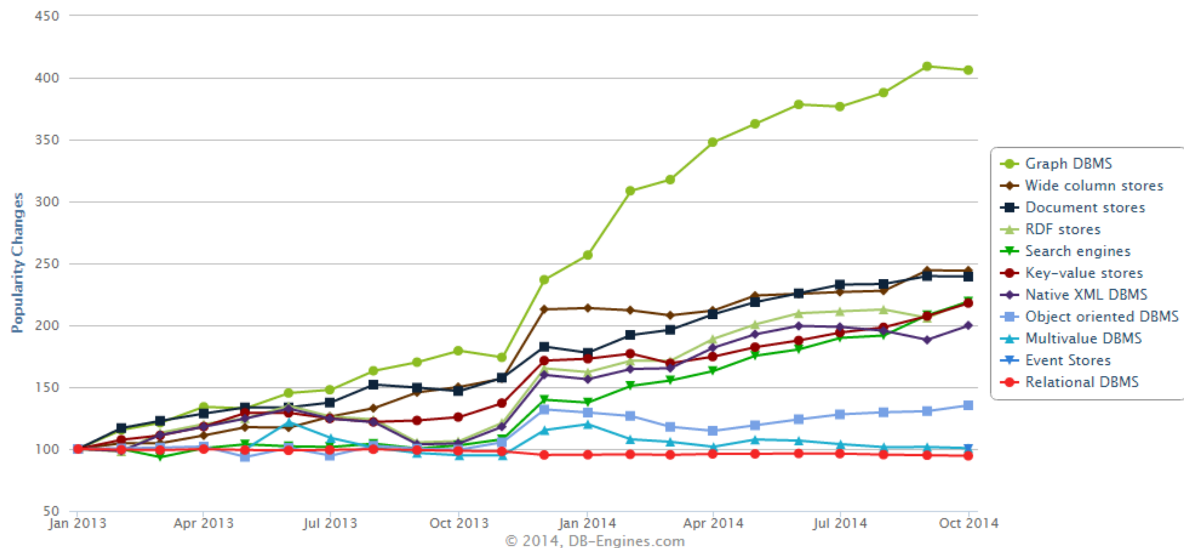
- gráf adatbázisok
- kulcs-érték adatbázisok
- dokumentum adatbázisok
- oszlop adatbázisok

Az, hogy ezek a fajta adatbázisok, mennyire állják majd meg a helyüket és terjednek el hosszú távon a relációs adatbázisokhoz képest az még kérdés, de egyelőre a népszerűségük jelentős növekedést mutat (3. ábra).

A relációs adatbázisok sok év tapasztalati előnnyel rendelkeznek, illetve uralják a piac nagyon nagy részét továbbra is, tehát a jelentős növekedés ellenére sem tudhatjuk biztosan, hogy a NoSQL adatbázisok vajon versenybe tudnak-e szállni a relációs adatbázisokkal jelentős piaci részesedésért hosszú távon. A 2014. októberi helyezések alapján az első tíz legnépszerűbben két NoSQL adatbázis kezelő rendszer kapott helyett, de a további helyeken is jó néhány társuk szerepel (4. ábra). (A pontos népszerűség alapú pontozási rendszer leírása a honlapon megtalálható. Beleszámít például, hogy

hány tweet érkezik Twitteren az egyes rendszerekről, és hogy hány weboldalon vannak megemlítve.)

Popularity changes per category, October 2014



3. ábra Adatbázis kezelő rendszerek népszerűségének változása [3]

229 systems in ranking, October 2014

Rank	Last Month	DBMS	Database Model	Score	Changes
1.	1.	Oracle	Relational DBMS	1471.90	+4.99
2.	2.	MySQL	Relational DBMS	1262.97	-34.17
3.	3.	Microsoft SQL Server	Relational DBMS	1219.60	+10.73
4.	4.	PostgreSQL	Relational DBMS	257.72	+1.92
5.	5.	MongoDB	Document store	240.41	-0.58
6.	6.	DB2	Relational DBMS	207.67	+10.64
7.	7.	Microsoft Access	Relational DBMS	141.64	+1.16
8.	8.	SQLite	Relational DBMS	94.95	+2.34
9.	↑ 10.	Sybase ASE	Relational DBMS	86.79	+1.37
10.	↓ 9.	Cassandra	Wide column store	85.70	-2.16

4. ábra Adatbázis motorok helyezései népszerűség alapján [7]

Miért is ilyen sikeresek manapság a NoSQL adatbázis rendszerek? Nagyrészt dinamikus sémákat használnak a statikus relációsakkal szemben és más jelleggel tárolják az adatokat. Általában komplexebb adatokat tárolnak és lehetővé teszik nem-strukturált adatok hatékony tárolását is. Valamint a relációs adatbázisok általában vertikális skálázódásával szemben horizontálisan skálázódnak. A vertikális skálázódás azt jelenti, hogy nagyobb igény eléréséhez vagy a hardver teljesítményét kell növelni, vagy el lehet az adatbázist több szerverre osztani, de az jelentős mérnöki munkát igényel általában. A horizontális skálázódásnál ezzel szemben új szerverek hozzáadásakor automatikusan áramlik át az adat [8]. A relációs adatbázis kezelők többsége is képes a horizontális skálázódás megvalósítására, de az adatok integritásának szigorúbb feltételei miatt, a

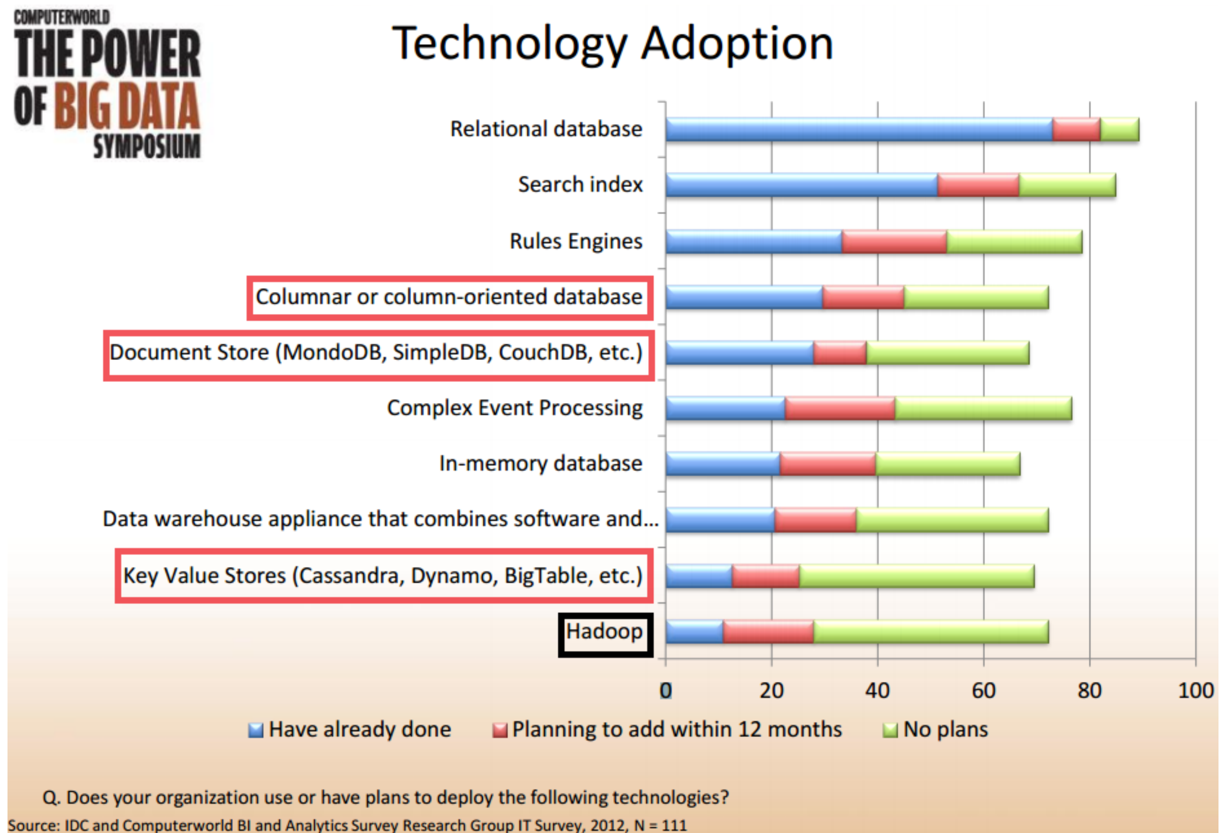
NoSQL adatbázisok ezt hatékonyabban teszik meg, hiszen ott alapvetően nincsenek ilyen megkötések. Valamint azért is sikeresek a NoSQL adatbázis rendszerek, mert képesek olyan nagy mennyiségű adat tárolására és elemzésére is, melyre a relációsak már nem, vagy nem hatékonyan képesek.

A fentebb vázolt trendet (adatmennyiség növekedése és a feldolgozási komplexitás növekedése) Big Data néven ismeri az irodalom. A fogalmat először 1997 júliusában használták NASA kutatók és azt állították, hogy rövidesen problémát fog jelenteni az eddigi számítógépes rendszerek számára. A 2000-es évek elején kezdett elterjedni a fogalom és azóta már sokan sokféleképpen definiálták [9]. A forrásban megjelölt helyen dolgozatom írásakor 32 definíciót szedtek össze. Ezek közül érdemes megemlíteni Doug Laney 2001-ben definiált híres 3V fogalmát, mely szerint a méret, sebesség és a változatosság (volume, velocity, variety) nagymértékű kiterjedésekor beszélhetünk Big Data-ról [10]. Ez a fogalom is bővült azóta és sok más is keletkezett. Az egyik elismertebb definíció szerint, akkor beszélünk Big Data-ról, ha az adatok mennyisége, milyensége, változási gyorsasága miatt már a hagyományos adatbázis kezelőkkel nem, vagy nem hatékonyan tudjuk analizálni az adatokat.

Manapság márpedig megjelent az igény az ilyen jellegű adatok tárolására és elemzésére is, hiszen keletkezik sok strukturálatlan adat, ami gyorsan is változik és hatalmas méretű. Gondoljunk például az internetforgalomra, de sok más területen is rengeteg adat keletkezik, melyeket a cégek saját céljaikra szeretnének, minél részletesebben elemezni [11]. A NoSQL adatbázisok elterjedtsége tehát nagyban annak is köszönhető, hogy képesek hatékonyan ilyen jellegű adatok feldolgozására is. A NoSQL adatbázisoknak megvannak a hátrányai is a relációsakkal szemben, amiket folyamatosan próbálnak javítani, mint például könnyen lehet, hogy a jövőben a NoSQL rendszerek is követni fogják az ACID elveket [12][13]. Várhatóan a jövőben a két rendszer megmarad egymás mellett és mindkettő érvényesülni tud majd ott, amely területen jobban használható, tehát a relációsak a hagyományos adatok körében, míg a NoSQL adatbázisok főleg a Big Data területen. [14].

A Big Data fogalomkör tehát egy új probléma elé állította a számítógépes rendszereket. Egyfelől a NoSQL adatbázisok biztosítanak megoldást az ilyen jellegű adatok hatékony tárolására és elemzésére, de nem ők az egyetlenek. Létezik egy Apache Hadoop nevű nyílt forráskódú keretrendszer, melynek segítségével szintén lehetőség nyílik ilyen adatok kezelésére. Mielőtt a Hadoop konkrét megvalósítására rátérnék fontos megjegyezni, hogy a Hadoop nem adatbázis kezelő rendszer. Egy olyan szoftver rendszer, mely lehetővé teszi nagy adathalmazok elosztott feldolgozását egy egyszerű programozási modellt használva. A Big Data problémakör kezelésére manapság tehát alapvetően ez a két rendszer biztosít megoldást. A NoSQL adatbázis kezelő rendszerek világában már láttuk, hogy rengeteg rendszer áll rendelkezésre. Ezek közül manapság a MongoDB a legnépszerűbb [15]. A Hadoop rendszerre is próbáltak alternatívákat létrehozni, de közülük egyértelműen a Hadoop a legelterjedtebb, ezt implementálják a legtöbb helyen. Ők tehát a két legelterjedtebb rendszer, melyek alapvetően különbözőek,

de használnak egymástól átvett elemeket [16]. (A MongoDB használja a MapReduce programozási modellt, a Hadoop pedig az HBase NoSQL adatbázist, tehát vettek át egymástól elemeket, ezekről később lesz szó.) Egy 2012-es felmérés szerint például a NoSQL adatbázisokat több helyen használják, mint a Hadoop-ot (5. ábra).



5. ábra Big Data megoldások használati adatai [17]

A kutatás során célul tűztem ki a BigData megoldások megismerését és különböző adatfeldolgozó algoritmusok skálázhatóságának vizsgálatát Hadoop környezetben. A dolgozatban elsőként bemutatom az elterjedt megoldásokat, ismertetem a Hadoop alapjait, melyek a munka megértéséhez szükségesek, majd bemutatom a megvalósított algoritmusokat. Ezt követően bemutatom az elvégzett méréseket és a mérések segítségével igazolom a skálázódási lehetőségeket Hadoop környezetben. A dolgozatban bemutatott eredmények tanulságul szolgálhatnak hasonló nagyméretű megoldások tervezése és megvalósítása esetén.

2. Hadoop

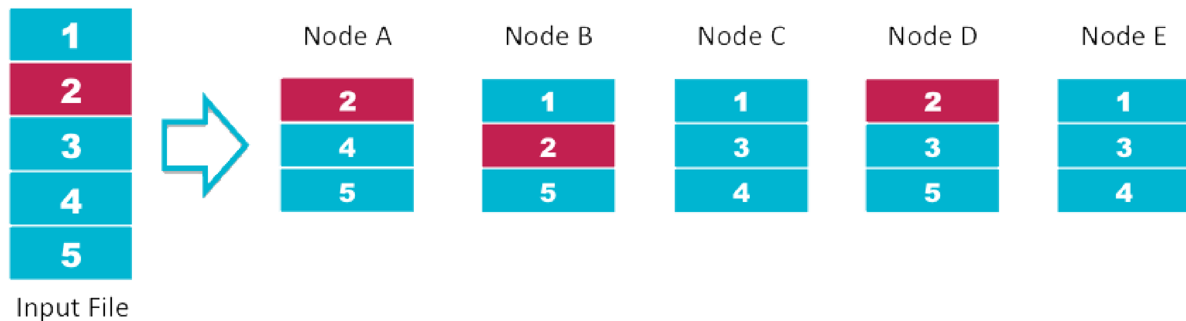
Doug Cutting és Mike Cafarella a weblapok indexelésének problémáját szerették volna megoldani a keresési motorok tökéletesítése érdekében. A Nutch nevű kereső rendszer 2002-ben indult, de weblapok milliárdjait már nem volt képes indexelni [18]. Ehhez nyújtott segítséget a Google által 2003-ban megjelentett Google File System-ről szóló cikkük [19]. Ebben egy elosztott, skálázható, adat-intenzív alkalmazások tárolására alkalmas fájlrendszert írnak le. Ez alapján 2004-re el is készítettek a Nutch Distributed Filesystem-et, melynek segítségével lehetővé vált hatalmas fájlok tárolása és adminisztratív időket megspóroltak a fájlrendszer használatával. Időközben 2004-ben a Google újabb anyagot publikált, melyben a MapReduce nevű programozási modellt mutatta be [20]. Fontos kiemelni, hogy itt valóban egy programozási modellről van szó, nem pedig egy programnyelvről. A MapReduce elosztott módon teszi lehetővé nagy adathalmazok feldolgozását és előállítását. Ezt szintén felhasználta a Nutch és 2005-re az algoritmusaik nagy része MapReduce-t és NDFS-t használt. 2006 februárjában ezt a projektek nevezték át és lett Hadoop a neve. (Ekkor az NDFS-t is átnevezték Hadoop Distributed File System, HDFS-re.) A Hadoop név nem egy mozaikszó, hanem az alkotó, Doug Cutting kisfiának játék elefántjáról kapta a nevét, mely a keretrendszer logójává is vált. Ugyan ebben az évben Doug Cutting a Yahoo!-hoz csatlakozott és itt folytatták a Hadoop fejlesztését. 2008-ra például a keresési indexüket egy 10.000 csomópontból álló Hadoop fürt segítségével hozták létre. 2008 januárjában az Apache Hadoop felső szintű projektté vált. Ekkoriban a Facebook, a Last.fm és a New York Times is használta már a Hadoop-ot [21].

A Hadoop-ot nagyon sok különböző probléma megoldására használják a mindennapokban. A Last.fm cég például a felhasználóik zenei ízlését és az egyes zeneszámok közötti korrelációkat elemezték a segítségével [22]. A New York Times a sok éves archívumát digitalizálta a Hadoop segítségével és létrehozta a TimesMachine nevű oldalukat, melyben 150 évre visszamenőleg tekinthetünk meg az újságban megjelent cikkeket [23][24]. És ezeken kívül is még sok helyen alkalmazzák [25].

2.1 HDFS

A Hadoop fájlrendszere tehát az elosztott és skálázható HDFS, melyet úgy fejlesztettek, hogy átlagos hardverrel rendelkező gépeken működhessen. Ezeken a gépeken tehát az összes adat elosztottan tárolódik. Mivel átlagos gépekről beszélünk és akár több ezer gépről is ipari környezetben, a gépek elromlása gyakori jelenség. Az adatvesztés elkerülése természetesen alapvető követelmény minden adattároló rendszer számára. A

HDFS-ben éppen ezért, az adatokat többszörözve tárolják, úgynevezett replikákat hoznak létre belőlük és a fájlrendszer igyekszik úgy elrendezni őket, hogy az egyes másolatok külön csomópontokra kerüljenek (6. ábra). (Általában három replika az alapértelmezett.) Így ha egy gép meghibásodik és elveszti az adatát a HDFS a megfelelő módon képes arról gondoskodni, hogy adatvesztés nélkül működhessen tovább.



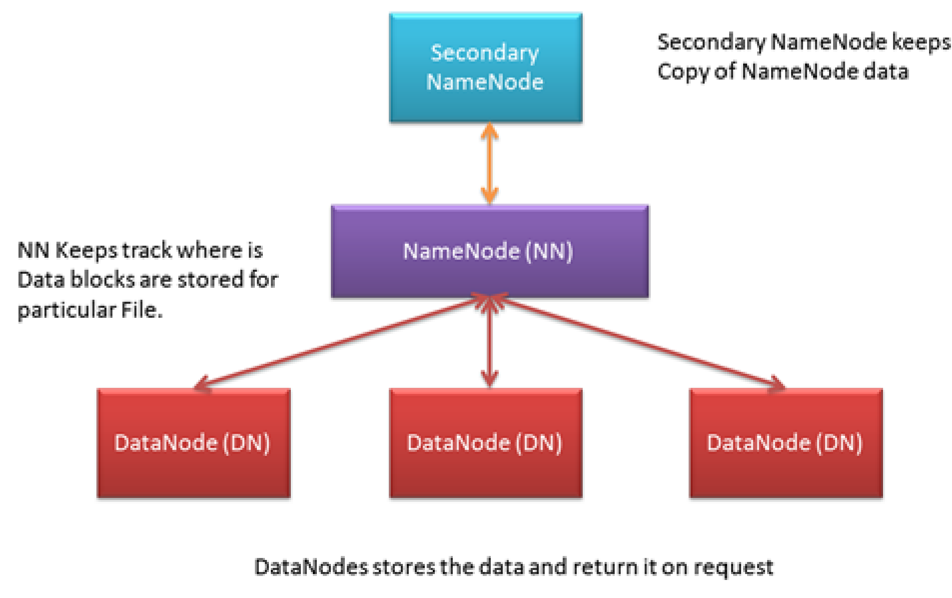
6. ábra Replikációk elhelyezkedése a HDFS csomópontokon [26]

A HDFS nagy fájlok tárolására van felkészítve alapvetően, így ezeket általában 64-256MB közötti blokkokban tárolja a fájlokat. (Általában 128MB.) (Tehát ha például 2MB-os fájlokat szeretnénk eltárolni a fájlrendszeren, azok mind külön blokkra fognak kerülni, tehát nagyon hely pazarlóan, érezhető tehát, hogy ez a nagy blokkméret a nagy fájlok tárolását támogatja.)

Egy másik nagyon fontos előnye a fájlrendszernek, hogy lehetővé teszi, hogy az alkalmazások az adatokhoz közel fussanak. Mivel elosztott rendszerről van szó, ezért külön fizikai gépeken történik az adatfeldolgozás, tehát az adatnak az adott gépre kell eljutnia, ami a hálózat áteresztőképessége miatt probléma lehet. A közel jelen esetben azt jelenti, hogy azon a gépen legyen tárolva az adat lehetőleg, amin fel lesz dolgozva, vagy minél kevesebb hálózati eszközön keresztül legyen elérhető. Ez nagyon nagyban elősegíti az adatok feldolgozásának teljesítményét főleg ilyen hatalmas adatmennyiségeknél, hiszen az adatok hálózaton keresztüli mozgatása könnyen szűk keresztmetszetté válna egyébként és lelassítaná az adatfeldolgozást.

A HDFS alapvetően master/slave felépítést követ. A *NameNode* a fő csomópont, aki a fájlrendszer névterét kezeli és a kliensek fájl elérését határozza meg. A többi csomópontot *DataNode*-oknak hívjuk, melyek általában egy-egy fürtbeli csomóponthoz tartoznak. Ők a rajtuk tárolt adatok kezeléséért felelősek. A *NameNode* végzi a fájlok megnyitását, bezárását, átnevezését, a *DataNode*-ok pedig a kliensektől érkező írási és olvasási kéréseket szolgálják ki. A *NameNode* utasítására a *DataNode*-ok blokkokat hozhatnak létre, törölhetnek ki vagy replikációkat hozhatnak létre. A *DataNode*-ok periodikus időközönként egyrészt küldenek egy úgynevezett heartbeat-et a *NameNode*-nak, mellyel jelzik, hogy megfelelően működnek, illetve egy blokkjelentést, melyben a rajtuk található blokkok listáját küldik el. Ebből a blokkjelentésből tudja a *NameNode*, hogy van-e olyan blokk, amiből nincs annyi replika a rendszerben, mint amennyi

specifikálva van neki. Ilyen esetben, amikor „ráér”, tehát van szabad kapacitása erre és nem éppen más feladatot hajt végre, akkor kiadja az utasítást a *DataNode*-oknak a megfelelő blokkok replikálására, ezzel helyreállítva a rendszer redundanciáját. A *NameNode*-dal egy fájlrendszerben található egy *EditLog* nevezetű fájl, melybe minden olyan tranzakció bekerül, mely a fájl rendszer meta adatának bármilyen változását tartalmazza. A *NameNode* indulásakor ennek segítségével állítható elő biztosan az előzőleg használt állapot. Általában van egy *Secondary NameNode* nevű elem is a rendszerben. Ennek a neve félrevezető olyan szempontból, hogy nem a *NameNode* valamilyen fajta helyettesítésére szolgál, hanem bizonyos időközönként lementi a *NameNode* és az *EditLog* tartalmát. A *NameNode* leállásakor, ha a *NameNode*-ot újra lehet indítani ugyanazon a fizikai gépen, akkor nem szükséges az összes *DataNode*-ot is újraindítani, hiszen a *Secondary NameNode* segítségével az adatok helyreállíthatók (7. ábra).

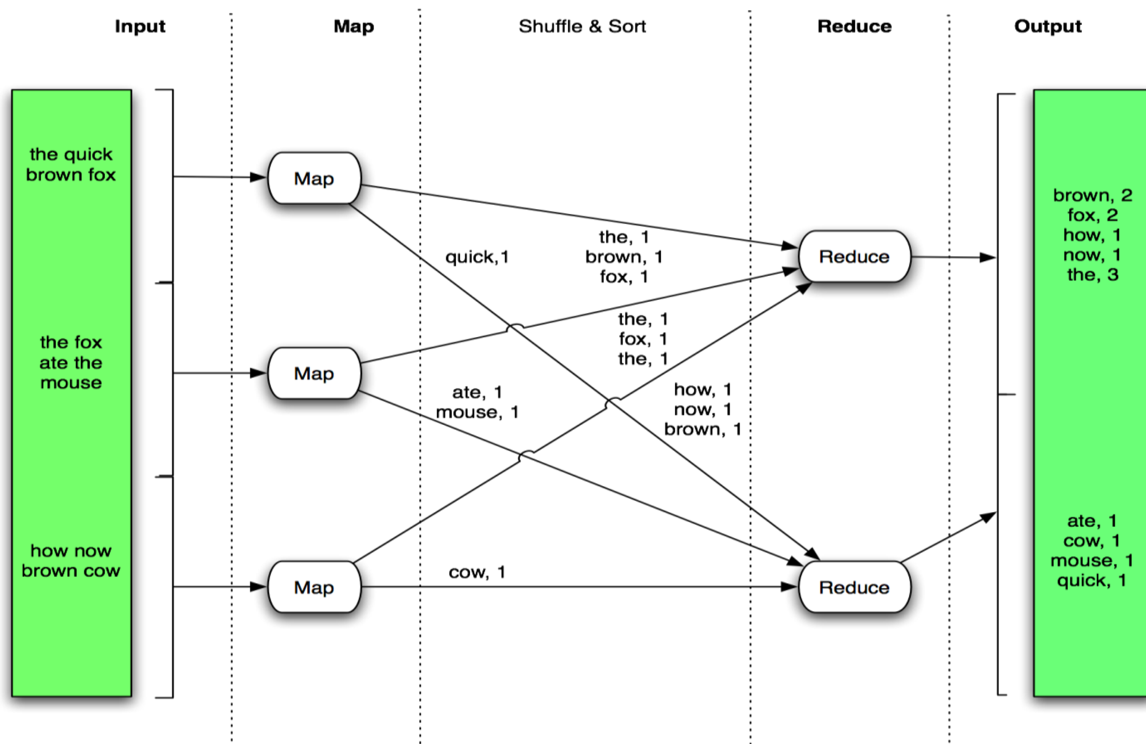


7. ábra HDFS felépítése [31]

A *DataNode*-ok meghibásodása tehát viszonylag gyakori esemény a fájlrendszerben és ezt automatikusan javítani is tudja a rendszer adatvesztés nélkül. A *NameNode* azonban Single Point of Failure a rendszerben, melynek meghibásodásakor kézi beavatkozás szükséges a helyreállításhoz, de ilyenkor bekövetkezhet adatvesztés. Ez egy sürgető problémája volt a Hadoop rendszernek a közelmúltig, de a 2012-2014-es években egyre több megoldás született a problémára [27][28][29][30].

2.2 MapReduce

A Hadoop másik alapvető része a MapReduce programozási paradigma és a hozzá tarozó szoftverrendszer, mely képes nagy adathalmazok feldolgozására és előállítására elosztott és párhuzamosított módon [32]. A név a modell két fő fázisából jön. Ezek a *map* és a *reduce* függvények. A *map* függvény kulcs-érték párt kap a bemeneten és kulcs-érték párok listáját adja vissza. Ez formálisan: *Map*(k1,v1) -> list(k2,v2). Tehát a *map* függvény tetszőleges új kulcsokat és azokhoz tartozó értéket állít elő a kapott kulcs-érték párból. Egyszerre több *map* függvény fog párhuzamosan lefutni, így több kulcs érték pár keletkezik a *map* függvények kimenetéül. Itt van egy úgy nevezett *shuffle* fázis, melyet a rendszer magától elvégez nekünk. Ebben a fázisban az azonos kulcsokhoz tartozó értékeket egy listába gyűjti a rendszer és ezeket adja a *reduce* függvény bemenetére. A *reduce* függvény ezekből a bemenő paraméterekből kulcs-érték párok listáját állítja elő, melyek összessége egyben a teljes MapReduce program kimenete is lesz. A *Reduce* formálisan: *Reduce*(k2, list(v2)) -> list(k3,v3). A működés megértését egy egyszerű MapReduce program bemutatásával segítem elő. Ez a program tekinthető a programozási modell „Hello World” programjának, mely egy szöveges fájlban az egyes szavak előfordulási számát adja meg eredményként (8. ábra).

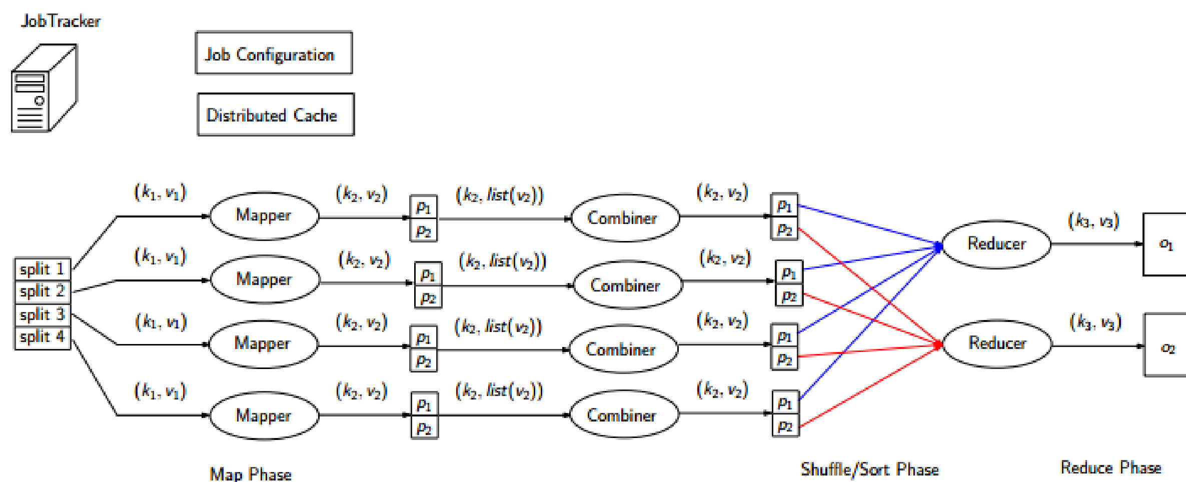


8. ábra WordCount MapReduce program

Egy szöveges fájlt alapértelmezett esetben sorokra bont szét a keretrendszer, így ezek lesznek az egyes *map*-ek bemenetei. A *map*-ek kulcs-érték párokban kapják a bemenetüket, így egy sor egész pontosan egy értéknek fog megfelelni. (Hogy a kulcs jelen esetben mi, az nem érdekes programozási szempontból.) A sorokat a szóközök

mentén feldaraboljuk, így jutunk az egyes szavakhoz. A *map* függvény kimenete pedig úgy fog kinézni, hogy mindig az egyes szavak lesznek a kulcsok és mindig egy lesz a hozzájuk tartozó érték. Ezeket a *shuffle* fázis fogja feldolgozni. Jelen esetben például a „the" szóhoz tartozó értékeket (vagyis az egyeseket) összegyűjti egy listába és garantálja számunkra, hogy egy adott kulcshoz tartozó értékek mind egy *reduce* függvény bemenetére kerüljenek. A *reduce* függvény egy adott kulcshoz tartozó lista elemszámát adja vissza eredményként. Ezzel gyakorlatilag meg tudjuk számolni egy szöveg szavainak a számát két egyszerű függvény implementálásával.

Egy MapReduce program írása fejlesztői szemmel elég egyszerűnek tekinthető, hiszen csak a *map* és a *reduce* függvényeket kell implementálni, az összes többi teendőt elvégzi helyettünk a keretrendszer. Nézzük meg, hogy is történik mindez (9. ábra).



9. ábra MapReduce program működése [33]

Egy futtatandó MapReduce programot *Job*-nak nevezünk. Ezek kezelését a Hadoop-ban a *JobTracker* és a *TaskTracker*-ök végzik. Ők a *NameNode* és *DataNode*-okhoz hasonlóan master/slave kapcsolatban állnak egymással. A *JobTracker* feladata a *Job*-ok, illetve a *Map* és *Reduce* taszkok ütemezése. A *TaskTracker*-ök feladata pedig a konkrét taszkok végrehajtása. Az ábrán látható *Job Configuration* állományba kerülnek a *Job* specifikus változók és a *JobTracker* őket az összes *Map* taszkhoz eljuttatja. Minden *TaskTracker*-höz tartozik egy elosztott cache is, melyben nagy fájlokat szoktak tárolni a gyorsabb elérés érdekében. Ezeket a *JobTracker* küldi szét a *TaskTracker*-öknek. A fájlrendszeren tárolt feldolgozandó fájlokat a keretrendszer valamilyen méretű részekre vágja és a *JobTracker* eljuttatja ezeket a darabokat az egyes *TaskTracker*-ökhöz és azokon belül egy-egy *Map* taszk kapja meg őket feldolgozásra. Itt jön tehát a *map* függvény, amelyet a programozó implementál és a bemenő kulcs-érték párokból kulcs-érték párok listáját állítja elő. Miután minden *map* függvény végzett a *shuffle* fázisban az egyforma kulcshoz tartozó értékeket egy listába gyűjti a rendszer és ezeket adja a *reduce* függvények bemenetére. Itt ismét a *JobTracker* rendeli el, hogy valahány *TaskTracker* végezze el a *reduce* műveletet. A *reduce* művelet pedig a fejlesztő által implementált módon a kulcshoz

rendelt listából előállítja a MapReduce *Job* végeredményét kulcs-érték párok formájában.

A MapReduce paradigma megvalósítja a programozó számára az elosztott és párhuzamos működést. Ha valamelyik *map* vagy *reduce* függvény valami oknál fogva meghiúsul (például, mert elromlik az adott számítógép), akkor a *JobTracker* egy új *TaskTracker*-t jelöl ki a feladat elvégzésére. Illetve ha egy *map* elvégezte a saját taszkját és még van másik *map* taszk, ami valami miatt még nem végzett, a *JobTracker* elrendelheti, hogy a felszabadult csomóponton végrehajtsa ugyanazt a taszkot. Amelyik hamarabb befejeződik, annak az eredményét fogja eltárolni a másikat pedig terminálja. A MapReduce tehát többféle módon gondoskodik a végrehajtás optimalizációjáról, kiegyensúlyozásáról elfedve azt a fejlesztő elől.

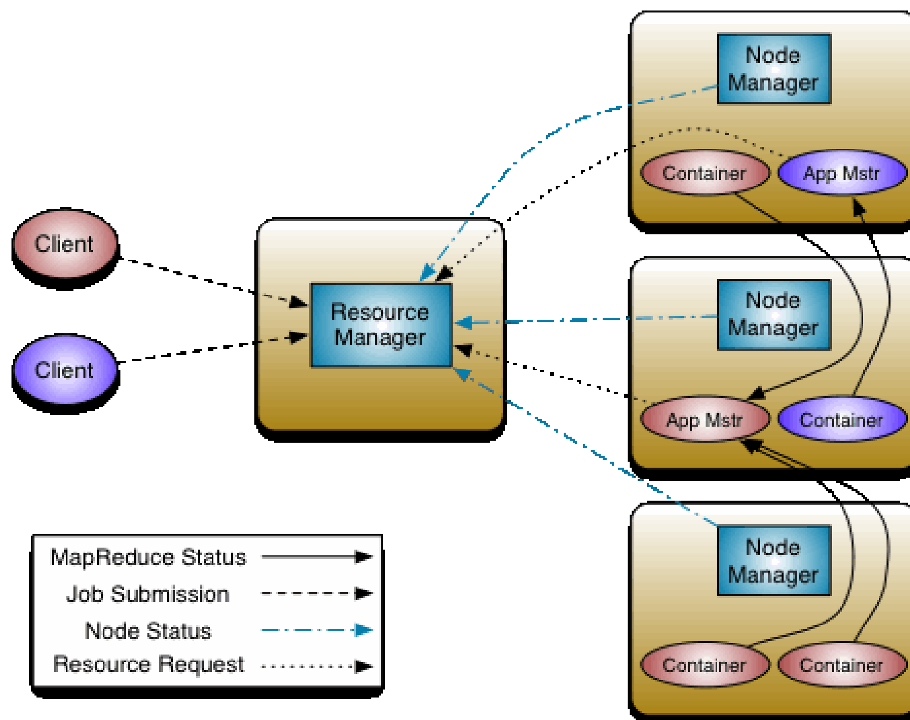
A MapReduce paradigma tehát a Hadoop része, de nem csak ott van jelen. Egy önálló programozási modell, melyet más rendszerek is használnak, például a MongoDB is. Ezen kívül használják például elosztott minta alapú keresésre, elosztott rendezésre és a gépi tanulás területén is. Használják a modellt több processzoros rendszerek, önkéntes számítási kapacitást kiajánló rendszerek és dinamikus felhő-alapú rendszerek is.

2.3 Kapcsolódó projektek

A Hadoop keretrendszer alapvetően 4 modulból épül fel. Ebből kettő a fentiekben tárgyalt HDFS és a MapReduce. Ezeken kívül tartalmazza a Hadoop Common nevű modult, mely a többi modul használatához szükséges segédprogramokat biztosítja. A negyedik modul pedig az úgynevezett Hadoop YARN (Yet Another Resource Negotiator) [34]. A MapReduce javítása során az új verzió kiadásakor bevezették ezt a modult, melyre gyakran MapReduce 2.0-ként is hivatkoznak. Az alapvető gondolata, hogy a *JobTracker* erőforrás kezelő és *Job* menedzselő tevékenységét szétválassza. Ezek lesznek a *ResourceManager* (RM) és az *ApplicationMaster*. A RM szolgái az úgynevezett *NodeManager*-ök lesznek. Az ábrán látható *Container* memória, processzor, diszk és hálózati adatokat tartalmaz. A *ResourceManager* felelős az erőforrások allokálásáért, a *Job* kérések elfogadásáért és hiba esetén az *ApplicationMaster* tároló újraindításáért. Az *ApplicationMaster* pedig folyamatok monitorozásáért felel (10. ábra).

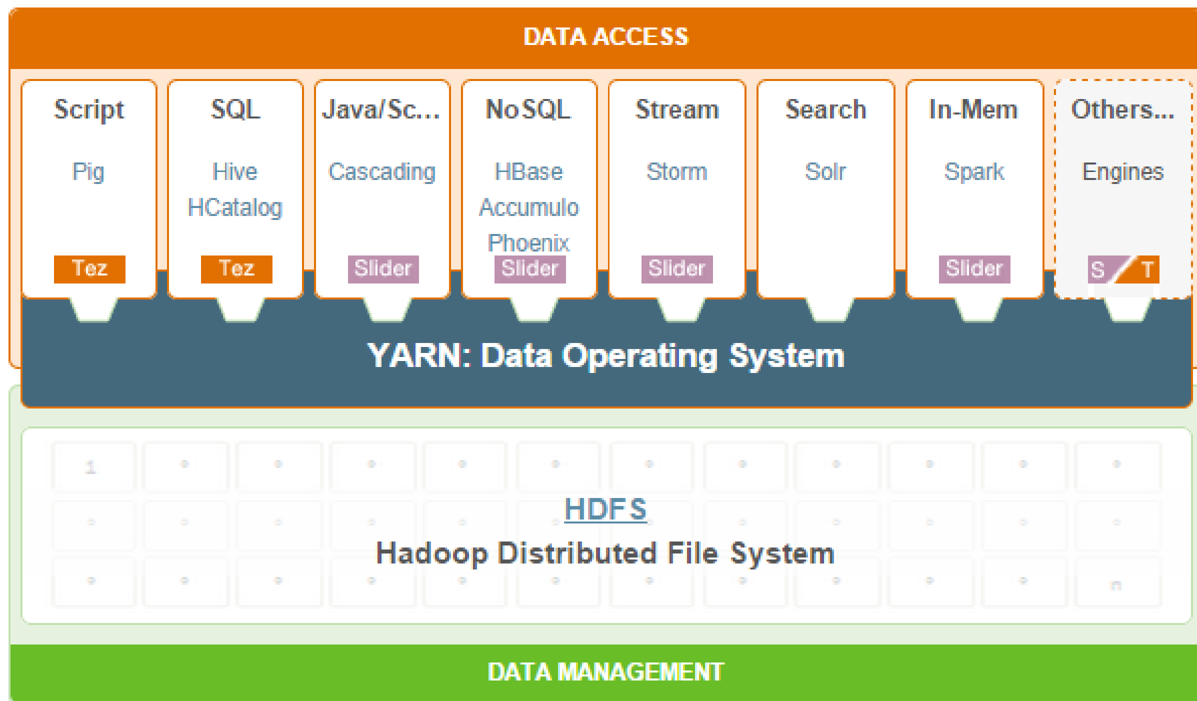
A Hadoop-nak ez tehát a négy alapvető modulja, de az évek során egyre több olyan projekt jött létre, melyek szintén Hadoop felett használhatók. Ezek közül néhány említésre méltóbbat ismertetek. Az HBase egy oszlopalapú NoSQL adatbázis kezelő rendszer, mely MapReduce-t használ az adatok elemzésére [36]. A Hive egy adattárház, melybe az adatainkat relációs adatbázishoz nagyon hasonló táblákban tárolhatjuk és

lekérdezéseket hajthatunk végre, melyek MapReduce kódra fordulnak és úgy hajtódnak végre [37]. A Pig egy magas szintű adat-folyam vezérelt nyelv, mely szintén MapReduce kódra fordul [38]. A Spark egy gyors és általános számítási motor Hadoop-ban tárolt adatok feldolgozására [39]. A ZooKeeper pedig egy összehangoló szolgáltatást biztosít elosztott alkalmazásokhoz [40]. Ezen alkalmazások egyrészt azért jöttek létre, hogy ne kelljen mindig konkrét MapReduce kódot írni. Vannak olyan feladatok ugyanis, melyeket MapReduce segítségével már nehéz implementálni és optimalizálni. Ezeknek az alkalmazásoknak a segítségével azonban automatikusan generált és egyre jobban optimalizált MapReduce kód fogja megoldani a feladatot. Másrészt a Hadoop keretrendszerhez sok olyan új hasznos segítő alkalmazásra volt szükség melyek bizonyos problémákra nyújtanak megoldást és ezek az alkalmazások ezeket a hiányosságokat próbálják megoldani. A Hortonworks cég Hortonworks Data Platform 2.2 nevű csomagjában például az ábrán látható projektek szerepelnek (11. ábra). (Dolgozatom írásakor ez a verzió még nem jelent meg, csak egy előzetes verzió érhető még el a honlapon.)



10. ábra YARN felépítése [35]

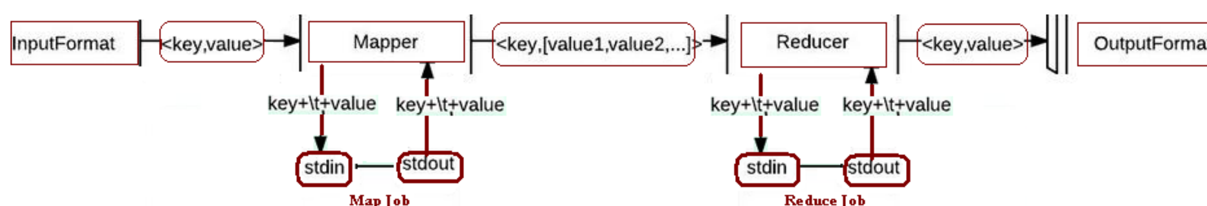
Dolgozatom írásakor (2014.10.02.-án) jelent meg például a Spring is Hadoop-ra [42].



11. ábra Hadoop komponensek [41]

2.4 Hadoop Streaming

A Hadoop-ot Java nyelven írták és az alapértelmezett fejlesztési nyelve is a Java, tehát ezen a nyelven írhatunk MapReduce kódokat. Az igény azonban hamar felmerült, hogy más nyelveken is lehessen MapReduce programot írni, hiszen nem minden fejlesztő jártas a Java nyelvben, gyakran az általuk jól ismert nyelvben szerettek volna fejleszteni. Erre hozták létre a Hadoop Streaming API-t, melynek segítségével lehetővé válik más programnyelvekkel is a fejlesztés. Az egyetlen kritérium, hogy a programnyelv képes legyen standard inputról olvasni és standard inputra írni, tehát gyakorlatilag bármilyen nyelvből lehet MapReduce-hoz programot írni. A rendszer folyamok segítségével továbbítja az adatot az alkalmazás és a Hadoop platform között. A kulcs-érték párok mindig szövegként jelennek meg és „tab" karakterrel vannak elválasztva. A programozó feladata ismét a *map* és *reduce* függvények megvalósítása. A *map* függvény a bemenetét a standard inputról veszi és a standard inputra írja. Itt jön a szokásos *shuffle* fázis, majd a *reduce* fázis ismét a standard inputról kapja a bemenetét és az eredményt a standard outputra írja (12. ábra).



12. ábra Hadoop Streaming *Job* folyamatábrája [43]

Azzal a kényelemmel azonban, hogy más programnyelveken fejleszthetünk, teljesítményt veszíthetünk. A folyamokon keresztüli adattovábbítás ugyanis ronthatja a rendszer teljesítményét, mely adat-intenzív *Job*-ok esetén érzékelhető igazán [44].

2.5 Használati módok

A Hadoop alapvetően három különböző módban telepíthető [45]. Az alapértelmezett a *Standalone* (önálló) mód. Ez a verzió egy számítógépből álló rendszerre telepíthető. A Hadoop-ot alapvetően elosztott adatfeldolgozásra találták ki és ott hatékony, de létezik olyan verziója melyet egy számítógéppel is lehet használni, tanulási és megértési célokra főleg. Ugyanígy egy számítógépből álló rendszerre telepíthető az úgynevezett *Pseudo distributed* (Ál-elosztott) mód. Ez a verzió azonban egy több számítógépből álló rendszert szimulál számunkra, így jobban lehetővé téve a Hadoop rendszer megismerését. A harmadik mód neve a *distributed* (elosztott) mód. Ez valóban több csomópontból álló rendszerhez telepíthető. Ez való tehát szokásos ipari használatra.

A Hadoop-ot az évek során jó néhány cég implementálta és fejlesztette tovább. Gyakori, hogy virtuális lemezképeket biztosítanak a felhasználók számára, melyre előre telepítve találjuk a Hadoop keretrendszert és néhány hozzá tartozó projektet is (például HBase, Hive, Pig). Így egy virtuális gépen *Standalone* vagy *Pseudo distributed* módban megismerhetjük a Hadoop használatát. A két legnagyobb cég, akik ilyen virtuális lemezképeket biztosítanak a HortonWorks és a Cloudera [46][47]. Ez a megoldás tehát ismét a Hadoop megismeréséhez lehet hasznos, de igazi használatra a felhő-szolgáltatók biztosítnak megoldást [48]. Ők a Hadoop-ot Platform-as-a-Service (PaaS) formájában biztosítják a felhasználóik számára. Segítségükkel tehát a teljes Hadoop keretrendszert felkonfigurált állapotban kapjuk és használhatjuk. Náluk létrehozhatjuk saját megadott számú csomópontból álló fürtünket és a megfelelő módszerekkel futtathatjuk a saját MapReduce kódunkat a feltöltött adatainkon. A felhő-szolgáltatók számítási idő alapú fizetési rendszere miatt, ezt gyakran megéri ipari környezetben is használni. Például a New York Times az Amazon rendszerét használta az adatai digitalizáláshoz. A felhő-szolgáltatók közül a jelentősebbek ezen a területen az Amazon Web Services-nek az Elastic MapReduce, a Google Compute Engine-nek a Mapr, a Microsoft Windows Azure-nek pedig az HDInsight nevű Hadoop alapú megvalósításai [49][50][51]. (Néhány felhő-szolgáltató ingyenes kipróbálási lehetőséget is biztosít, így nagyobb környezetben is megismerhető a Hadoop.)

3. A feldolgozandó adathalmaz

A dolgozatban azt tűztem ki célul, hogy megvizsgáljak egy olyan adathalmazt MapReduce technológia segítségével, amelyet még nem elemeztek ezzel a megoldással. A célom eléréséhez első lépésben kerestem egy nagy adathalmaz, mely elég nagy ahhoz, hogy érdemes legyen Hadoop segítségével feldolgozni és hasznos adatok nyerhetők ki belőle. Az interneten manapság sok ilyen adathalmazt lehet találni, ezek közül a választásom egy repülési adatokat tartalmazó adathalmazra esett [52][53]. Ezt az adathalmazt a Research and Innovative Technology Administration (RITA) nevű szervezet gyűjti és biztosítja publikusan elérhetővé. Az adatokat 1987 októbere óta gyűjtik és hónapról hónapra az újabb eredményeket is közzéteszik egészen napjainkig. Az adatok között az összes általuk nyilvántartott Egyesült Államokon belüli járat különféle adatait gyűjtik össze. Ezek közül a fontosabbak:

- év
- hónap
- nap
- tervezett indulási idő
- tervezett érkezési idő
- valós indulási idő
- valós érkezési idő
- légitársaság neve
- indulási hely
- érkezési hely
- járatot törölték-e
- járatot eltérítették-e
- légitársaság miatti késés percben
- extrém időjárás miatti késés percben
- nemzetközi repülési rendszer miatti késés percben
- biztonság miatti késés percben
- más járatok miatti késés percben

(A teljes listának egy jó összefoglalása megtalálható a forrásban megjelölt helyen [54].) A választásom azért is esett erre az adathalmazra, mert már sokan sokféleképpen feldolgozták, de hivatalosan elérhetően Hadoop keretrendszert használva még nem. Az American Statistical Association (ASA) ugyanis kétévente hirdeti meg az úgynevezett Data Exposition (adat kiállítás) nevű versenyét, melyben egy nagy adathalmazt tesznek elérhetővé és a megadott feltételeket betartva a résztvevőknek ezt az adathalmazt kell elemezniük [55]. A kapott eredményeket pedig főként grafikus formában kell prezentálniuk egy poszteren. 2009-ben az általam választott adathalmazt írták ki feldolgozásra [56]. Megkötés arra nézve nem volt, hogy pontosan milyen adatot kell az adathalmazból kiszedni, ezt tetszőlegesen választhatta mindenki. A nagy adathalmazra

való tekintettel két megoldást javasoltak a feldolgozására. Egyrészt szokványos linux parancssori eszközöket az adatok rendezésére, szűrésére [57]. Másrészt az SQLite adatbázis kezelőt az adatok tárolására [58][59]. A versenyre kilenc poszter érkezett, ők az imént említett két megoldás mellett a következőket használták még például: R és hozzátartozó csomagok például ggplot2, mySQL adatbázis és SAS szoftvereket [60][61][62]. Én pedig most a Hadoop keretrendszert használva szerettem volna hasznos adatokat kinyerni az adathalmazból. Hogy pontosan milyen adatokat, azt a konkrét megvalósítások résznél tárgyalom.

Megvolt tehát a feldolgozandó adathalmaz, így szükségem volt egy környezetre is, melyre a Hadoop keretrendszert telepíthettem és azon elemezhettem az adatokat. Először a Cloudera által kibocsátott VMware virtuális lemezképet használtam *Standalone* módban. Erre előre telepítve volt a Hadoop és egy felkonfigurált Eclipse is volt rajta. Az adat feldolgozásához először is a HDFS fájlrendszerre kellett feltöltenem az adatokat. A repülési adatok éves bontásban külön *csv* fájlokban voltak meg. Itt egy sorban volt megtalálható egy adott járat minden adata a következő sorrendben:

Year,Month,DayofMonth,DayOfWeek,DepTime,CRSDepTime,ArrTime,CRSArrTime,UniqueCarrier,FlightNum,TailNum,ActualElapsedTime,CRSElapsedTime,AirTime,ArrDelay,DepDelay,Origin,Dest,Distance,TaxiIn,TaxiOut,Cancelled,CancellationCode,Diverted,CarrierDelay,WeatherDelay,NASDelay,SecurityDelay,LateAircraftDelay

Például egy sor:

2008,1,3,4,1937,1830,2037,1940,WN,509,N763SW,240,250,230,57,67,IND,LAS,1591,3,7,0,,0,10,0,0,0,47

Ilyen formában érhetők el tehát az adatok, melyeket a virtuális gép fájlrendszeréből a HDFS fájlrendszerébe kellett töltenem. Miután az adataim a HDFS fájlrendszeren voltak a MapReduce kódokat írtam meg Java nyelven Eclipse-ben. Ehhez tehát a felkonfigurált Eclipse a rendelkezésemre állt és a Hadoop API-ban található *Map* osztályon belüli *map* és a *Reduce* osztályon belüli *reduce* függvényeket kellett felüldefiniálnom. Alapértelmezetten a szöveges fájlokat sorokra osztja fel a keretrendszer és sorokat fog az egyes *map* függvények bemenetére adni, ami alapvetően nekem megfelelt, hiszen soronként voltak az egyes járatok adatai. Miután a kódok elkészültek *jar* fájlokat fordítottam belőlük és futtattam őket.

Futás közben végig követhető százalékosan, hogy a *map* és a *reduce* fázis éppen hogy áll. A futás végén pedig különböző metrikákat tekinthetünk meg a *Job* futásáról. A HDFS-en a futás paramétereül megadott mappában megtaláltam a futások eredményét. Egy üres *_SUCCES* nevű fájl jelzi a sikeres lefutást és mellette egy *part-00000* nevű fájlban található a futási eredményem. Ha nem csak egy *reduce* taszk indul, akkor további fájlok is lehetnek itt folyamatosan növő számozással és ilyenkor több fájlban kapjuk meg a megoldást.

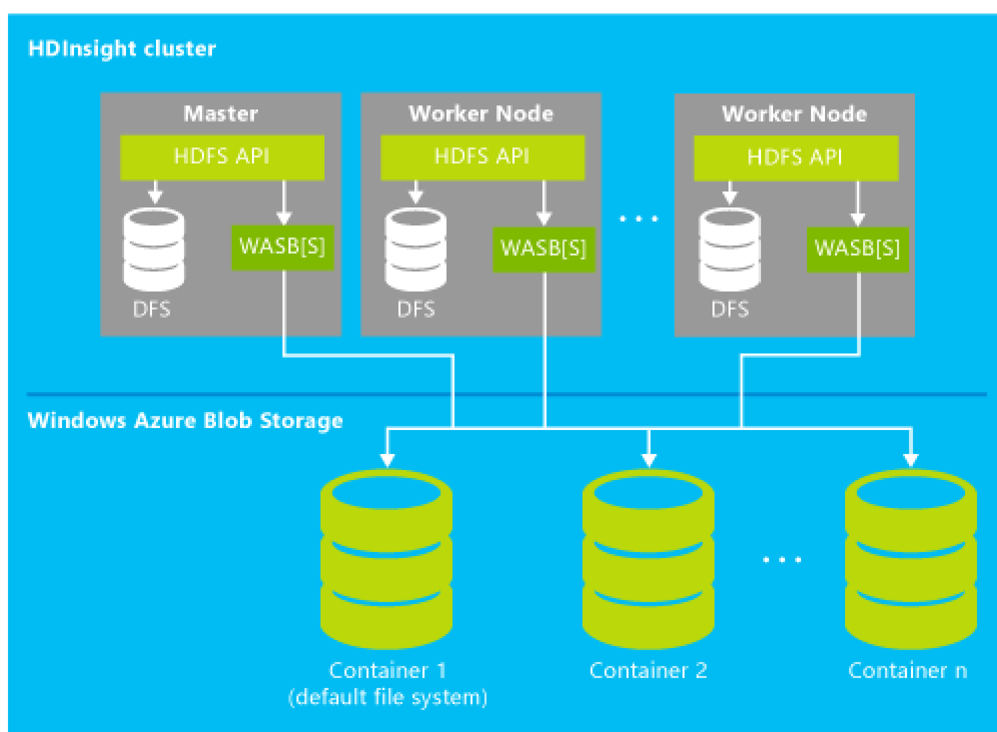
Alapvetően tehát ez volt a futtatás menete, de ezek után a MapReduce skálázódását is meg szerettem volna vizsgálni különböző paraméterek változtatása esetén. A paraméterek, melyeket változtatni lehetett a következő voltak. Egyrészt lehetett változtatni az input méretét, tehát jelen esetben legegyszerűbben azt, hogy hány darab *csv* fájlt adok a *Job*-oknak feldolgozásra. Kódból lehetett állítani bizonyos esetekben a *map* és *reduce* taszkok számát. Lehetett változtatni, hogy ne soronként, hanem valamilyen más módszerrel dolgozza fel a fájlokat. Ezen kívül van egy *uber* módnak nevezett beállítás is, melyet akkor lehet használni, ha egy konfigurációs beállításokban beállított *map* taszk, *reduce* taszk és input méret alatt van a futtatandó *Job*. Ilyen esetben, ha ez be van kapcsolva, a *map* és *reduce* taszkok egy JVM-ben fognak lefutni a szokásos két különböző helyett. A Hadoop általános használatakor optimálisabb külön JVM-ben futtatni őket, de ez a mód arra szolgál, hogy ha elég kicsi adatokon dolgozunk, akkor érdemesebb így futtatni őket és jobb sebességet kaphatunk.

Ezekkel a beállításváltásokkal tudtam tesztelni a rendszert, de az egy számítógépből álló Hadoop fürt hátrányai egyből érezhetők voltak, hiszen például több *map* taszk esetén alapvetően gyorsabb működésre számítunk, hiszen azokat párhuzamos tudja végrehajtani több számítógép egy szokványos több gépes környezetben, de egy gépes esetben mindig az egy *map* taszkot futtató megoldás volt a gyorsabb. A futási idő tehát egy gépes rendszeren semmiképp nem volt reprezentatív. Ezen kívül lehetett bizonyos más metrikákat is vizsgálni, melyekkel a MapReduce futási jellemzőit mérhetjük, de hamar felmerült az igény több gépes fürt kiépítésére, hogy a valós használathoz közelebb álló rendszeren tesztelhessem a skálázódást és így jóval pontosabb és érdekesebb eredményeket kaphassak. (Természeten ipari rendszerekben akár több tízezer gépet is egy fürtbe kötnek, amire nekem persze nem volt lehetőségem, de akár csak egy pár számítógépből álló rendszeren is lehet már tendenciákat megfigyelni és nagyobb fürtön való működésre következtetni.)

4. Adatelemzés a felhőben

Az algoritmusok tesztelésére és a skálázódás vizsgálatára a Microsoft Azure szolgáltatásait vettem igénybe [63]. Ők az HDInsight nevű, teljesen Apache Hadoop alapú rendszert biztosítják használatra. A használatba vételéhez először is egy tárhelyet biztosító szolgáltatást kellett elindítanom a felhőmben. Ezután már létre tudtam hozni egy HDInsight számítógép fürtöt. Itt a fürtnek három féle típust lehet választani most már. Lehet HBase, tehát a korábbiakban már ismertetett NoSQL adatbázis kezelő rendszer alapú. Ezen kívül 2014. október 16-a óta lehet Storm alapú is. A Storm-ot először a BackType nevű cégnél (amit azóta megvett a Twitter) Nathan Marz fejlesztette és körülbelül egy éve kezdték el a Microsoft Azure-nál is az implementálását az HDInsight rendszerükhöz [64]. A Storm milliós nagyságrendű események megbízható, valós idejű feldolgozására képes a leírásuk szerint [65]. A harmadik féle típus pedig a Hadoop, ez az, amit választottam. Verziót is lehetett választani, én a 3.1-el jelzett verziót használtam, mely a HortonWorks Data Platform 2.1-es és az Apache Hadoop 2.4-es verzióját használja [66]. A fürtben található csomópontok számának pedig 13-at adtam meg a feladataim elvégzéséhez.

Ezzel tehát rendelkezésemre állt a rendszer, melynek használatához ismét azzal kellett kezdenem, hogy az adatokat a HDFS fájlrendszerre feltöltöm. Erre az HDInsight a Windows Azure Storage Blob (WASB) nevű tároló rendszert használja (13. ábra).



13. ábra HDInsight tároló architektúra [67]

Ez a rendszer a Microsoft saját Azure Blob nevű tároló rendszerén implementálja a HDFS-t. Ezzel a saját tároló rendszerük előnyeit próbálják hozzáadni a HDFS képességeihez. Az HDInsight biztosítja számunkra, hogy az egyes csomópontokon lévő elosztott fájlrendszerekhez és a Blob Storage-hoz is hozzáférhessünk.

Ahhoz, hogy ebbe a fájlrendszerbe feltöltsem az adataimat, az Azure Explorer nevű szoftvert használtam [68]. A segítségével, a felhőben kapott hozzáférési adatokat megadva, egy jól áttekinthető grafikus interfészen könnyedén tölthetünk fel és le fájlokat, illetve módosíthatjuk és tekinthetjük meg a fájlrendszer adatait.

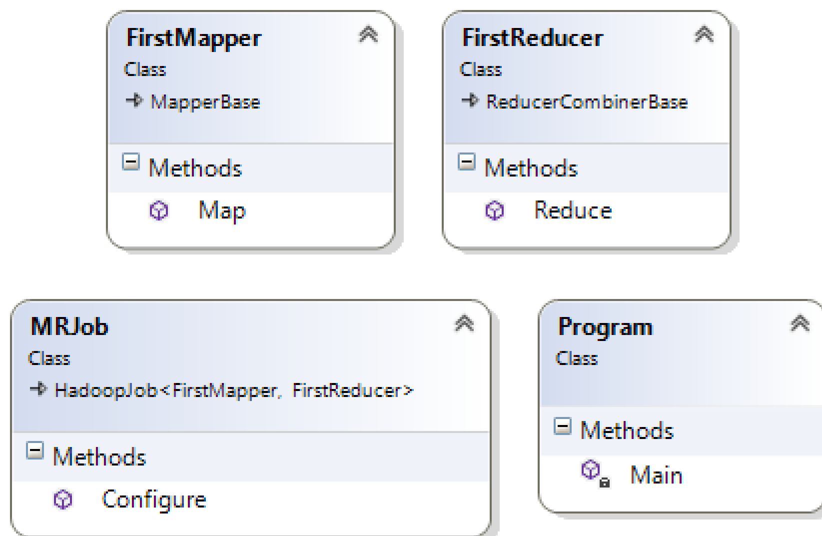
Miután az adatokat feltöltöttem a fájlrendszerre, keresnem kellett egy módot, hogy MapReduce kódot futtathassak a felhőben lévő rendszeremen. Ehhez egy Hadoop Streaming megoldást választottam, Visual Studio-ban C#-ban fejlesztettem a MapReduce algoritmusaimat. A hivatalos oldalon sok támogatást adnak ehhez a platformhoz, tehát viszonylag rugalmassá tették a segítségével az alkalmazásfejlesztést és sok tutorial-t is biztosítottak hozzá, így ráesett a választásom.

A fejlesztés megkezdéséhez két csomagot kellett telepítenem. Az egyik a Windows Azure HDInsight nevű csomag, melynek segítségével a felhőn lévő HDInsight rendszeremnek a menedzsmentjét tudtam végezni és tudtam neki MapReduce *Job*-okat beadni futtatásra. A másik csomag a Microsoft .Net Map Reduce API for Hadoop nevű volt. Ennek segítségével tudtam a MapReduce modellt használni .Net környezetben a Hadoop Streaming segítségével. Ezen csomagok birtokában fejlesztettem az alkalmazásokat.

4.1 C# programok

A futtatás eléréséhez először a kapcsolatot kellett felépítenem a felhőbeli számítógép fürtömmel. Ehhez meg kellett adnom a megfelelő metódusnak a fürtöm általam megadott nevét, a fürtömhöz használt felhasználó nevemet, az úgynevezett hadoop felhasználó nevemet, a jelszavamat, a Blob tárolóm nevét, a rendszer által generált elsődleges kulcsot, a tárolóm nevét és hogy szeretném-e hogy létrehozzon egy tárolót a megadott névvel, ha még nem létezik. A metódus által visszaadott objektumra már meglehet hívni egy MapReduce *Job* futtatását, így ezek után magát a *Job*-ot kellett megírnom. Ehhez a már jól ismert *Map* és *Reduce* függvényeket kellett megvalósítanom. Ehhez létrehoztam egy *FirstMapper* nevű osztályt mely a *MapperBase* absztrakt osztályból származott le és felüldefiniáltam a *Map* nevű metódusát. Bemeneti paraméterként egy *string* és egy *MapperContext* típusú objektumot kapott. A *string* ezek közül a konkrét feldolgozandó érték, a *MapperContext* pedig egy olyan objektum, melynek *EmitKeyValue* nevű függvényével az objektumba gyűjthetjük a *Map* metódusunk eredményét kulcs-érték párok formájában. (Az eredeti *Map* implementációkhoz képest, melyben a *Map* függvény a bemenetén is kulcs-érték part

kapott, a C# Hadoop Streaming megoldásában ez megváltozik és csak az értéket kapja meg a bemenetén.) A *MapperContext* objektumokba gyűjtött kulcs-érték párokat az összes *Map* taszk végeztével a rendszer rendezi és az egy kulcshoz tartozó értékek listáját adja a *Reduce* függvény bemenetére. Így tehát fejlesztőként nekem csak a *Reduce* függvény megírásáról kellett gondoskodnom, az imént említett lépést a rendszer automatikusan elvégzi. Ehhez a *ReducerCombinerBase* nevű osztályból kellett leszármaztatnom és a *Reduce* metódusát kellett felüldefiniálnom. (Megjegyzem, hogy itt lehetőségünk lenne egy plusz metódus megvalósítására a *map* és *reduce* fázisok között, mellyel valamiféle rendezést adhatnánk meg, hogy az adatok, hogy kerüljenek a *Reduce* metódus bemenetére.) A *Reduce* metódus kap egy *string* kulcsot, egy iterálható *string* listát, melyben a kulcshoz tartozó értékek szerepelnek, és egy *ReducerCombinerContext* típusú objektumot, melybe a *MapperContext*-hez nagyon hasonlóan az *EmitKeyValue* metódussal vehetünk fel kulcs-érték párokat. Ezek a kulcs-érték párok azonban már magának programnak a kimenetét adják meg (14. ábra).



14. ábra Osztálydiagram

Miután ezt a két osztályt definiáltam, létre kellett hoznom egy *Job*-ot, mely ezeket az osztályokat használja. Ehhez a HadoopJob nevű osztályból származtattam le, melynek generikus paraméterekként megadtam a két általam definiált osztályt. Ennek a *Configure* nevű metódusát definiáltam felül. A metódus segítségével a konfigurációs beállításokat állíthatjuk. Én alapvetően a be és kimeneti elérési utat definiáltam, tehát, hogy mely mappában lévő adatokat dolgozza fel a program és mely mappába kerüljön az eredmény (a felhőn belüli fájlrendszerben természetesen). Illetve még használtam egy hasznos beállítását, amivel megszabtam, hogy ha létezik a kimeneti mappa - amelybe az eredményét szeretné írni - a futás elején, akkor azt törölje ki. Ugyanis ha azt nem tenné, létező mappa esetén hibát dob és leáll a program futása, ugyanis nem megengedett a mappa tartalmának felülírása. Így pedig nem kell egyes futások között kézzel törölgetni a kimeneti mappát, ami például batch futások esetén is hasznos. Ezzel kész voltam a *Job* definiálásával.

Azon az objektumon keresztül pedig, mellyel kapcsolatot létesítettem a felhőben lévő fürtömmel, futtatni tudtam az elkészült MapReduce *Job*-okat.

Mikor egy *Job* sikeresen lefutott, a kimenetéül megadott mappában megjelentek a már ismert *_SUCCES* és *part-00000* fájlok.

4.2 Algoritmusok

Miután képes voltam a felhőben lévő fürtömön MapReduce *Job*-okat futtatni, implementáltam a pontos algoritmusokat C#-ban. A következő négy algoritmus született:

- Maximum-keresés
- Átlag-számítás
- Százalék-számítás
- Arány-számítás

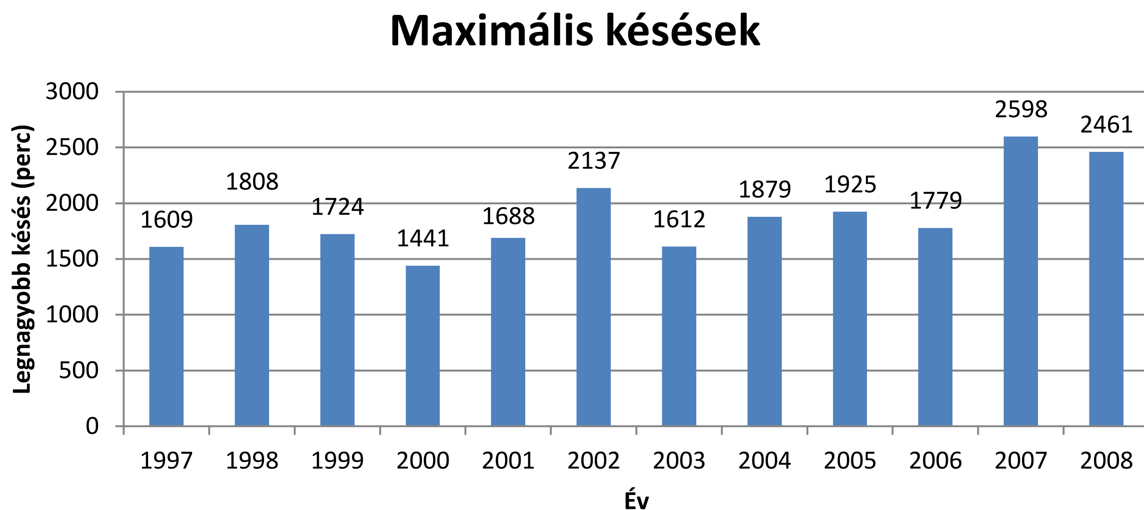
4.2.1 Maximum-keresés

Az első algoritmus minden évre megkereste, hogy abban az évben hány perc volt a legnagyobb késés. Tehát a program eredményeképpen egy olyan fájlt szerettem volna kapni, amelyben minden évhez tartozik egy szám, ami az adott évben a legtöbbet késett járat késési idejét mutatja percekben.

Miután a *csv* fájljaim a felhőbeli fájlrendszeren voltak és elkezdtem futtatni egy MapReduce kódot, a rendszer alapértelmezetten sorokra bontotta fel a tartalmukat és az egyes *Map* metódusok bemenetére ezeket a sorokat adta. Tehát a *Map* metódusom bemenetén a *string* érték egy olyan sort reprezentált, melyek egy adott járat adatait tartalmazták. Ahhoz, hogy a sort részleteiben értelmezni tudjam, a vesszők mentén darabjaira kellett bontanom a *stringet*. Ez által hozzáfértem a járat megfelelő adataihoz, melyek közül jelen algoritmusnál az évszám és a késés nagysága érdekelt. A *Map* metódus *MapperContext* objektumába pedig egy darab kulcs érték párt adtam vissza, az évszámot kulcsként és a késés nagyságát értékként. A *shuffle* fázis, amit a rendszer automatikusan elvégez, így az összes ugyanahhoz az évhez tartozó késési értéket összegyűjt és így adja a *Reduce* függvény bemenetére. Tehát a *Reduce* függvény bemenetén kapott kulcs egy adott év, a hozzá tartozó *string* értékek listája pedig az ahhoz az évhez tartozó összes késés nagysága percben. Ezek után a *Reduce* metódusban annyi volt a dolgom, hogy bejárva a listát csak a legnagyobb értéket tartsam meg. A *Reduce* metódus *ReducerCombinerContext* típusú objektumába pedig a kulcsként az adott évet, értékként pedig a megtalált legnagyobb késési időt adtam vissza. Ez által

minden évhez megtaláltam a hozzá tartozó legnagyobb késés nagyságát. (Valóban minden évhez, hiszen attól, hogy a *Reduce* metódus a bemenetén csak egy évhez tartozó késési adatokat kapunk meg, attól még futásnál több *Reduce* taszk fog indulni, még pedig annyi, hogy az összes kulcshoz tartozó értékeket feldolgozza.)

A maximum-keresési algoritmust tehát a fentiek szerint valósítottam meg, most pedig lássuk, milyen eredmények születtek a futtatásából (15. ábra).



15. ábra Maximális késések

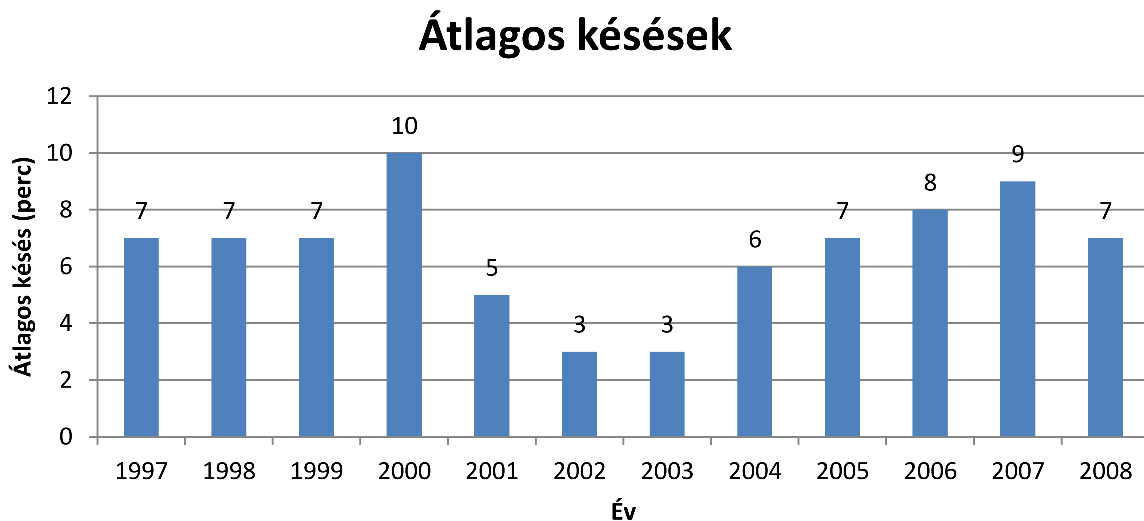
Az algoritmusokat változó input mérettel futtattam a skálázódás megfigyelésének érdekében. Ezek közül ez az 1997-től 2008-ig tartó intervallum volt a legnagyobb, amit vizsgáltam, így ennek az eredményét prezentálom. Rögtön látszik például, hogy ez alatt a 12 éves periódus alatt 2598 percet késett a legtöbbet késő járat, tehát több mint 43 órát. 2000-ben volt a legkisebb maximális késés a vizsgált intervallumban 1441 perccel, azaz még mindig több mint egy nappal. Nem volt tehát olyan év ezek között, melyben legalább egy gép ne késett volna legalább egy napot. Látjuk tehát, hogy ezek a késések minden évben egy-két nap közöttiek, melyek 1997-től 2006-ig ide-oda ingadozást mutatnak, az azonban látszik, hogy az utolsó két vizsgált év már sajnos jelentősen kiemelkedett a többi közül.

A kapott eredmények helyességét egyrészt a MapReduce modell aggregáló algoritmusokat támogató struktúrája is biztosítja, tehát hogy elegendő volt egy egyszerű maximum-keresést megvalósítani egy listára. Illetve kisebb, kézzel előállított adathalmazokon is teszteltem az algoritmus helyes működését.

4.2.2 Átlag-számítás

A második algoritmus sokban hasonlít az elsőre. Itt is adott évhez kerestem a késési időket, azonban nem a maximumukat, hanem az átlagukat számoltam ki.

A *Map* metódusom teljesen megegyezett az első algoritmusnál használttal, tehát itt is az adott évekhez gyűjtöttem az késési adatokat. Egyedül a *Reduce* metóduson változtattam. Itt ugyanis az értéket végigiterálva egyrészt összeadtam őket, másrészt megszámoltam őket. A két érték hányadosából pedig megkaptam, hogy átlagosan mennyit késtek adott évben a járatok. Lássuk az eredményeket (16. ábra):



16. ábra Átlagos késések

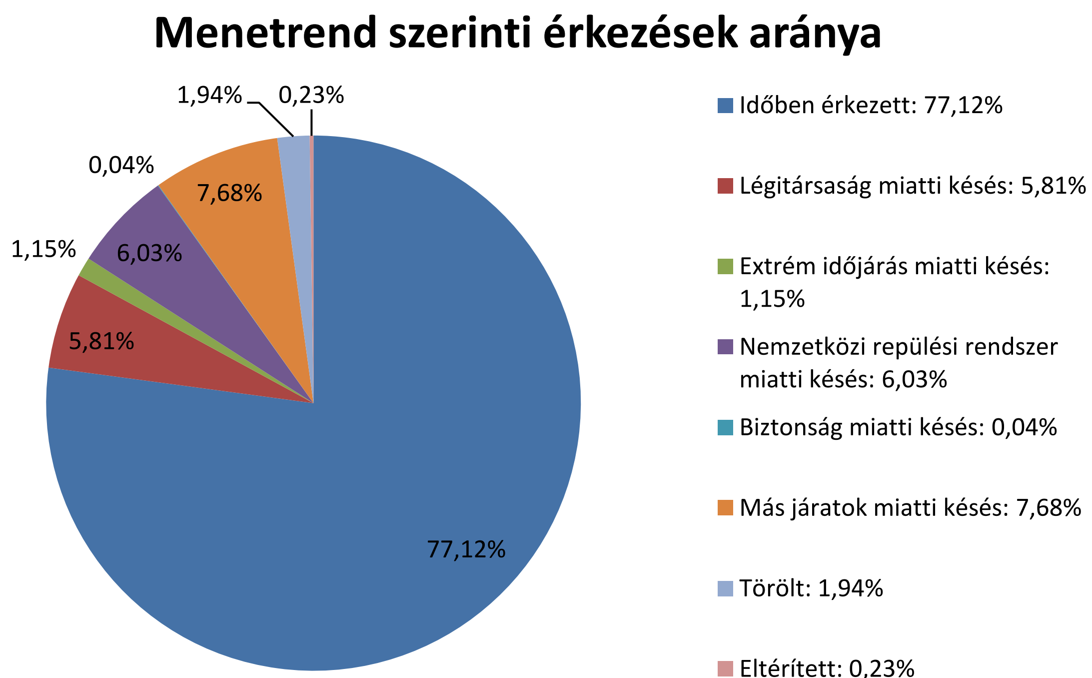
Ennél az algoritmusnál is a legbővebb intervallumon végzett futás eredményét mutatom be. Látszik, hogy a 2000-es év volt a legrosszabb olyan szempontból, hogy átlagosan 10 percet késtek a járatok. Érdekes, hogy ehhez képest éppen 2000-ben volt a legkisebb maximális késés, ezek közül az évek közül az első algoritmus alapján. A 2002 és 2003-as évek bizonyultak a legjobbnak, amikor csak 3 percet késtek átlagban a gépek. (Pedig 2002-ben volt az egyik legnagyobb késés a maximum-keresés algoritmus alapján.) Ezek után az évek után folyamatosan növekedés történt a késési percekben, azonban épp az utolsó mért évben ez valamelyest mérséklődött.

Az algoritmus helyességét, itt is kisebb általam előállított adathalmazokon teszteltem.

4.2.3 Százalék-számítás

A harmadik algoritmus segítségével azt szerettem volna megkapni, hogy a járatoknak hány százaléka érkezett meg időben, lett törölve, lett eltérítve, és hány százaléka milyen pontos ok miatt késett.

Ehhez a *Map* metódusban több adatot kellet kinyernem a sorokból. Szükségem volt az egyes konkrét késési okok percben megadott értékére, illetve, hogy az adott járat törölve vagy eltérítve lett-e. (Ezek az adatok explicit kinyerhetők voltak a sorokból.) Ezeket a *MapperContext* objektumba minden esetben ugyanazzal az általam kreált egyszerű *string* kulccsal adtam hozzá, illetve értékként ezekből a kinyert adatokból fűztem össze egy *stringet*, melyet majd a *Reduce* metódusban szétbontottam az egyes értékekre. A *Reduce* metódus tehát a bemenetén az összes sorhoz tartozó adatokat egy kulcshoz rendelten kapta meg. Megvizsgáltam először, hogy az összes járat közül hány lett törölve, eltérítve, illetve érkezett meg időben. (A hivatalos oldal szerint egy gép, akkor érkezett meg időben, ha a tervezett érkezési időn legfeljebb 15 percen belül valóban megérkezett.) Ezeknek így megkaptam a százalékát és a maradék járat késett. Az egyes okok miatt késett perceket külön-külön összeadtam és ebből kiszámoltam, hogy az egyes okok miatti késés hány százalékban volt jelen. A kulcshoz ezeket a százalékos adatokat gyűjtöttem a kimeneti objektumba. Ezek alapján a következő eredményeket kaptam (17. ábra):



17. ábra Menetrend szerinti érkezések aránya

Ez a 2006 januárjától 2008 decemberéig terjedő időszakról készült kimutatás. (A konkrét késési okok perceit 2003 júniusa óta jegyzik. Ennél korábbi időszakról ezért

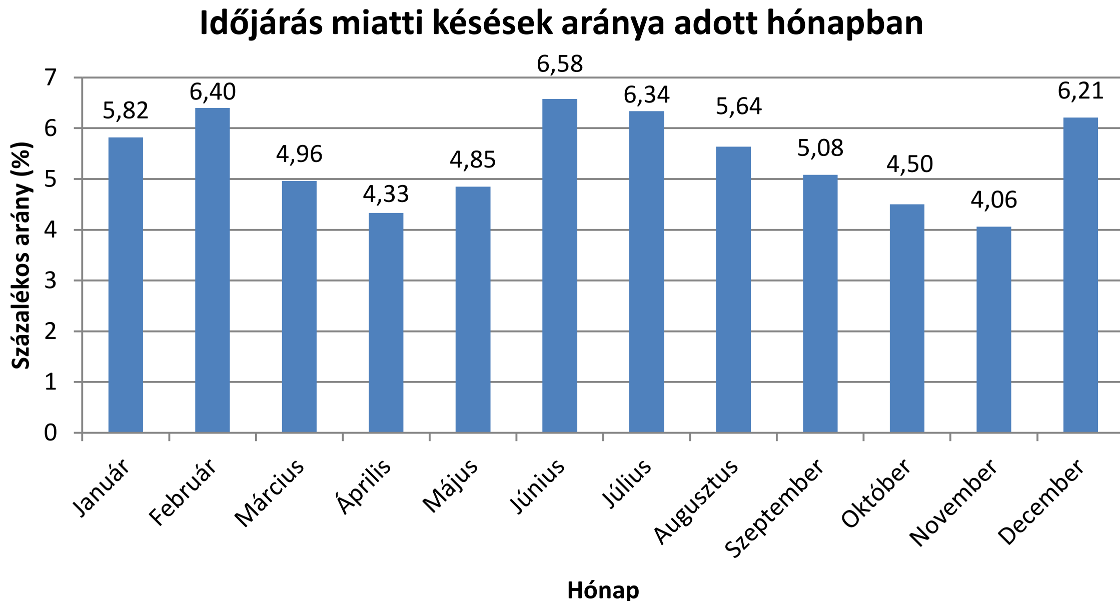
nem érdemes ezt a kimutatást elvégezni. A 2003 június utáni időpontok közül pedig a 2006. januárjával kezdődő intervallum volt az első, amire teszteltem az adatokat.)

Amit ebből láthatunk, hogy majdnem minden 400. repülőt eltérítették és majdnem minden 50.-et törölték. 100-ból átlagosan 98 járat elérte az úti célját, de csak 77 érkezett meg időben (tehát 15 perc késésen belül), a maradék 21 pedig különböző okok miatt késett. Közülük a legtöbbet más késő járatok miatt, a légitársaság hibájából vagy a nemzetközi repülési rendszer miatt késnek. Ez utóbbiba tartoznak a repülőtereken végzett teendők, a légi forgalom miatti késések és a nem extrém időjárási körülmények miatti késések, melyek nem a konkrét repülési időt növelik, hanem a rendszerhez kapcsolódó teendőket.

Ugyanezeket az eredményeket a hivatalos oldalon is megtaláljuk [69]. Beállítva tehát 2006 januárjától 2008 decemberéig az intervallumot, ugyanazokat az eredményeket kellene kapnunk. Az eltérített és törölt járatok százaléka századra pontosan megegyezik az általam kapottakkal, azonban az időben érkezett járatok arányában körülbelül 2%-kal eltér és emiatt kisebb eltérések vannak a konkrét késési százalékokban is. (Ez a többi még általam futtatott intervallumokra is így van, melyeket összehasonlítottam a hivatalos oldal eredményeivel, hogy a törölt és eltérített adatok pontosan megegyeznek azonban az időben érkezettek arányában van egy 2% körüli különbség.) A törölt és eltérített járatok számolására tehát úgy tűnik helyes algoritmust implementáltam, az időben érkezettek megállapítására viszont úgy tűnik némileg eltérőt. (A hivatalos oldal szerint akkor tekinthető egy járat késettnek, ha legalább 15 perccel később érkezett, mint a menetrend szerinti érkezése. Ezzel én is így számoltam, de valamiért ezekre nem kaptam teljesen pontos eredményt.) Ezt az eltérést azonban elég kicsinek ítéltem meg ahhoz, hogy elfogadjam az algoritmust, hiszen a fontosabb célom nem ezeknek az eredményeknek a megkapása volt, hanem a MapReduce skálázódásának a vizsgálata.

4.2.4 Arány-számítás

A negyedik algoritmussal minden hónapra kiszámoltam, hogy az abban a hónapban történt összes késésnek hány százaléka történt extrém időjárási követelmények miatt. Ehhez a *Map* metódusban a hónap, késés és extrém időjárás miatti késés adatokat kellett kigyűjtenem. A hónap volt a kimeneti objektumban a kulcs és a késés és extrém időjárás miatti késés pedig egy *stringgé* összefűzve a kulcs. A *Reduce* metódusban minden hónaphoz kiszámoltam, hogy összesen hány perc késés volt a hónapban és összesen hány perc volt extrém időjárás miatt, a kettő hányadosából pedig megkaptam, hogy az adott hónapban a késések hány százaléka történt extrém időjárási körülmények miatt. Az eredmények a következők: (18. ábra):



18. ábra Extrém időjárás miatti késések aránya adott hónapban

A vizsgált időszak szintén 2006 januárjától 2008 decemberéig tartott az előzővel megegyező okok miatt. (A késések a 4-7% intervallumba esnek. Egy adott hónapban valójában nem csak ennyi százaléka a késéseknek történik bármilyen időjárás okozta késés miatt. A nemzetközi repülési rendszer okozta késéseknek nagy része származik időjárás okozta késésekből. Valójában 35-50% körül van a bármilyen időjárás miatti késések aránya. Ebben az adathalmazban arról nem volt adatom, hogy a nemzetközi repülési rendszer okozta késéseknek hány százaléka történik időjárás miatt, ezért én csak az extrém időjárási körülmények miatti késéseket vizsgáltam [70].) Az eredményből jól látszik, hogy a téli és nyári hónapokban jóval jelentősebb az extrém időjárás okozta késés. Általában nagyobb váltás a késési százalékokban évszakok váltásakor következik be. Extrém időjárási körülmények szempontjából június a legkevésbé ajánlott utazási hónap, november pedig a leginkább preferált.

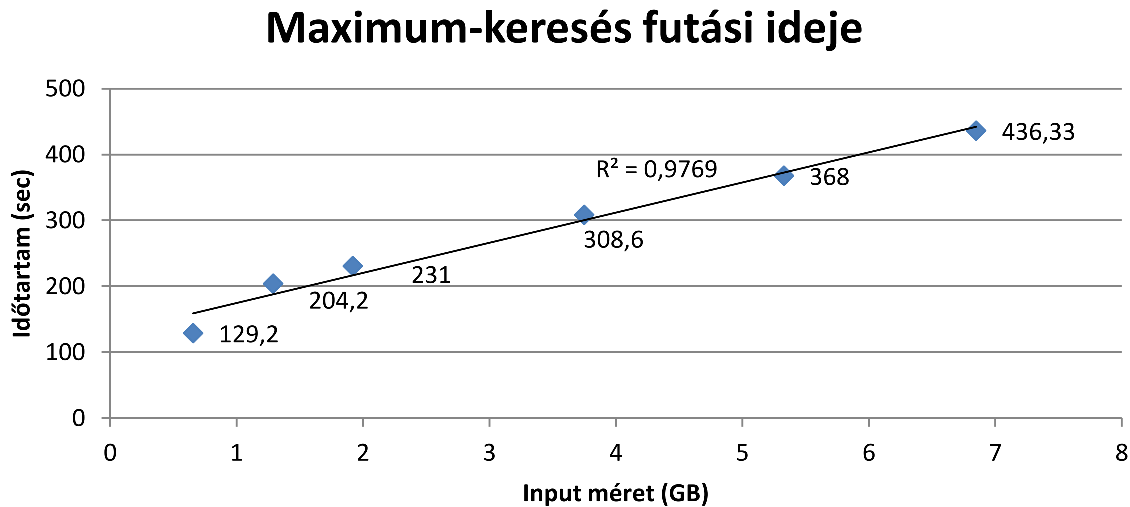
Ezen algoritmus helyességét szintén kisebb adatokon teszteltem.

Láthattuk tehát, hogy ezzel az újfajta programozási modellel a szokványostól eltérően lehet programozási feladatokat megoldani. Vannak olyan problémák, melyekre nehézkes MapReduce algoritmus létrehozni, hiszen a modell csak nehezen teszi lehetővé. (A különböző Hadoop-on belüli projekteknek ezért is van létjogosultsága, mert segítségükkel más módokon sokkal egyszerűbben megfogalmazhatók bizonyos problémákra az algoritmusok.) Az ezekhez hasonló aggregáló jellegű problémák megoldására azonban kifejezetten alkalmas a modell. Programozói szempontból is egyszerű megvalósítani az algoritmusokat és hatékony sebességgel is tudja lefuttatni őket a keretrendszer.

4.3 Skálázódás

Adott volt tehát a 4 algoritmus, melyeket futtathattam. A célom a MapReduce skálázódásának vizsgálata volt. Ehhez ismét kerestem olyan változókat, melyeket futások között változtatni tudtam. Egyik volt tehát maga az algoritmus. A másik pedig az input mérete, melyet a beadott *csv* fájlok számával tudtam változtatni. Összesen 6 féle input méretet használtam. Alapvetően ennek a két paraméternek a változtatásával teszteltem a futásokat egy 13 számítógépből álló számítógép fürtön. Minden esetet ötször futtattam le, hogy minél pontosabb mérési eredményeket kaphassak. (Ötször elég volt, hiszen a futási idők minimálisat változtak a futtatások között.) A mérési eredményeknek pedig az átlagát számoltam ki a futások összehasonlítására. A következő futási időket kaptam:

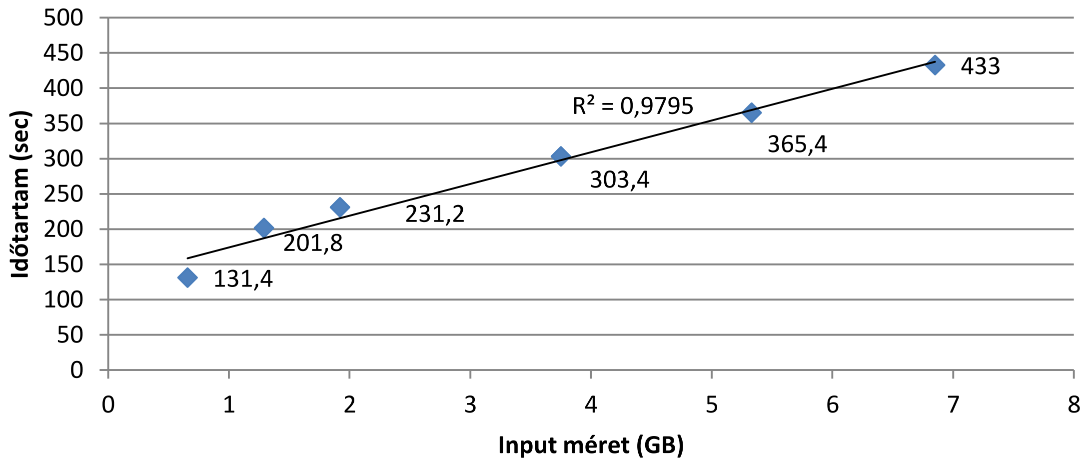
Az első algoritmus futási ideje az input méret függvényében (19. ábra):



19. ábra Maximum-keresés futási ideje az input függvényében

Az ábrán látható hat négyszög jelöli, hogy az adott input mérethez hány másodpercig tartott az átlagos futási idő. Az ábrából látható, hogy az input méret növelését majdnem 7 GB-ig folytattam, tehát valóban nagy-méretű adathalmazon teszteltem az algoritmusokat. Látható, hogy a legkisebb adatméret esetében a futási idő nem sokkal több, mint két perc volt, a legnagyobb adathalmaz elérésekor pedig már több mint hét perc. A mért eredményekre egy lineáris trendvonalat illesztve és az R^2 értékét megjelenítve láthatjuk, hogy az input méret növelésével a futási idő jó közelítéssel lineárisan nő. A Hadoop keretrendszer ismeretében, ezt az eredményt is vártam, hiszen keretrendszer komoly előnye más rendszerekkel szemben, hogy lineárisan skálázódik. Ezt tehát ki is tudtam mutatni.

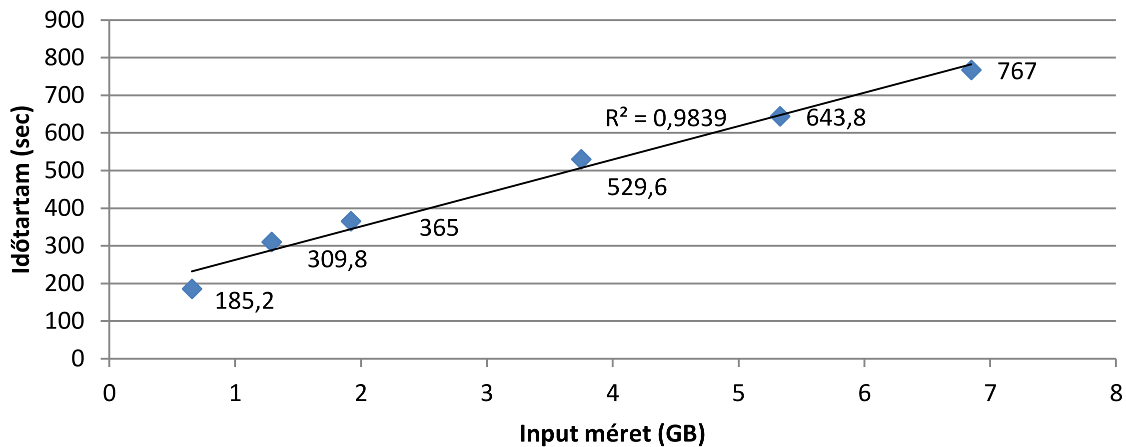
A második algoritmus futási ideje az input méret függvényében (20. ábra):



20. ábra Átlag-számítás futási ideje az input függvényében

A futási idők itt is több mint két perctől több mint hét percig terjednek. A lineáris trendvonal pedig itt is jól illeszkedik.

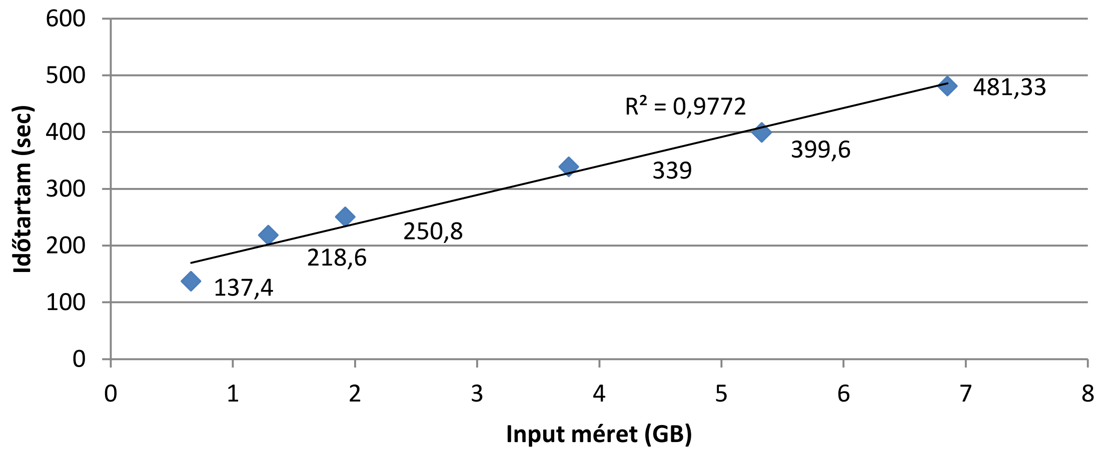
A harmadik algoritmus futási ideje az input méret függvényében (21. ábra):



21. ábra Százalék-számítás futási ideje az input függvényében

Ennél az algoritmusnál a futási idő több mint három perctől indul és majdnem tizenhárom perces futást mértem átlagban a legnagyobb inputra. A trendvonalra pedig jól illeszkedik.

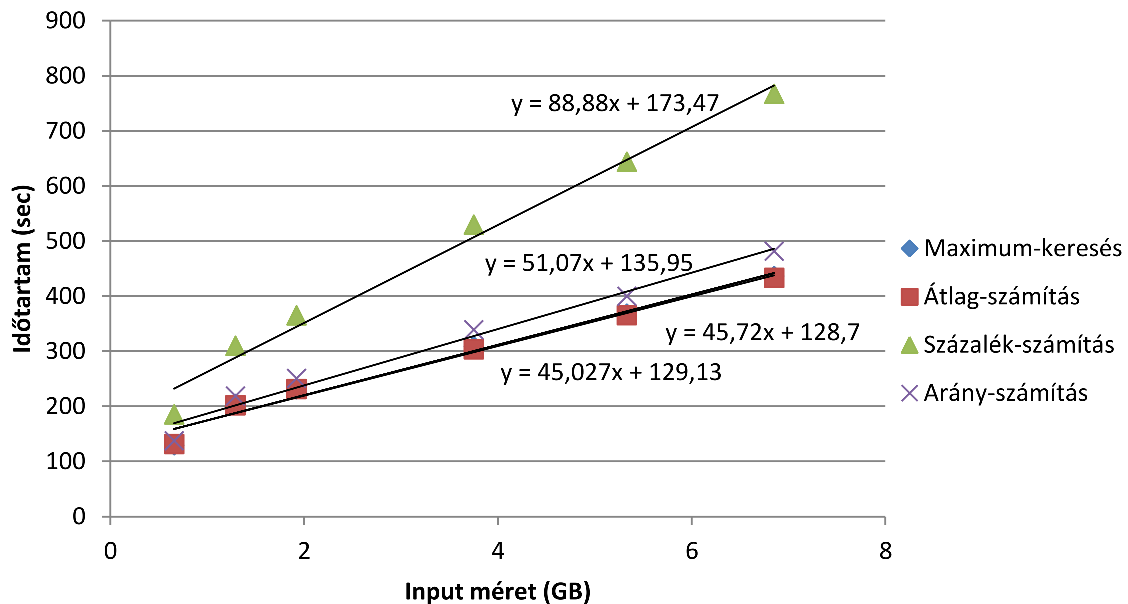
A negyedik algoritmus futási ideje az input méret függvényében (22. ábra):



22. ábra Arány-számítás futási ideje az input függvényében

Ennél az algoritmusnál két percnél nagyobb futási időről indultunk és az utolsó futás ideje pedig éppen több mint nyolc perc volt. A trendvonalra való illeszkedés ebben az esetben is jó.

A négy algoritmusnál tehát látható a futási idő lineáris-növekedését, amit vártam is a MapReduce elosztott működése miatt, de ennél érdekesebb, hogy egymáshoz viszonyítva, hogy futottak le (23. ábra).



23. ábra Algoritmusok futási ideje

A maximum-keresés és az átlag-számítás algoritmusok (ahogy az egyedi ábrájukon is láthattuk) nagyon hasonló eredményeket adtak és gyakorlatilag azonos futási idejük lett. Az egyeneseik egyenlete alapján is láthatjuk, hogy közel megegyeznek. A *Map* metódusa teljesen megegyezett a két algoritmusnak, csak a *Reduce* metódusok különbözött. A *Reduce* taszkok bemenetére tehát ugyanazok az adatok kerültek a két algoritmus esetében. Az értékeken ugyanúgy végigment mind a két algoritmus. Az egyik mindig csak egy értéket tartott meg közülük (a maximumot), a másik pedig egy változóba gyűjtötte az összegüket és a végén még egy osztást végzett, ami érezhetően elhanyagolható különbség. A két algoritmus nagyon hasonló futási ideje tehát várható volt a *Map* metódusuk egyezése és a *Reduce* metódusuk futási időben hasonlósága miatt. Az arány-számítás algoritmus meredeksége már valamivel nagyobb az előző két algoritmus meredekségéhez képest. Itt a *Map* metódusban két értéket is kiszedtem a kulcshoz (késés nagyságát és az időjárás miatti késés nagyságát). Ezeket összefűztem egy *stringgé* és ezt az értéket adtam tovább a *shuffle* fázisnak, tehát nagyobb adatokat kellett összerendeznie. A *Reduce* metódusban pedig költséges *string* (*split*) művelettel bontottam újra részekre. Itt a meredekség eltérése még nem volt túl számottevő, azonban a százalék-számítás algoritmusnál jelentősen megugrott a futási idő. Itt már nyolc adatra volt szükségem minden sorból, így belőlük fűztem össze egy *stringet* értéknek. Ezt a jóval nagyobb méretű adatot kellett a *shuffle* fázisnak rendezni és a *Reduce* metódus bemenetére adni. Itt pedig ismét költséges *split* műveletet használtam ráadásul. Ennyivel több adat átadásakor már jelentősen nőtt a futási idő is.

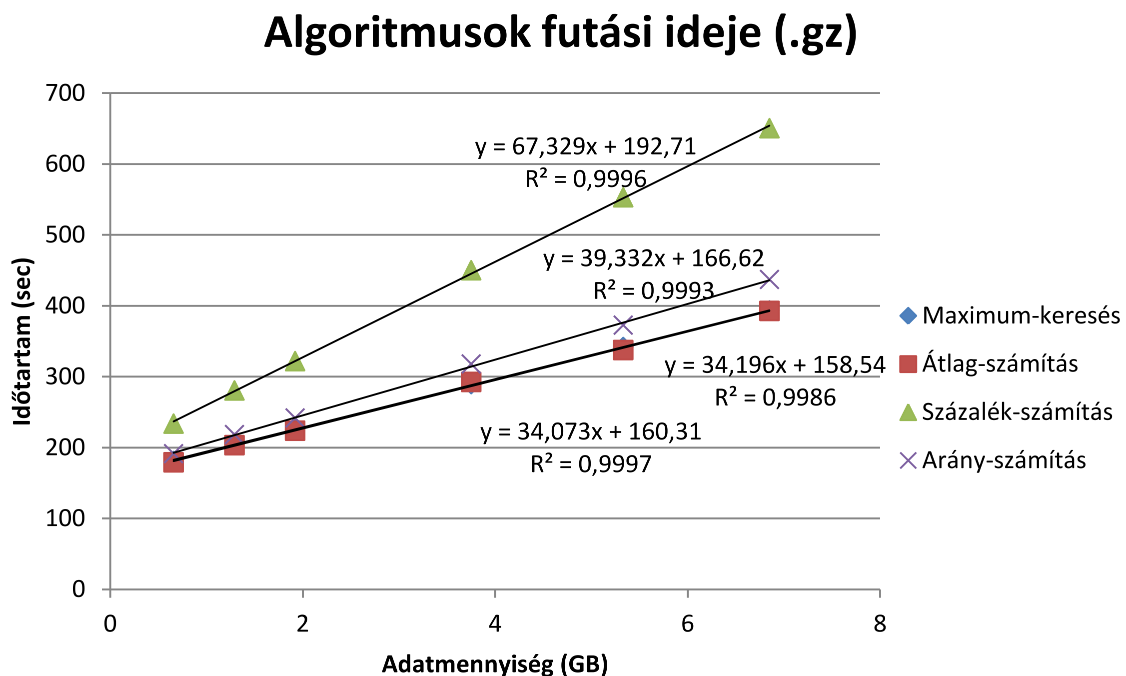
5. Optimalizáció

Láttam tehát, hogy az algoritmusok az input méret növekedésével hogyan futnak le, illetve hogy az algoritmusok egymáshoz képest milyen tendenciákat mutatnak. Felmerült azonban, hogy ezeket az elsőre megírt, célt megvalósító algoritmusokat, hogyan lehetne még optimalizálni is. A Hadoop *Job*-ok optimalizációjának nem mindegy, hogy hogyan állunk neki. Lehet például, hogy sok új számítógéppel növeljük a fürtünk méretét és a teljesítmény mégis alig-alig nő. Ennek oka vélhetően az, hogy a szűk keresztmetszet nem a *cpu* teljesítmény volt, hanem valami más. A Hadoop *Job*-ok szűk keresztmetszete lehet a *cpu*, I/O, hálózat, RAM általában. Ezek közül pedig leggyakrabban az I/O olvasások és írások, illetve a hálózaton való adattovábbítás a szűk keresztmetszet Hadoop *Job*-ok esetén. Egy Hadoop program optimalizációjakor érdemes tehát a szűk keresztmetszetet meghatározni, hogy jelentősen tudjuk növelni a teljesítményt. Az én esetemben csak sejtettem, hogy nálam is az I/O és a hálózat lehet a szűk keresztmetszet. A fürtben lévő számítógépek számát próbaképpen lecsökkentettem és alig lassabb futási időt kaptam, ebből arra következtettem, hogy nem a *cpu* a szűk keresztmetszet. A futási adatokat vizsgálva pedig meg tudtam nézni, hogy melyik futás mennyi *heap* memóriát használt összesen. Az input növelésével ez a szám mindig szépen nőtt, nem volt olyan, hogy valaminél már ne kapott volna több memóriát. Ebből azt gondoltam, hogy nem is a RAM a szűk keresztmetszet. Az egyes algoritmusok nem egyforma adatmennyiségeket olvastak be a futásuk során, mert például miután a maximum-keresés algoritmus beolvasta a neki szükséges év és késés adatot az összes maradék adatot a sorban már nem is olvastattam be vele az algoritmusban. Az arány-számítás algoritmus ezzel szemben például az időjárás okozta késés adattagot is beolvasta, ami majdnem az utolsó volt a sorokban, így összességében jóval több adatot olvasott be mégsem volt sokkal lassabb. Ebből arra következtettem, hogy nem is az I/O igazán a szűk keresztmetszet. Sokkal inkább a hálózat. Az arány-számítás algoritmus inkább amiatt lehetett valamivel lassabb a maximum-keresésnél például, mert több adatot adott ki a *Map* metódus kimeneti objektumában, így több adatot kellett a hálózaton átküldeni a *reduce* taszk bemenetére. Ez főleg a százalék-számítás algoritmusból látszott, mely jóval lassabb volt a többinél sejtésem szerint amiatt, hogy jóval több adatot kellett a hálózaton küldözgetnie. A százalék-számítás alig több adatot olvasott be, mint az arány-számítás algoritmus és mégis sokkal lassabb volt. Ebből is gondoltam tehát, hogy a hálózat lesz a szűkkeresztmetszet és nem az I/O.

A hálózaton történő adatküldés lassúságának a problémájára pedig az egyik lehetséges megoldás, hogy ha tömörítjük az adatokat és akkor kevesebb adatot kell küldeni, tárolni és I/O műveletekkel kezelni. Másrészt viszont növeli a *cpu* időt a ki és betömörítés. A tömörítésre Hadoop-ban három lehetőség van. Lehet a bemenő adatot tömöríteni, lehet a *map* és *reduce* fázis közötti adatot és lehet a *reduce* fázis kimenetén lévő adatot. A *reduce* fázis kimenetén, akkor lenne érdemes, hogy ha tárolására helyet akarnánk spórolni, vagy ha újabb *Job* futtatásának lenne ez az eredmény az inputja.

A *map* és *reduce* fázis között gyorsabb kódolókat érdemes használni, mint például az LZO vagy a Snappy [71][72]. A *map* fázis előtt pedig nagy adatokat érdemes tömöríteni. Én a jelenlegi dolgozatomban ez utóbbival próbáltam a futások sebességét növelni, mert amúgy is az input méret változtatásával dolgoztam eddig is és növekvő input méret mellett, egyre jobban javuló teljesítményre számítottam. A *map* metódusom nagy részében pedig csak I/O műveleteket végeztem, hiszen olvastam be az adatokat, jelentős *cpu* időt igénylő feladatot nem végeztem ott. Ezért érdemesnek tartottam az input adatok tömörítését kipróbálni és ezzel a hálózaton történő adattovábbítás és az I/O műveletek teljesítményét növelni.

A tömörítéshez *gzip* tömörítést használtam. Ez a fajta tömörítés nem támogatja az adat szétvágását. Ez azt jelenti, hogy nem lehet benne olyat csinálni, hogy a folyam egy adott pontjától kezdve olvasni benne. Emiatt ha egy *gzip* fájl a Hadoop fájlrendszerén például tíz blokkot foglal el, akkor mind a tíz blokkot egy *map* fogja feldolgozni. Emiatt nem érdemes nagy *gzip* fájlokat feldolgoztatni a rendszerrel jobb sebességet várva. A gondolatom az volt ezek után, hogy ha lenne egy nagy fájlom, azt nem lenne érdemes egyben *gzip*-elni és feldolgoztatni, de ha részekre tudnám vágni és azokat *gzip*-elném be, akkor valóban nyerhetek teljesítményt. Az én adataim azonban éves bontásban voltak meg és körülbelül 500MB méretűek voltak egyenként. *Gzip*-elve pedig 100-200MB körül. Így tehát maximum 1-2 blokknyi helyet foglaltak el a fájlrendszeren, így úgy döntöttem így fogom használni őket. (Egyébként kipróbáltam azt is, hogyha fogok több évnyi *csv* fájlt és egy *gzip* állományt képzek belőlük és arra futtatom a kódot, akkor valóban jóval lassabb futást kaptam, ahogy vártam.)



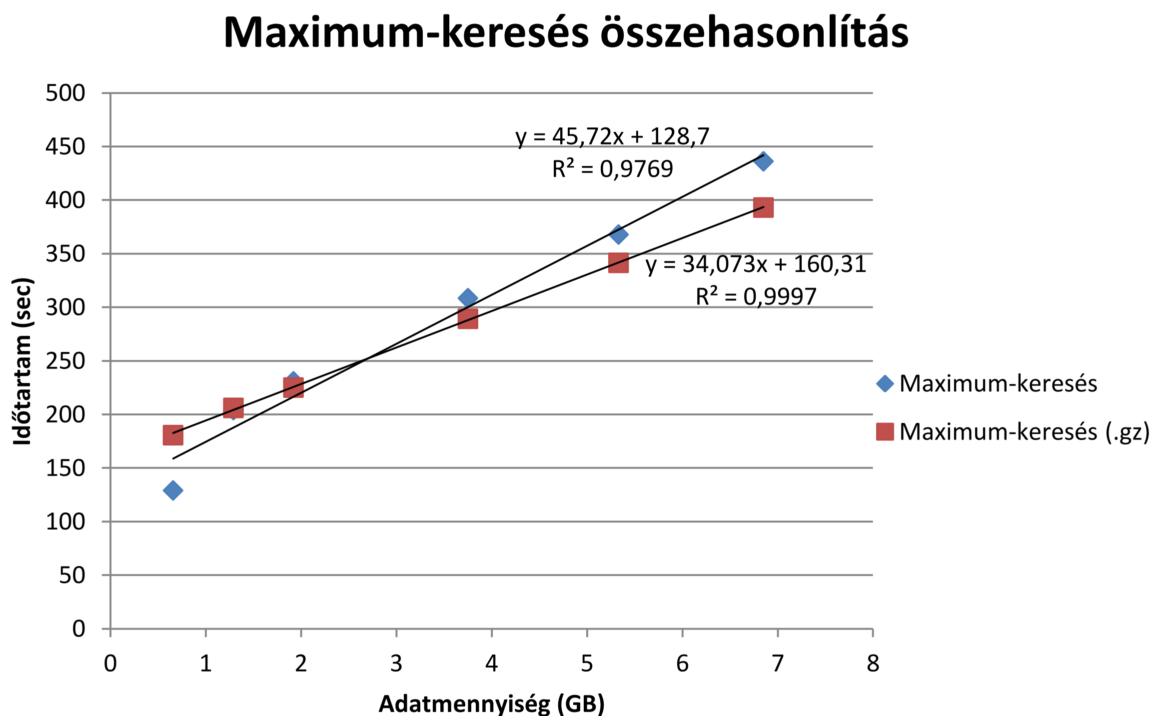
24. ábra Algoritmusok futási ideje a feldolgozott adatmennyiség függvényében *gzip*-elt input fájlokkal

Tehát a *csv* fájlokat egyenként *gzip*-elve ugyanazokkal az input adatokkal futtattam, mint amivel az első futtatásoknál is, csak most a *csv*-k helyett a *gzip*-ekkel. Egyedül arra kellett figyelnem, hogy *.gz* kiterjesztést adok a fájloknak, onnantól kezdve az HDInsight automatikusan fel tudta dolgozni. A futások eredményeképpen ugyanazokat az adatokat nyertem ki az adathalmazból természetesen minden beadott inputra, mint amit az első futtatásoknál is. Most pedig nézzük, hogy néztek ki ezek a futások (24. ábra).

(Az x tengelyen tehát, a valóban feldolgozott adatmennyiség szerepel, mely megegyezik a nem tömörített esetekben lévőkkel. Az input méret ezekben az esetekben jóval kisebb a tömörítés miatt, de mikor a feldolgozásra kerül a sor, az adatot kitömöríti a rendszer és ugyanakkora adatmennyiséget dolgoz fel ezáltal.) Az R^2 értékek alapján ezek a futások is jól illeszkednek lineáris trendvonalra (még jobban, mint az első futtatásoknál). Tehát ahogy várható volt, attól, hogy *gzip*-elt fájlokkal dolgozunk a MapReduce lineáris skálázódása megmarad. Ami még látható az ábrából, hogy a futások egymáshoz viszonyított arányai megmaradtak, hiszen a konkrét algoritmusok nem változtak, csak mindegyik futásnak kisebb nagyságú adatokat kellett eljuttatni a *map* taszkok bemenetére és feldolgozni.

A fontosabb, hogy az eredeti futásokat a *gzip*-elt futásokkal összehasonlítsuk és megvizsgáljuk, hogy az input méret növekedésével valóban egyre többel gyorsabb futási időket kapunk.

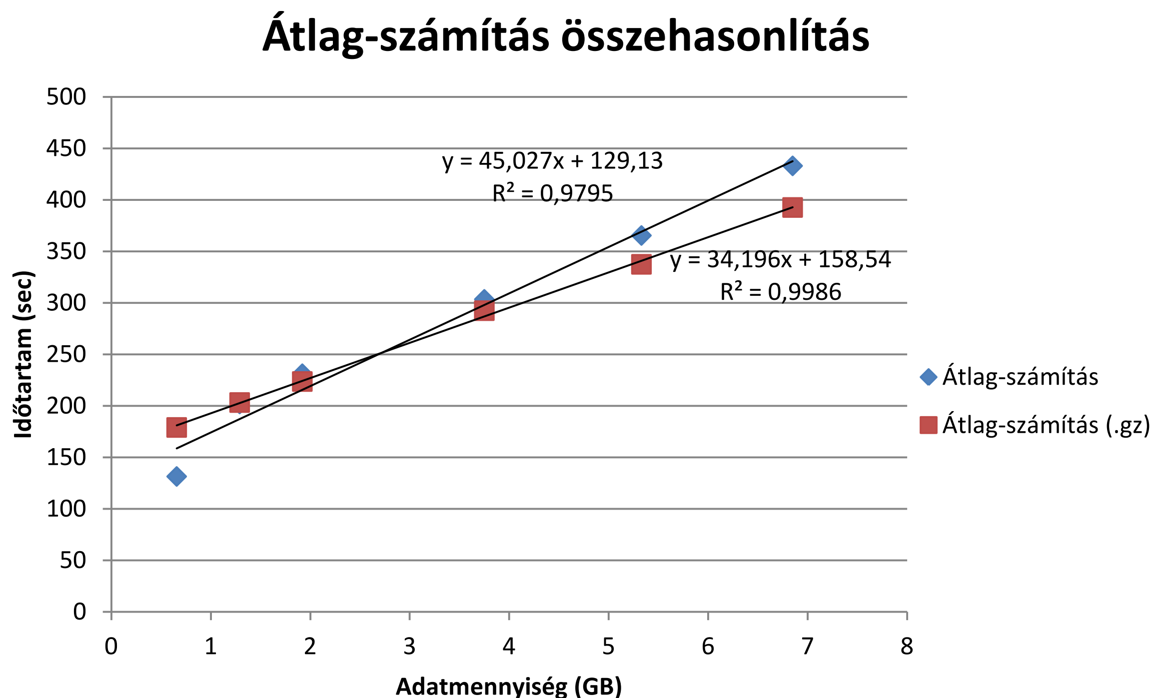
Az első algoritmus összehasonlítása (25. ábra):



25. ábra Maximum-keresési algoritmus összehasonlítása az input típusa alapján

A *gzip*-elt futás esetén láthatóan valamivel jobb linearitást kapunk. Ennek oka az első input mérettel történő futás miatt lehet, ahol is láthatóan a mért érték a trendvonal alá esik valamivel. A MapReduce *Job*-ok indítási overhead-je általában elég nagy, így kis input esetén valószínűleg eltérő meredekségű egyenest kapnánk. Az első input méretnél még valószínűleg emiatt van a lineáristól némileg eltérő eredmény, utána azonban már nagyon jó közelítéssel lineáris eredményeket kapunk. Az első inputnál még láthatóan nem érte meg tömöríteni a bemeneti fájlt, nem volt elég nagy, többe került a kitömörítéssel járó *cpu* művelet. A második input méretnél egyenlítődött ki a futási idő, tehát körülbelül már 1,5GB-os adatméretnél. Ezután pedig láthatón egyre több másodperccel hamarabb végzett a *gzip*-elt megoldás, mint az eredeti. A meredekségek aránya alapján az eredeti meredekség körülbelül 75%-ára esett vissza a meredekség a *gzip*-elt futásokra.

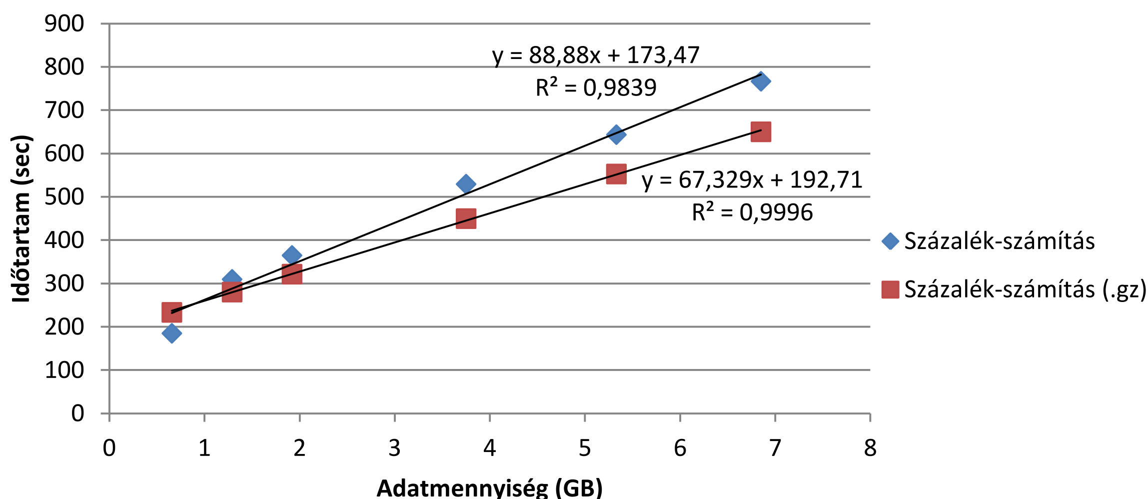
Az második algoritmus összehasonlítása (26. ábra):



26. ábra Átlag-számítás algoritmus összehasonlítása az input típusa alapján

Az átlag-számítás algoritmus, ahogy már a korábbiakban is láthattuk, gyakorlatilag egybeesik a maximum-keresési algoritmussal, így ugyanazok mondhatók el róla, mint az előző összehasonlításnál. A linearitás jobb a *gzip*-elt esetben, kis inputra nem éri meg a tömörítés, egyre növekvő input méretre viszont egyre jobban megéri. A meredekség itt körülbelül 76%-ára esik vissza az aránypár alapján.

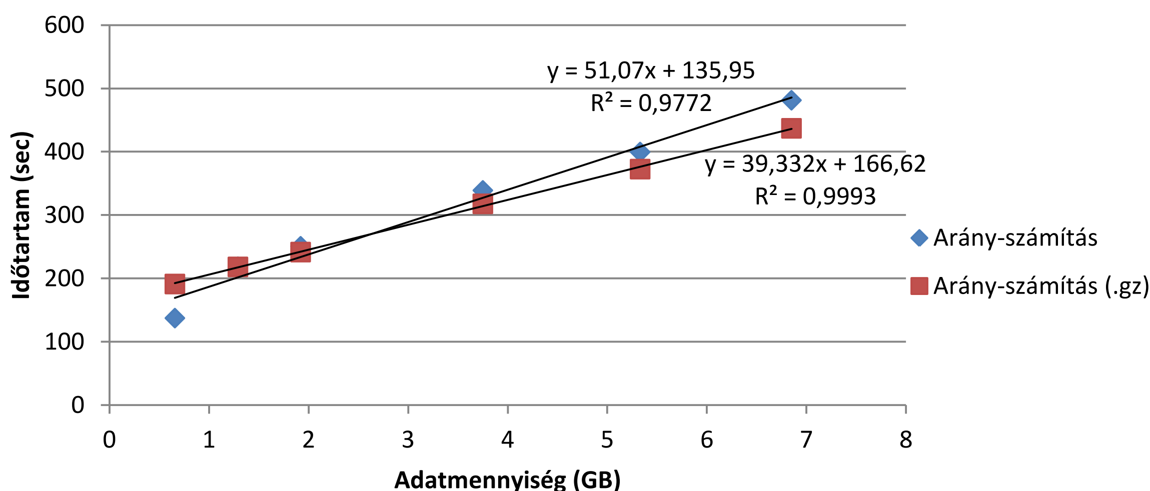
Az harmadik algoritmus összehasonlítása (27. ábra):



27. ábra Százalék-számítás algoritmus összehasonlítása az input típusa alapján

A linearitás szintén a *gzip*-elt esetben a jobb. Kis inputra valamivel ennél az algoritmusnál sem éri meg a tömörítés a plusz *cpu* idő miatt. Körülbelül 1GB körül egyenlők a futási idők és innentől egyre jobb a *gzip*-elt esetben. A meredekségek aránya alapján 76%-os itt is a visszaesés.

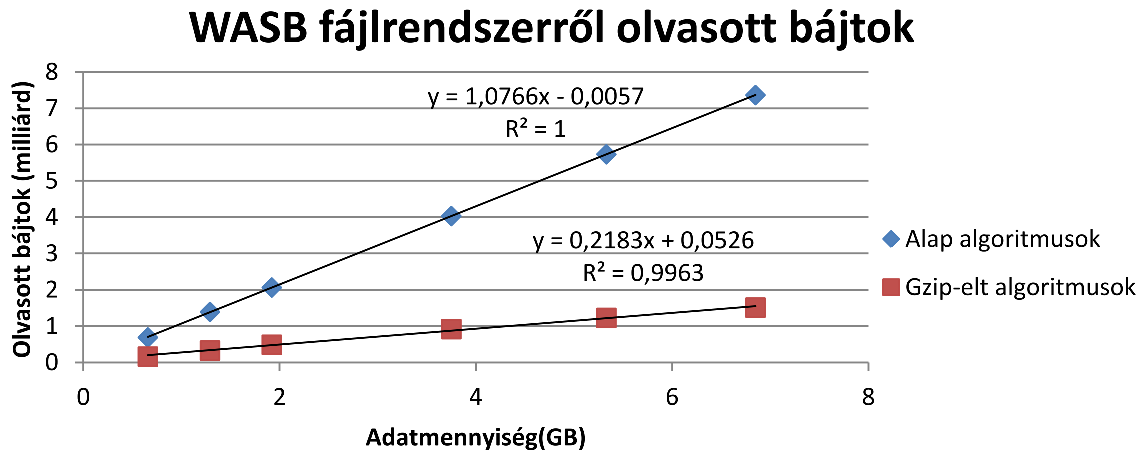
A negyedik algoritmus összehasonlítása (28. ábra):



28. ábra Arány-számítás algoritmusok összehasonlítása az input típusa alapján

Az utolsó algoritmusnál csak körülbelül 2,5GB mérettől érte meg a tömörítést használni. A meredekség pedig 77%-ára csökkent a *gzip*-elt megoldással.

A rendszerben tárolt logok-ból a futási időn kívül még a WASB fájlrendszerről olvasott bájtok számát elemeztem. Kitömörítés után a *map* taszkoknak a helyi számítógépeken már a teljes adathalmazt kell olvasniuk, de a WASB fájlrendszerről először még a tömörített adatokat kell csak olvasniuk *gzip*-elt esetben. Az ábrán az látható, hogy ezen bájtok száma mennyivel csökkent (29. ábra):



29. ábra WASB fájlrendszerről olvasott bájtok száma az alap és *gzip*-elt algoritmusok esetében

A jelentős méret csökkentéssel sok I/O műveletet megspóroltunk, ezáltal nagyban megnövelve a teljesítményt.

A mérésekkel tehát bizonyítottam, hogy nagy adathalmazok esetén a futási időt körülbelül 76%-ára csökkenthetjük csak azzal, hogy a bemeneti adatainkat *gzip* formátumba tömörítjük, még pedig olyan kis méretekben, melyek csak 1-2 blokkot foglalnak el a fájlrendszeren. (Több blokk esetén értelemszerűen nem érhető el ilyen hatékonyság. A futások között a meredekségen kívül a tengelymetszet is eltért, mégpedig a *gzip*-elt megoldásnál volt nagyobb, így emiatt a 76%-os javulásnál valamivel gyengébb értéket tudunk elérni ilyen nagyságrendű adatoknál. Nagyon nagy adathalmaz feldolgozása esetén azonban ez az érték elhanyagolhatóvá válhat már.)

6. Összefoglalás

A munkám keretében először is irodalomkutatást végeztem Hadoop témakörben. A technológia még nagyon új, de így is rengeteg irodalom érhető el róla az interneten. Az elméleti tudás mellett gyakorlatban is próbáltam több helyről megismerni. Végül az HDInsight nevű rendszert kezdtem el használni a tesztelésre. Kiválasztottam egy érdekes adathalmazt, melyből megpróbáltam hasznos eredményeket kigyűjteni a Hadoop keretrendszer használatával. Az HDInsight segítségével C# kóddal ez sikerült is és megpróbáltam a MapReduce skálázódását is megvizsgálni. Az input méretének és az algoritmusnak a változtatásával kaptam meg az eredményeimet. A MapReduce modell lineáris skálázódását nagyon jó közelítéssel ki tudtam mutatni. Az algoritmusok futási idejei, illetve más metrikái alapján pedig próbáltam a rendszer szűk keresztmetszetére következtetni. A hálózaton való adatküldést találtam szűk keresztmetszetnek. Optimalizációs lehetőségek közül pedig az input adat tömörítését választottam kipróbálásra, ezzel csökkentve az általa foglalt helyet, az I/O műveletek számát és a hálózaton *Map* taszkokhoz küldendő adat nagyságát, viszont növeltem a *cpu* időt, amit kitömörítésre is kellett fordítani onnantól. Eredményként azt kaptam, hogy kis adathalmazokra még nem éri meg a tömörítést választani, túl sok időt emészt fel az adat kitömörítése ilyen esetben. Azonban egyre nagyobb adathalmaz esetén (nálam olyan 1-2,5GB-tól) már megérte a tömörítéses megoldást használni. Körülbelül 76%-ára tudtam csökkenteni a futási időt nagy adathalmaz esetén. A munkám tehát összefoglalja a Hadoop rendszerek jellemzőit és alapjául szolgálhat további hasonló kutatási és tervezési feladatok számára.

7. További tervek

A témakörben való további kutatásokban mindenképp terveim között van az, hogy a *map* és *reduce* fázis közötti tömörítési lehetőségeket kipróbáljam, hiszen a mérési eredményeim alapján ott még nagyon sokat lehetne nyerni a teljesítményen. A két fázis közötti teljesítményjavításra egy másik lehetőség a *Combiner* írása, mellyel rendezni lehet az adatokat, mielőtt azok a *reduce* bemenetére kerülnek. Ezt is érdemes lenne kipróbálni. Lehetőség van kódból megadni, hogy hány *reduce* taszkot indítson a rendszer. Ennek segítségével például az arány-számítás algoritmusnál 12 *reduce* taszkot beállítva valószínűleg jelentős javulást lehetne tapasztalni. (Alapból egy *reduce* taszkot indított a rendszer minden futáshoz.) Az első két algoritmusnál érdemes lenne mindig annyira állítani, amennyi évet feldolgoztatok vele, ezáltal nekik is javulna a futási idejük. Egyedül a százalék-számítás algoritmusnál nem tudnék ezzel gyorsítani, hiszen ott összesen egy kulcsot definiáltam a *Map* metódusban. Ezt is szeretném még megpróbálni. Az algoritmusaim során mindig csak egy MapReduce *Job*-ot futtattam. Terveim között van, hogy több *Job*-ot is tudjak futtatni egymás után és ezzel bonyolultabb algoritmusokat is implementálhassak. Ebben a modellben teljesen másképp lehet algoritmusokat megvalósítani, ezeket is meg szeretném ismerni. Ezeken kívül szeretném jobban megismerni a Hadoop-hoz kapcsolódó projektek teljesítményét is (Hive, Pig, Hbase, Spark), hogy kiderítsem, hogy segítségükkel lehet-e vajon optimalizálni az ilyen jellegű algoritmusokon.

Irodalomjegyzék

[1] Amazing facts and figures about the evolution of hard disk drives
http://royal.pingdom.com/2010/02/18/amazing-facts-and-figures-about-the-evolution-of-hard-disk-drives/ (2010.02.18.)

[2] A Timeline of Database History
http://quickbase.intuit.com/articles/timeline-of-database-history

[3] DBMS popularity broken down by database model
http://db-engines.com/en/ranking_categories (2014.10.)

[4] Thomas Zimmermann, Yves Dubois-Pélerin and Patricia Bomme: Object-oriented finite element programming: I. Governing principles (Section 2.)
http://ac.els-cdn.com/004578259290180R/1-s2.0-004578259290180R-main.pdf?_tid=e4c323e65a1211e4907a00000aab0f26&acdnat=1413999972_9f81a187cc8d5a87dd0bb61d00bc993d (1991.01.29.)

[5] JONATHAN ROBIE, DIRK BARTELS: A COMPARISON BETWEEN RELATIONAL AND OBJECT ORIENTED DATABASES FOR OBJECT ORIENTED APPLICATION DEVELOPMENT
http://www.fing.edu.uy/inco/grupos/csi/esp/Cursos/cursos_act/2000/DAP_DisAvDB/documentacion/OO/POET_rel_vs_obj.pdf (2000.)

[6] LIST OF NOSQL DATABASES
http://nosql-database.org/

[7] DB-Engines Ranking
http://db-engines.com/en/ranking (2014.10.)

[8] Rick Cattell: Scalable SQL and NoSQL Data Stores
http://dl.acm.org/citation.cfm?id=1978919 (2010.12.)

[9] Definitions of Big Data
http://www.opentracker.net/article/definitions-big-data

[10] Big Data defined
http://www.sas.com/en_us/insights/big-data/what-is-big-data.html (2013.)

[11] Big Data, for better or worse
http://www.sciencedaily.com/releases/2013/05/130522085217.htm (2013.05.22.)

[12] The ACID Model
http://databases.about.com/od/specificproducts/a/acid.htm (2014.)

[13] Neal Leavitt : Will NoSQL Databases Live Up to Their Promise?
http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=5410700 (2010.02.08.)

[14] Sandy Mappic: Will NoSQL and Big Data Kill the DBA?
http://www.appdynamics.com/blog/java/will-nosql-kill-the-dba/ (2011.05.18.)

[15] MongoDB
http://www.mongodb.org/ (2013.)

[16] Deep Mistry: MongDB vs Hadoop
http://osintegrators.com/sites/default/files/Mongo.Whitepaper.pdf

[17] Dan Vesset: Transforming Massive Data into Better Outcomes
http://www.afceabethesda.org/PDF%20Files/Vesset_Presentation_3282012.pdf
(2012.03.28.)

[18] Nutch
http://nutch.apache.org/ (2014.)

[19] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung: The Google File System
http://dl.acm.org/citation.cfm?id=945450 (2003.12.)

[20] Jeffrey Dean and Sanjay Ghemawat: MapReduce: Simplified Data Processing on Large Clusters
http://dl.acm.org/citation.cfm?id=1327492 (2004.)

[21] Tom White: A Brief History of Hadoop
https://www.inkling.com/read/hadoop-definitive-guide-tom-white-3rd/chapter-1/a-brief-history-of-hadoop

[22] Tom White: Hadoop Usage at Last.fm
https://www.inkling.com/read/hadoop-definitive-guide-tom-white-3rd/chapter-16/hadoop-usage-at-last-fm (2012.)

[23] Derek Gottfrid: The New York Times Archives + Amazon Web Services = TimesMachine
http://open.blogs.nytimes.com/2008/05/21/the-new-york-times-archives-amazon-web-services-timesmachine/?_php=true&_type=blogs&_r=0
(2008.05.21.)

[24] TimesMachine
http://timesmachine.nytimes.com/browser (2014.)

[25] Alex Holmes: Hadoop in Practice
http://dl.acm.org/citation.cfm?id=2543981 (2012.)

[26] HDFS & MapReduce
http://www.cloudera.com/content/cloudera/en/products-and-services/cdh/hdfs-and-mapreduce.html

[27] Mohammad Asif Khan, Zulfiqar A. Memon, Sajid Khan: Highly Available Hadoop NameNode Architecture
http://ieeexplore.ieee.org/xpl/articleDetails.jsp?tp=&arnumber=6516346&url=http%3A%2F%2Fieeexplore.ieee.org%2Fxpls%2Fabs_all.jsp%3Farnumber%3D6516346 (2012.11.26.)

[28] Luis Bautista, Alain April: SUSTAINABILITY OF HADOOP CLUSTERS
http://publicationslist.org/data/a.april/ref271/Sustainability%20of%20Hadoop%20Bautista-April%20(CLOSER-CR).pdf

[29] Peng Hu and Wei Dai: Enhancing Fault Tolerance based on Hadoop Cluster
http://www.sersc.org/journals/IJDTA/vol7_no1/4.pdf (2014.)

[30] Ohnmar Aung, Thandar Thein: Enhancing NameNode Fault Tolerance in Hadoop Distributed File System
http://research.ijcaonline.org/volume87/number12/pxc3894020.pdf (2014.02.)

[31] Jagat Singh: Hadoop Daemons
http://jugnu-life.blogspot.com/2012/04/hadoop-daemons.html (2012.04.23.)

[32] Margaret Rouse: MapReduce
http://searchcloudcomputing.techtarget.com/definition/MapReduce (2010.02.)

[33] Jeffrey Jestes: MapReduce job processing
http://www.cs.utah.edu/~lifeifei/cs6931/MRJob.pdf (2012.04.17.)

[34] Margaret Rouse: Apache Hadoop YARN (Yet Another Resource Negotiator)
http://searchdatamanagement.techtarget.com/definition/Apache-Hadoop-YARN-Yet-Another-Resource-Negotiator (2013.12.)

[35] Apache Hadoop NextGen MapReduce (YARN)
http://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html (2014.09.05.)

[36] Welcome to Apache HBase™
http://hbase.apache.org/ (2014.)

[37] APACHE HIVE TM
https://hive.apache.org/ (2014.)

[38] Welcome to Apache Pig!
http://pig.apache.org/ (2014.07.05.)

[39] Apache Spark™
https://spark.apache.org/ (2014.)

[40] Welcome to Apache ZooKeeper™
http://zookeeper.apache.org/ (2014.)

[41] HDP 2.2 - A major step forward for Enterprise Hadoop.
http://hortonworks.com/hdp/whats-new/ (2014.)

[42] Spring for Apache Hadoop
http://projects.spring.io/spring-hadoop/ (2014.10.)

[43] Kauschik Pal: Introduction to Hadoop Streaming
http://www.devx.com/opensource/introduction-to-hadoop-streaming.html (2014.03.18.)

[44] Mengwei Ding, Long Zheng, Yanchao Lu, Li Li, Song Guo, Minyi Guo: More Convenient More Overhead: The Performance Evaluation of Hadoop Streaming
http://dl.acm.org/citation.cfm?id=2103444 (2011.)

[45] Kauschik Pal: Exploring Various Hadoop Installation Modes
http://www.devx.com/opensource/exploring-various-hadoop-installationmodes.html (2014.04.25.)

[46] Hortonworks Data Platform
http://hortonworks.com/hdp/downloads/ (2014.10.)

[47] QuickStart VMs for CDH 5.1.x
http://www.cloudera.com/content/cloudera/en/downloads/quickstart_vms/cdh-5-1-x.htmlv (2014.)

[48] Derrick Harris: It's everywhere! The day Hadoop took over the cloud
https://gigaom.com/2013/10/28/its-everywhere-the-day-hadoop-took-over-the-cloud/ (2013.10.28.)

[49] Amazon EMR
http://aws.amazon.com/elasticmapreduce/ (2014.)

[50] MapR
https://www.mapr.com/partners/partner/google-compute-engine-and-mapr (2014.)

[51] HDInsight
http://azure.microsoft.com/hu-hu/services/hdinsight/ (2014.)

[52] Finding Data on the Internet
http://www.inside-r.org/howto/finding-data-internet (2011.10.06.)

[53] On-Time Performance
http://www.transtats.bts.gov/Fields.asp?Table_ID=236 (2014.08.)

[54] Get the data
http://stat-computing.org/dataexpo/2009/the-data.html (2014.)

[55] Bi-Annual Data Exposition
http://stat-computing.org/dataexpo/ (2014.)

[56] Airline on-time performance
http://stat-computing.org/dataexpo/2009/ (2014.)

[57] Command line tools
http://stat-computing.org/dataexpo/2009/unix-tools.html (2014.)

[58] Using a database
http://stat-computing.org/dataexpo/2009/sqlite.html (2014.)

[59] SQLite
http://sqlite.org/

[60] ggplot2
http://ggplot2.org/ (2013.)

[61] mySQL
http://www.mysql.com/ (2014.)

[62] SAS
http://www.sas.com/en_us/home.html

[63] Microsoft Azure
https://azure.microsoft.com/hu-hu/ (2014.)

[64] Using Storm on Windows Azure
http://azurecoder.azurewebsites.net/using-storm-on-windows-azure/
(2013.10.25.)

[65] HDinsight Storm overview
http://azure.microsoft.com/en-us/documentation/articles/hdinsight-storm-overview/ (2014.)

[66] Apache Hadoop 2.4.0
http://hadoop.apache.org/docs/r2.4.0/ (2014.03.31.)

[67] Use Azure Blob storage with Hadoop in HDInsight
http://azure.microsoft.com/en-us/documentation/articles/hdinsight-use-blob-storage/ (2014.)

[68] Azure Explorer
http://www.cerebrata.com/products/azure-explorer/introduction (2014.)

[69] Airline On-Time Statistics and Delay Causes
http://www.transtats.bts.gov/OT_Delay/OT_DelayCause1.asp?pn=1

[70] Understanding the Reporting of Causes of Flight Delays and Cancellations
http://www.rita.dot.gov/bts/help/aviation/html/understanding.html

[71] LZO
http://www.oberhumer.com/opensource/lzo/ (2014.06.29.)

[72] Snappy
https://code.google.com/p/snappy/