Budapesti Műszaki és Gazdaságtudományi Egyetem
Villamosmérnöki és Informatikai Kar
Távközlési és Médiainformatikai Tanszék

Alkalmazásfüggetlen Big Data erőforrás elosztás

Készítette
Haja Dávid

Konzulens
Dr. Toka László

2018. október 28.

Tartalomjegyzék

Kivonat

A folyamatosan fejlődő digitális világ, a mobil eszközök és az IoT (Internet of things) technológiák megjelenésével az utóbbi időben robbanásszerűen megnőtt az összegyűjtött adatok mennyisége, amelynek következtében az információ elfogadható idő alatti feldolgozása a hagyományos értelemben vett adatbázis technológiákkal sok alkalmazási esetben szinte lehetetlen. A Big Data fogalma alatt azokat az elosztott rendszereket értjük, amelyek képesek az ilyen nagy mennyiségű, komplex, gyorsan változó, olykor jelentős zajt tartalmazó adatok korlátos időn belül vett feldolgozására. A Big Data a gyors adatfeldolgozás mellett az adatok biztonságos tárolását is feladatául tűzi ki. Ennek érdekében fontos szerepet játszik az egyes technológiák elosztottságon alapuló tulajdonsága.

A különböző elosztott adatfeldolgozó rendszereket, köztük a Big Data technológiákon alapuló rendszereket is általában úgy telepítik, hogy az alkalmazás egy egész adatközpontot igénybe vesz. A virtualizációs technológiák elterjedésével azonban megjelentek a virtuális gépekben vagy konténerekben futtatható szolgáltatások. Tehát az alkalmazások a fizikai környezet mellett már virtualizált infrastruktúrára is telepíthetők. Ezek sok esetben továbbra is központosított hardvereken, pl. szerverparkokban, kerülnek együttesen elhelyezésre. A hálózatok fejlődése és a virtualizált számítási platformot nyújtó technológiák, pl. OpenStack, elterjedése azonban lehetővé teszi a földrajzilag elszórt infrastruktúra nyújtotta előnyök kihasználását Big Data alkalmazások alatt is, pl. a feldolgozó egységek elhelyezését közel adatok keletkezési helyéhez. Erre alkalmas az edge computing architektúra, mely kiterjeszti a hagyományos központosított topológiát azzal, hogy a hálózat szélein virtualizált környezet futtatására alkalmas csomópontokat helyez el. Ezek a hálózat szélén elhelyezett csomópontok kevesebb számítási erőforrással rendelkeznek mint a nagy adatközpontok, azonban hálózati szempontból közelebb találhatóak a felhasználóhoz vagy az adatot szolgáltató entitáshoz.

Tudományos dolgozatomban ismertetem a Big Data technológiák legelterjedtebb komponenseit. Bemutatom a jelenlegi Hadoop ökoszisztémát, majd részletesen kitérek a feladatom által meghatározott rétegre, az erőforrás-vezénylésért felelős megoldásokra. Összevetek különböző erőforrás-vezénylő algoritmusokat, majd a jövőben használatos földrajzilag elosztott topológián felmerülő hálózati problémákra világítok rá, melyekre saját megoldásokat készítek. A megoldások tesztelésére saját szimulátort implementálok, melynek segítségével földrajzilag elosztott topológián szimulációkkal bizonyítom az elkészített algoritmusok helyességét, továbbá ugyanebben a szimulációs környezetben más vezénylő megoldásokkal hasonlítom össze saját megoldásaimat. Végezetül ismertetem az eredményeimet.

Abstract

With the continuous progression of digital world, mobile technologies and the „Internet of Thing", the amount of collected data increased so much that the traditional database technologies cannot process these large data sets for most of the applications in an acceptable timely manner. The term Big Data collects those systems that can handle the processing of these fast changing, various, complex and huge amount of data. In addition to fast data processing, Big Data technologies sets the requirement of storing our datas securely. Regarding this matter, the distributed nature of each technology play an important role.

Different distributed data processing systems, including systems based on Big Data technologies are generally installed in data centers utilizing all of their physical resources. However with the spread of the virtualisation, services installed in virtual machines or containers have appeared, which means applications can run in a virtual environment. Although in many cases, the components are still deployed together on centralized infrastructure, e.g. in data centers. The continuous development of technologies providing a virtualized computing platform, e.g., OpenStack, allows taking the advantage of geographically distributed infrastructure for Big Data applications, e.g., processing data close to where it was generated. There is a relatively new architecture for this purpose called fog computing or edge computing which essentially extends the cloud computing to the edge of the network enabling resource deployment near the clients. These edge nodes own less computational resource than a centralized infrastructure, but from network aspect they are closer to where the data is created.

In this paper I give a brief explanation on the main components of Big Data technologies. I present the current Hadoop ecosystem and I give some details about the resource orchestration layer. I compare different resource orchestration algorithms. I identify three possible problems related to network resources of a geographically distributed topology, which I then solve in this paper. I implement simulation environments where I prove the correctness of the prepared algorithms with simulations in large scale geographically distributed topologies. In these simulations I also compare my algorithm with existing resource orchestration algorithms. Finally, I present my results.

1. fejezet

Bevezető

Ebben a fejezetben bemutatom a Big Data alapjait, valamint a teljesség igénye nélkül leírom a Hadoop ökoszisztémának a rétegeit. Ezt követően ismertetem a dolgozatom motivációját, amely során bepillantást nyújtok egy új hálózati topológiába, a felhő-, és peremszámítási koncepcióba.

1.1. A Big Data rendszerek tulajdonságai

Az informatika fejlődésével tárolt adat mennyisége az utóbbi évek alatt ugrásszerűen megnövekedett. Ennek oka kettős. Az első ok a különböző adatrögzítő, adatmegosztó technológiák rohamos fejlődése, mind hardveres, mind szoftveres értelemben. Számos hardver képes különböző típusú adatokat rögzíteni. Például audovizuális adatok rögzítésére használható nagyon sokféle eszköz, például mobiltelefon, videokamera, fényképezőgép, stb. A rengeteg szenzor elterjedésével lehetővé vált különböző környezeti tulajdonságok digitális rögzítése, mint például hőmérséklet, páratartalom, fényerő és még sorolhatnánk. A különböző szoftveres technológiák fejlődésével megnőtt a virtuális úton létrehozott adatok mennyisége, melyek szintén egyszerűen megoszthatóak bárkivel a világon, ilyen technológiák közé tartozik például az Internet, a közösségi oldalak és egyéb adatfeldolgozó szoftverek.

Az adatokat három különböző osztályba sorolhatjuk:

- *Struktúrált*: például gráfok, n-dimenziós mátrixok, $n \times m$-es táblázatok, ahol egy sor egy megfigyelés, egy oszlop egy változó.
- *Szemi-struktúrált*: Részhalmaza a struktúrált adatoknak. Nem ábrázolható hatékonyan adattáblában. Azonos típusú objektumok más attribútumokkal is előfordulhatnak. Az attribútumok sorrendje nem számít. Ilyen például az XML.
- *Struktúrálatlan*: Nincs előre rögzített tárolási/értelmezési modell, csak metaadat. Például természetes szöveg.

Azonban ezek a rögzített adatok, csak akkor lehetnek hasznosak, ha megfelelően tudjuk tárolni, illetve feldolgozni őket. Erre a problémára nyújt megoldást egy Big Data technológiákat használó rendszer. A Big Data tehát nem csupán egy technológia, sokkal inkább olyan technológiák szintézise, melyek lehetővé teszik egy rendszer számára hatalmas mennyiségű

és változatos adatok tárolását és gyors feldolgozását. A Big Data négy alap tulajdonságát írja le az úgynevezett 4V model, mely a következőket tartalmazza:

- *„Volume”*: nagy mennyiségű adat;
- *„Variety”*: nagyszámú forrás és/vagy struktúrálatlan/részben strukturált adatok;
- *„Velocity”*: gyors adatfeldolgozás;
- *„Veracity”*: nagy mennyiségű zaj.

Az adatok feldolgozásának kétféle típusát különböztetjük meg. Az első az úgynevezett „At rest Big Data”. Ebben az esetben az adatok előre össze vannak gyűjtve és tárolva vannak a Big Data klaszterünkben. Ekkor az elemzett adatok nem frissülnek a futás ideje alatt, és van időnk szinte mindent elemezni. Ennek az ellentéte az úgynevezett „streaming” adatfeldolgozás, amikor az adatok folyamatosan érkeznek és az érkezésük idejében szinte azonnal dolgozzuk fel őket.

A Big Data technológiákat használó rendszerek elosztott rendszerek. Ezekben az elosztott rendszerekben minden egységnek megvan a feladata. Minden szerver/komponens a kapott adaton egy megadott feladatot old meg, és az eredményt küldi tovább egy másik szervernek/komponensnek, aki szintén a saját feladatát fogja megoldani a kapott adatokon. A „bring the computation to the data” elv lényege az, hogy egy adott feladatot elvégző kód/kódrészlet azon a szerveren lesz lefuttatva, ahol az adat éppen van. Az adatok a számítási erőforráshoz való mozgatása túl költséges, így a számítást kell az adathoz szervezni.

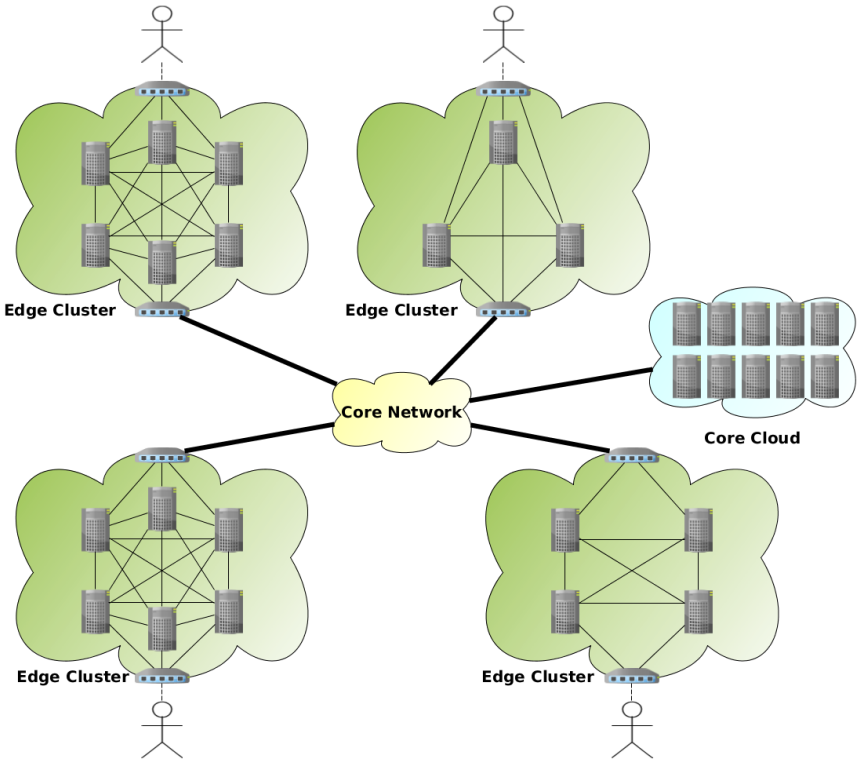
1.2. Elosztott Big Data rendszerek

A virtualizált platformok elterjedésével megjelentek a felhőben futó szolgáltatások. Köztük az egyre nagyobb teret nyerő Big Data szolgáltatások is. A felhő szolgáltatások szerepe első sorban a költség optimalizálás. Felhő platformon akkor éri meg futtatni egy Big Data rendszert/alkalmazást, ha az nem folyamatosan nagy terhelést igényel, inkább periodikusan változik a terheltségi szintje. Ebben az esetben a felhőben bérelt erőforrások mennyisége változtatható a terheltséghez mérten, így költséghatékonyabban lehet üzemeltetni.

A technológiai fejlődés megpróbálja egyre közelebb vinni a feldolgozó egységeket az adatok keletkezési helyéhez. Ez azt jelenti, hogy szükség van a nagy adatközpontok mellett hasonló, azonban kisebb számítási kapacitással rendelkező adatközpontokra, amelyek azonban mind földrajzi, mind hálózati tekintetben közelebb vannak az adat keletkezésének helyéhez. Ezekben az úgynevezett peremklaszterekben nem csak az adatok tárolása, hanem bizonyos feldolgozási feladatok is megjelennek. Az ilyen jellegű alkalmazásokra talán legjobb példa az IoT (Internet of Things) felhasználás. Az IoT rendszerek számos szenzort használnak, melyek folyamatosan monitorozzák a környezetüket és továbbítják az adatokat. Ezeket az adatokat sok esetben nem csak tárolni kell, hanem véges időn belül valamilyen választ is kell nyújtani rájuk. Amennyiben a feldolgozó egység túlságosan távol helyezkedik el a szenzoroktól, úgy nincs esély arra, hogy időben megkapják a szenzorok a válaszokat.

1.3. Az elosztott adatfeldolgozás hálózati kihívásai

A Big Data rendszerek egyik fő alapelve a „scale out instead of scale up” elv. Ez azt jelenti, hogy érdemesebb inkább több különálló fizikai erőforrással egy elosztott rendszert létrehozva kibővíteni az infrastruktúrát, mint csupán egy szerver kapacitásait bővítve terjeszkedni. Ez az elv egyre inkább elterjedté válik egy absztrakciós szinttel magasabban is. Érdemes több adatközpontot létrehozva egy földrajzilag elosztott rendszert alkalmazva skálázni az infrastruktúrát, mint csupán egy adatközpont kapacitásait bővítve skálázni. Azonban a földrajzi elosztottsággal új problémák lépnek fel, melyek jelentős része a hálózattal van összefüggésben. Egy példa földrajzilag elosztott topológia látható az 1.1 ábrán.



1.1. ábra. *Földrajzilag elosztott topológia sávszélesség szerint*

A hálózati topológia késleltetés és rendelkezésre álló sávszélesség szempontjából mutatja az elosztottan működő számítási klaszterek kapcsolatát. Az elképzelés az, hogy földrajzilag egymástól viszonylag távol helyezkednek el kisebb méretű klaszterek (az ábrán „Edge Cluster”-ek), melyek egy központi hálózaton („Core Network”) vannak összekötve. Továbbá a topológiában szerepel egy sok erőforrással rendelkező felhő klaszter (az ábrán „Core Cloud”) is, amely lehet privát vagy publikus szolgáltatás egyaránt. Minden klaszter rendelkezik egy dedikált interfésszel, amely a külső, publikus hálózat felé/felől nyújt átjárást. Számos jelenlegi kutatás folyik annak céljából, hogy belássák, hogy hogyan lehetséges optimálisan használni a földrajzilag elosztott topológiákat Big Data rendszerek használatakor. A dolgozatommal én is ebben a jelenleg folyó kutatási területen mutatok be újszerű eredményeket: megfogalmazok és megoldást nyújtok olyan problémákra, melyek fontosak lehetnek az új topológiák alkalmazása során az elosztott Big Data alkalmazásoknak.

Egy hálózat tervezése, felépítése során a fő elvárások a hálózattal szemben, hogy gyors legyen és megbízható. A gyorsaságot befolyásoló két legjellemzőbb érték a hálózaton található sávszélesség, valamint a késleltetés, ami a cél eszköz elérése során található. Számos kutatás foglalkozik azzal, hogy különböző új topológiákat vizsgálva, hogyan tudják javítani ezeket az értékéket. Egy új topológia megjelenésével a hálózatba kapcsolt alkalmazásoknak új problémákkal kell, hogy szembesüljenek, melyek befolyásolják a teljesítményüket. Munkám során én is ez a három hálózati tulajdonság szerint vizsgálom a Big Data rendszerét a földrajzilag elosztott topológiában, és ajánlok olyan algoritmust, amely a hálózati metrikákat felhasználva javít a gyakorlatban elterjedt alkalmazások teljesítményén.

2. fejezet

A legelterjedtebben használt Big Data technológiák

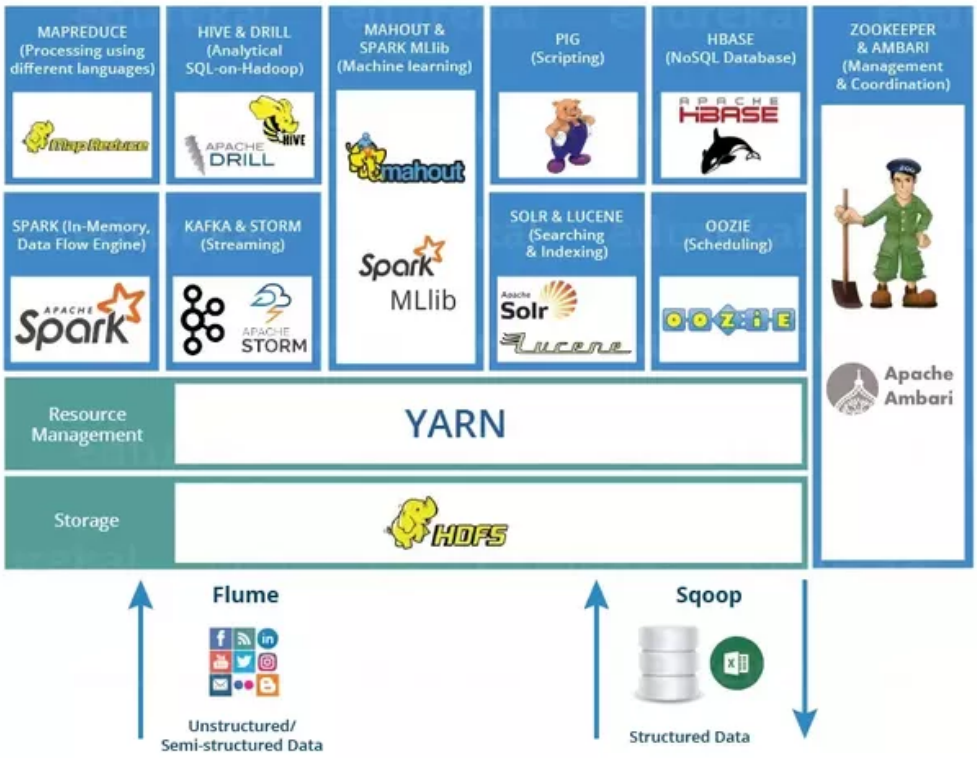
Ebben a fejezetben bemutatom a Big Data legismertebb és legfontosabb techonológiát, valamint az egészet összefogó keretrendszer, a Hadoop felépítését, ökoszisztémáját.

2.1. A Hadoop ökoszisztéma

A Hadoop egy nyílt forráskódú keretrendszer, amelyet az Apache Software Fundation támogat. Adatintenzív, elosztott rendszerben működő alkalmazások támogatására szolgál. Sok szerverből álló, viszonylag alacsony költségű, általános célú hardverből épített klasztereken teszi lehetővé az adatfeldolgozó technológiák futását. Mondhatni, hogy Big Data problémák megoldására szolgáló keretrendszer a Hadoop. Az Apache Hadoop magja egy adattároló részből áll, közismert nevén a Hadoop Distributed File System (HDFS), továbbá egy feldolgozó komponensből, a MapReduce programozási modellből. Az alap Apache Hadoop keretrendszer a következő modulokból áll [1]:

- *Hadoop Common*: más Hadoop modulok által használt könyvtárak és segédprogramokat tartalmaz;
- *Hadoop Distributed File System (HDFS)*: elosztott fájlrendszer, amely adatok tárolására szolgál általános célú gépeken, továbbá nagy sávszélességet biztosít a klaszterben az egyes számítógépek között;
- *Hadoop YARN*: 2012-ben bevezetett platform, amely a klaszterek számítási erőforrásainak vezényléséért felelős, továbbá a felhasználói alkalmazások ütemezését is végzi;
- *Hadoop MapReduce*: a MapReduce programozási modell egy implementációja nagyméretű adatok feldolgozására.

A 2.1. ábrán látható a Hadoop ökoszisztémájából egy részlet [2]. Magához az ökoszisztémához számos más technológia, szoftver is hozzátartozik, azonban a dolgozatomban ezek nem kerülnek bemutatásra.



2.1. ábra. *Hadoop ökoszisztéma*

Vizsgáljuk meg a Hadoop ökoszisztéma egyes szintjeit és a bennük található technológiákat.

- *Adatbevitel*: Ezen a szinten találhatóak azok a technológiák, melyek segítségével adatokat, fájlokat tudunk egyszerűen a Big Data klaszterbe (legtöbbször HDFS-be) juttatni. A legelterjedtebb technológiáak a következőek:
  - *Flume*: A Flume egy olyan szolgáltatás, amely segíti a strukturált és szemistrukturált adatok HDFS-be való bevitelét. Olyan megoldást kínál, amely megbízható, elosztott működésű, továbbá segít nagy mennyiségű adathalmazok készítésében, aggregálásában és mozgatásában. Segítségével online adatfolyamokat tudunk HDFS-be juttatni olyan különböző forrásokból mint például hálózati forgalom monitorozó, napló fájlok, email üzenetek.
  - *Sqoop*: A Sqoop importálhat és exportálhat strukturált adatokat RDBMS (Relational Database Management System) vagy vállalati adattárházakból a HDFS-be vagy fordítva.
- *Adattárolás*: A Hadoop-ban az adatok-fájlok tárolásáért felelős fájlrendszer a HDFS [3]. A HDFS-ről bővebben a 2.2 alfejezetben található leírás.
- *Erőforrás-vezénylés*: A Hadoop keretrendszerben futtatott alkalmazások erőforrásainak a vezényléséért felelős legjelentősebb technológiák a YARN és a Mesos. Az erőforrás-vezénylésről bővebben a 2.4 alfejezetben található leírás.

- *Adatfeldolgozás*: A Big Data klaszter adatainak feldolgozását lehetővé tevő legelterjedtebb technológiák a MapReduce és a Spark. Erről a két technológiáról bővebben a 2.3 alfejeztben található leírás.

- *Alkalmazások*: Ezen a szinten találhatóak azok a technológiák, melyek az eddig ismertetett szintekre építenek, használják a bennük található megoldásokat és esetlegesen ki is egészítik a funkcionalitásukat.

  - *Hive*: A Hive egy adattárház komponens, amely nagy és elosztott adathalmazok olvasását, írását kezeli SQL-szerű interfészt nyújtva a felhasználónak.
  - *Pig*: Az Apache Pig nagy adatkészletek elemzésére szolgáló platform, amely magas-szintű programozási nyelvet nyújt az adatelemző programok implementálására, valamint az ezeknek a programoknak az értékelésére is nyújt szolgáltatást.
  - *Oozie*: Az Oozie egy munkafolyamat menedzselő rendszer Apache Hadoop munkák ütemezésére.
  - *Solr*: Az Apache Solr egy megbízható, jól skálázódó, elosztott indexelést és kereső szolgáltatást nyújtó technológia.
  - *HBase*: Az HBase egy NoSQL elosztott adatbázis a HDFS felett, amely képes kezelni mindenféle adatot a Hadoop klaszteren belül.

- *Menedzsment és koordináció*: Olyan szolgáltatások tartoznak ebbe a csoportba, melyek a többi Hadoop technológiát menedzselik és támogátják a működésükben.

  - *ZooKeeper*: A ZooKeeper egy központosított szolgáltatás, mely konfigurációs információk menedzselését, szinkronizálását biztosítja és a többi Hadoop komponens csoportos szolgáltatásainak helyes működését biztosítja.

2.2. Adattárolás

Ebben az alfejezetben ismertetem a Big Data alap szolgáltatását, amely az adattárolásért felelős, a HDFS-t.

Amikor egy adathalmaz kinövi az egy fizikai gép által nyújtható maximális tárhely kapacitást, elkerülhetetlen, hogy az adatot több partícióra bontsuk szét, melyeket aztán több szeparált gépre helyezünk el. Az olyan fájlrendszereket, melyek több gépből álló hálózatba kapcsolt tárhelyet menedzselnek, elosztott fájlrendszernek nevezzük. A hálózatba kapcsolt elosztott tárhely megvalósításának egyik legnagyobb kihívása azt elérni, hogy a fájlrendszer toleráns legyen csomópontok különböző meghibásodására adatvesztés nélkül.

A Hadoop ökoszisztéma által használt elosztott fájlrendszer a HDFS. A HDFS kifejezés a „Hadoop Distributed Filesystem” rövidítése. A HDFS olyan fájlrendszer, amely akár nagy méretű fájlokat is tud tárolni, folyamszerűen lehet benne a fájlokat elérni és közönséges hardverből összeállított klaszteren is használható. Vizsgáljuk meg mit jelentenek részletesebben ezek a tulajdonságok.

- Nagyméretű fájlok tárolása: A HDFS kihasználtsága annál jobb, minél kevesebb számú, de nagyobb méretű fájlokat tárolunk benne. Ez a tulajdonság abból fakad, hogy az egyes fájlokra különböző metaadatokat kell tárolnunk, melyek plusz helyet igényelnek. Így a kevesebb számú, de nagyobb méretű fájl jobban kihasználja a HDFS által nyújtott lehetőségeket.

- Folyamszerű fájlelérés: A HDFS-ben a fájlokra jellemző, hogy egyszer írjuk és utána többször olvassuk őket. A tárolt fájlok módosítása csak a fájl végén történő bővítéssel lehetséges.

- Közönséges hardver: A HDFS nem igényel speciális eszközöket. Általános szervereken is képes futni.

A HDFS a hagyományos tárhelyekhez hasonlóan szintén blokkokat használ a fájlok tárolására. Azonban HDFS esetében egy blokk mérete sokkal nagyobb (alapértelmezetten 128 MB), mint a hagyományos esetben. Egy tradicionális fájlrendszerhez és az általa kezelt merevlemezhez hasonlóan, a HDFS fájlok blokk-méretű darabokra vannak tördelve, melyeket független egységek tárolnak az elosztott rendszerben. Azonban HDFS-ben egy fájl, ami kisebb mint a beállított blokk méret, megtartja a méretét és nem foglal el egy blokknyi területet. Például ha 1 MB méretű fájlt tárolunk 128 MB beállított blokk mérettel, akkor is csak 1 MB területet foglal el a fájl a fájlrendszerben.

A fájlok absztrahálása blokkokra számos előnyt hoz be egy elosztott fájlrendszerbe. Először is egy fájl nagyobb méretű lehet, mint bármelyik egyedülálló merevlemez a hálózatban. Egy másik szintén jelentős előny, hogy a blokk használata jól illeszkedik a replikáció által nyújtotta hibavédelmet és elérhetőséget kínáló funkcionalitásba. A blokk, lemez, gép meghibásodásának kezelése érdekében minden blokkot több (alap beállítás szerint három) fizikailag különálló gépre replikálnak.

A HDFS-nek két fő komponense van: a NameNode (név hoszt), DataNode (adat hoszt). Ez a két komponens mester-dolgozó viszonyban van egymással. A NameNode felelős a fájlrendszer névterének a menedzseléséért. Karbantartja a fájlokhoz és könyvtárakhoz tartozó metaadatokat. Az ilyen információk a NameNode merevlemezén vannak eltárolva kettő fájlban: a névtér képfájlban és a szerkesztés naplófájlban. A NameNode tudja, hogy melyik adat hoszton melyik fájlok mely blokkjai találhatóak. Azonban a blokkok helyeit nem tárolja perzisztensen, mert ezt az információt újraépíti a DataNode-ok segítségével a rendszer elindulásakor. Egy kliens úgy fér hozzá a fájlrendszerhez, hogy kommunikál a név hoszttal és a megfelelő adat hosztokkal.

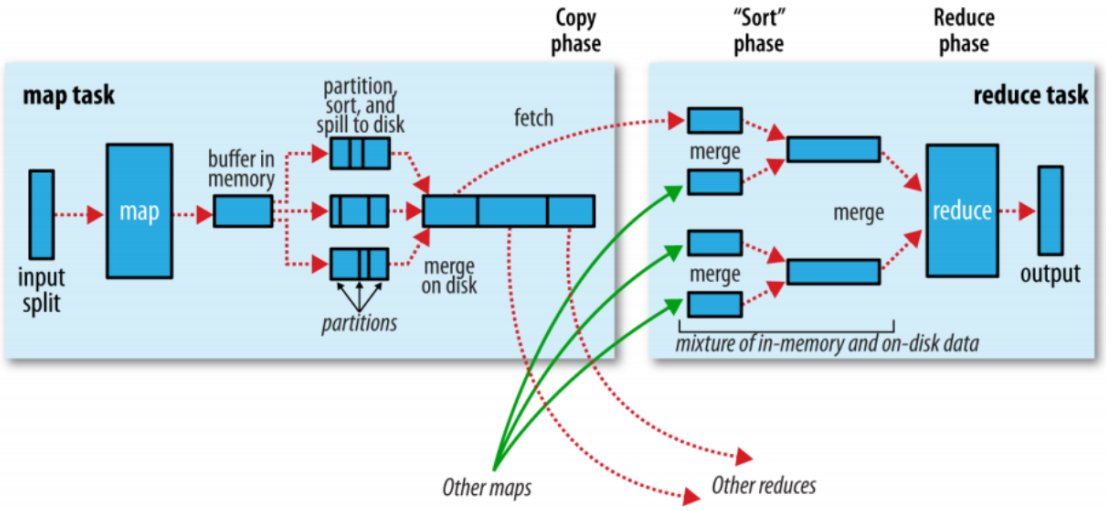
Az adat hosztok a fő dolgozók a rendszerben. Ők tárolják fizikailag a blokkokat, valamint rajtuk keresztül érik el a kliensek a szükséges adatokat. Fontos feladatuk továbbá, hogy az általuk tárolt összes blokkot periodikusan jelentsék a név hosztnak. A név hoszt nélkül az egész fájlrendszer használhatatlan. Nélküle elveszne az adatbázis, amelyben megtalálható az összes információ a fájlok tárolását és elérhetőségét illetően.

2.3. Adatfeldolgozás

Ebben az alfejezetben az olyan technológiákat mutatom be, melyek az adatok feldolgozásáért, transzformálásáért felelősek. Ezek a technológiák az előzőleg látott adattárolási szint felett helyezkednek el.

2.3.1. A Hadoop MapReduce keretrendszere

A Hadoop MapReduce egy szoftver keretrendszer olyan alkalmazások írására, amelyek nagy mennyiségű adatot (akár több terabájtos adatkészleteket) képesek párhuzamosan feldolgozni általános célú hardvereket nagy számban, akár ezer csomópontot tartalmazó klaszteren megbízhatóan és hibatűrő módon. A MapReduce úgy működik, hogy a feldolgozásnak két fázisa van: az úgynevezett *map* fázis és a *reduce* fázis. Mindegyik fázis bemenete és kimenete kulcs-érték párosokból áll, melyeket az alkalmazás programozója határoz meg. A programozó két funkciót határoz meg az alkalmazásnak: a *map* funkciót és a *reduce* funkciót. A MapReduce keretrendszer felelős a feladatok ütemezéséért, a monitorozásukért és a hibás futások újraindításáért. A 2.2. ábrán látható az általános felépítése egy MapReduce alkalmazásnak [4].



2.2. ábra. *MapReduce működése*

A MapReduce munkák/alkalmazások általában szétválasztják a bemeneti adatkészletet olyan független darabokra, amelyeket a *map* funkciók teljesen párhuzamosan fel tudnak dolgozni. Megfigyelhető a 2.2. ábra bal oldalán egy *map* funkció működésének folyamata. Egy *map* funkció egy bemeneti részletet kap, ami általában HDFS által tárolt adatrészlet (többnyire egy blokk). Egy *map* funkció az adott blokk tartalmát olvassa be és generál belőle kulcs-érték párokat, melyeket majd továbbít a következő (*reduce*) funkciónak. A köztes adatokat funkciók kiírják a fájlrendszerbe. Mielőtt azonban a *map* funkció a lemezre írná a kimenetét, az elkészített kulcs-érték párokat partíciókra bontja. Az adatok aszerint kerülnek az egyes partíciókba, hogy melyik *reduce* funkció példánynak kerülnek továbbküldésre. Minden egyes partíción belül egy háttérszál végrehajt a memóriában lévő adatokon egy

kulcs szerinti sorbarendezést. Amennyiben implementálva van kombináló funkció is, akkor az is lefut a sorbarendező algoritmus kimenetén. A kombináló funkció futtatása kompaktabbá teszi a *map* funkció kimenetét, így kevesebb adatot kell a merevlemezre írni és tovább küldeni a *reduce* funkció példányoknak.

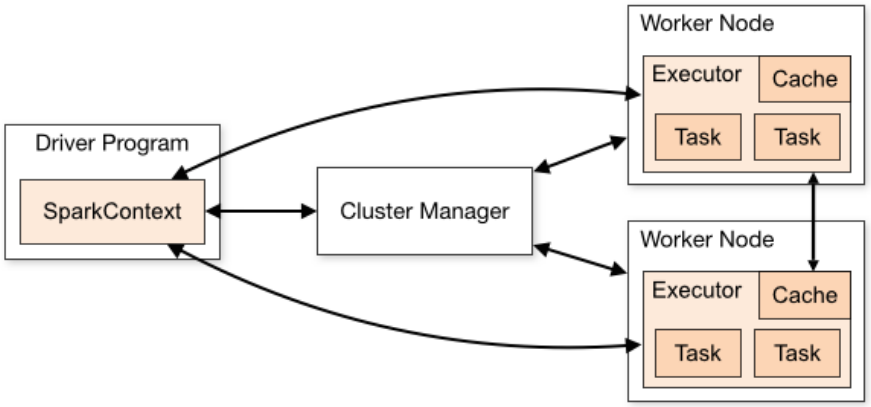
A *reduce* funkció példányai először összegyűjtik a *map* funkciók által adott kimeneteket. Pontosabban a kimenet azon partícióit, melyeket az egyes példányoknak kell feldolgozni. Alap esetben a *reduce* funkciók csak akkor kezdik el összegyűjteni az adatokat, amikor az összes *map* funkció példány befejezte a futását. Általánosságban a *reduce* feladatnak van néhány másoló szálja, melyek párhuzamosan összegyűjtik a *map* funckiók által generált adatokat. Honnan tudják az egyes *reduce* példányok, hogy hol helyezkednek el a számukra fontos adatrészletek? A *map* funkciók miután befejezeték a futásukat, értesítik az alkalmazás mestert. Az alkalmazás mester tudni fogja, hogy az egyes adatrészletek hol találhatóak. A *reduce* példányok indulásuk után addig kérdezik az alkalmazás mestert az adatok helyéről, amíg meg nem kapják az összes szükséges adatrészletet.

Miután egy *reduce* példány összegyűjtötte az összes szükséges adatrészletet, a összefésülő fázisba lép, amely összevonja a *map* által adott kimeneteket, megtartva annak a rendezési sorrendjét. Az összevont adathalmaz lesz az utolsó fázis (a *reduce* fázis) bemenete. A *reduce* fázis alatt a program végigmegy minden kulcson és a hozzá tartozó értékeken, és végrehajtja a programozó által megszabott műveleteket, transzformációkat. A *reduce* fázis kimenete általában a HDFS-re lesz lementve. HDFS esetén, amennyiben a dolgozó hoszt egy adat hoszt is egyben, úgy a kimenet blokkjainak első replikája a példányt futtató hoszton lesz eltárolva.

2.3.2. Spark

A Spark egy nyílt forráskódú szoftver, melynek segítségével Big Data rendszerekben tudunk adatokat feldolgozó alkalmazásokat készíteni. A Spark alkalmazásokat kettő különböző módon használhatjuk a klaszterünkben: kliens módban és klaszter módban. Ebben az alfejezetben bemutatom a Spark fontosabb megvalósításait, melyek a Big Data klaszterben futnak.

A Spark alkalmazások független processz halmazokként futnak egy klaszterben, melyet a Spark kontextus (SparkContext) nevű komponens koordinál. A SparkContext komponens a felhasználó által írt főprogramban helyezkedik el. A 2.3. ábrán látható a Spark architektúrája [5]. Az ábrán a főprogram a bal oldalon található „Driver Program” néven. A klaszter módban történő futtatás esetén a SparkContext hozzákapcsolódik egy klaszter menedzserhez (a 2.3. ábrán Cluster Manager), aminek a feladata az erőforrások allokálása az alkalmazások számára.



2.3. ábra. *Spark architektúrája*

A klasztermenedzser négy fajta lehet: önálló, Mesos, YARN, Kubernetes. Amikor sikeresen kiépült a kapcsolat, a menedzser végrehajtókat szerez a futtatandó alkalmazás számára, azokon a hosztokon, melyek rendelkeznek számítási erőforással és releváns adatokat tárolnak az alkalmazás számára. A következő lépésben elküldi az alkalmazás kódját (JAR vagy Python fájlokban implementálva a SparkContext-ben) a végrehajtóknak. Végezetül a SparkContext feladatokat küld a dolgozóknak, amit azok végrehajtanak.

Vizsgáljunk meg néhány hasznos tulajdonságát az architektúrának.

1. Minden alkalmazás saját végrehajtó processzeket kap, amelyek csak az alkalmazás végeztével törlődnek és a feladatokat több szálon futtatják. Ez a felépítés előnyös az alkalmazások egymástól való szeparálására mind ütemező, mind végrehajtó tekintetében. Azonban ez azt is jelenti, hogy az alkalmazások nem tudják az adatokat megosztani egymás között anélkül, hogy kiírnák az adatokat egy háttértárra.

2. Amíg a Spark képes végrehajtó egységeket szerezni, amik kommunikálnak is egymással, addig a Spark alkalmazások relatíve egyszerűen futtathatóak olyan klaszter menedzserekkel is, melyek más típusú alkalmazásokat is támogatnak.

3. A főprogramnak figyelnie kell és el kell fogadnia a beérkező kapcsolódási kérelmeket a végrehajtók felől a futása során. Tehát a főprogramnak hálózaton keresztül elérhetőnek kell lennie a dolgozó hosztok felől.

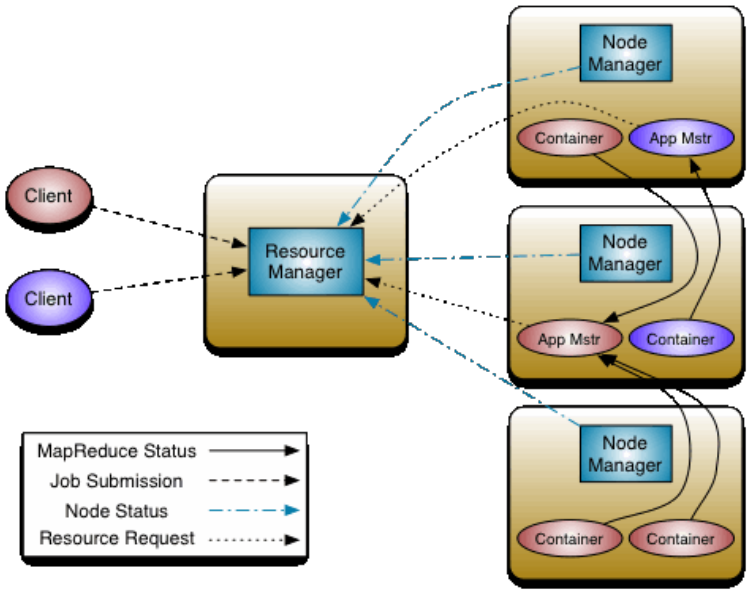
4. Mivel a főprogram ütemezi a feladatokat az alkalmazásban, így annak közel kell lennie a dolgozó hosztokhoz. Ha távolról akar egy felhasználó kéréseket küldeni egy klaszternek, érdemesebb RPC-ket (Remote Procedure Call) küldeni a főprogramnak (ami a klaszterben fut), mint a főprogramot a klaszteren kívülre helyezni.

2.4. Erőforrás-vezénylés

Ebben az alfejezetben bemutatom a Hadoop ökoszisztéma legelterjedtebb erőforrás-vezénylő technológiáját, a YARN-t.

2.4.1. YARN

A YARN a Hadoop ökoszisztéma egyik legelterjedtebb erőforrás-vezénylő technológiája. Segítségével fizikai erőforrásokat tudunk foglalni a Big Data alkalmazásainknak a rendelkezésre álló számítási infrastruktúrán. Az alapvető ötlet a YARN mögött az, hogy ketté bontja az erőforrás-menedzsment és a munkaütemezés, -monitorozás funkcionalitását két egymástól szeparált démonná. Megvalósításban az ötlet azt jelenti, hogy legyen egy globális erőforrás-menedzser és alkalmazásonként egy alkalmazásmester. Egy alkalmazás állhat egy munkából vagy munkák által alkotott körmentes gráfból. A YARN architektúrája a 2.4. ábrán látható [6].



2.4. ábra. *A YARN architektúrája*

A YARN egyes komponensei és azok szerepei a következőek:

- Resource Manager (erőforrás-menedzser): A központi erőforrás-vezénylő logikát tartalmazza. Számon tartja az egyes dolgozó hosztok elérhetőségét és szabad erőforrásuk mennyiségét, valamint az alkalmazásoknak kijelöli, hogy mely hoszton/hosztokon legyen elhelyezve az alkalmazásmester és a többi konténer.
- Node Manager (hoszt menedzser): A hoszt menedzser a szervereken található ügynök, ami felelős a konténerekért, az általuk használt erőforrások monitorozásáért. Továbbá az ügynök feladata jelenteni a monitorozott értékeket az erőforrás-menedzsernek.
- Application Master (alkalmazásmester): Az alkalmazásmester feladata az általa kezelt alkalmazásra erőforrást egyeztetni az erőforrás-menedzserrel, továbbá a hoszt menedzserekkel együttműködve felügyelni az alkalmazás futását, és az alkalmazáshoz tartozó konténerek állapotát.
- Container (konténer): Ezek a konténerek futtatják az alkalmazásban található feladatok kódját, valamint virtualizált környezetet nyújtanak a feladatoknak.

Az erőforrás-menedzsernek kettő fő komponense van: az ütemező (Scheduler) és az alkalmazás-menedzser (ApplicationsManager). Az ütemező felelős az erőforrások hozzárendeléséért a különböző futó alkalmazásokhoz figyelembe véve a korlátokat és sorokat. Az ütemező kizárólag csak ütemezői funkciókat lát el. Ez azt jelenti, hogy nem nyújt semmiféle monitorozással vagy állapotkövetéssel kapcsolatos szolgáltatást az alkalmazások irányába. Továbbá nem biztosít garanciát hibásan futó alkalmazások újraindítása érdekében. Az ütemező az ütemezési funkcióját az alkalmazás által meghatározott erőforrás-követelmények alapján hajtja végre. Az ütemező rendelkezik egy irányelvvel, ami a klaszter erőforrásainak felosztásáért felelős a különböző sorok és alkalmazások között. Ez alapján az irányelv jelenleg kettő féle ütemezőt különböztetünk meg: a kapacitás ütemezőt (CapacityScheduler) és a fair ütemezőt (FairScheduler).

Az alkalmazás-menedzser felelős a kliensek által beküldött alkalmazások elfogadásáért, feldolgozásáért, és az ő felelőssége letárgyalni az első konténer (az alkalmazás mester) erőforrásainak a lefoglalását és indítását. Amennyiben az alkalmazásmester futás közben meghibásodik, az alkalmazás-menedzser feladta újraindítani azt.

A YARN támogatja az erőforrás-foglalás fogalmát a foglalási rendszer (ReservationSystem) segítségével, amely lehetővé teszi a felhasználók számára, hogy meghatározzák az erőforrások túlterhelési idejét és időbeli korlátozásait (pl. határidőket). Továbbá lehetőség van arra, hogy tartalékeszközöket biztosítsanak a fontos feladatok kiszámítható végrehajtása érdekében.

2.5. Kapcsolódó irodalom

Ebben az alfejezetben ismertetem azokat a kutatási eredményeket, melyek relevánsak az általam feldolgozott témában: földrajzilag kiterjedt infrastruktúrán futtatandó elosztott Big Data alkalmazások hálózati kihívásai, és az azokra javasolt megoldások.

Hálózati megbízhatósággal kapcsolatos irodalom

A hálózati megbízhatóság klasszikus téma a hálózatokkal foglalkozó kutatásokban. A mai napig megannyi kutatás foglalkozik ezzel a területtel [7, 8, 9, 10]. Legjobb tudomásunk szerint azonban a HDFS blokkelhelyezési megoldásának eredményét nem vizsgálta még senki a hálózati megbízhatóság szempontjából.

Virtuális funkciók elhelyezésével foglalkozó irodalom

Általánosságban véve a Big Data alkalmazások gyorsaságában jelentkező hálózati késleltetés megfogalmazható abban a formában, hogy virtuális csomópontokat, funkciókat helyezünk el fizikai topológián úgy, hogy figyelembe vesszük a hálózati erőforrásokat, mint például a késleltetést. Ez a terület nem új a hálózati kutatásokban. A VNE (Virtual Network Embedding) [11] egy általános megfogalmazása az ilyen jellegű problémának, amelynek vizsgálatából már számos innovatív eredmény született [12, 13, 14, 15]. A sok eredmény mellett úgy látom, hogy a Big Data rendszerekben az egyes végrehajtók, feladatok elhelyezését még nem vizsgálták hálózati késleltetés szempontjából. Dolgozatom során ezt a hiányosságot is pótolni igyekszem.

A MapReduce hibrid felhőben futtatásával foglalkozó publikációk

A MapReduce alkalmazások felhőben vagy hibrid felhőben futtatása lehetséges és számos előnnyel járhat. Azonban nehéz meghatározni a megfelelő erőforrás mennyiséget és a lefoglalandó erőforrások optimális helyét az ilyen elosztott rendszerekben. A [16, 17, 18, 19] cikkek a MapReduce keretrendszer által végzett adatfeldolgozás teljesítményében lehetséges javításokat tárgyalják elosztott, több adatközpontból álló környezetben.

A [16] szerzőinek a célja MapReduce alkalmazásoknak az erőforrásfoglalása hibrid felhőben két felhasználói metrikát figyelembe véve: a költséget és a munkavégzés idejét. Ezt a problémát formalizálják ILP-ben (Integer Linear Programing), és egy ILP megoldó segítségével gyorsan, az optimálishoz közelítő megoldásokat keresnek. Ehhez a munkához képest a dolgozatomban ajánlott algoritmusok nem veszik számításba az egyes feladatok futási idejét és a hálózati erőforrások pénzbeli költségét.

A [17] diploma munkában a szerző egy olyan ütemezőt fejlesztett, mely képes MapReduce feladatok ütemezésére földrajzilag elosztott klaszterek között. Az implementációjuk célja továbbá a MapReduce alkalmazások futási idejét csökkenteni az új topológián. A szerző megoldásában egy új vezénylő réteget tervezett és implementált, amin keresztül menedzseli az infrastruktúra erőforrásait és az alkalmazás komponenseinek ütemezését.

A MapReduce bizonyítottan hatékonyan működik az adatintenzív alkalmazások széles körében, azonban úgy lett tervezve, hogy egy adatközpontból álló homogén klaszteren működjön. A [18] cikk szerzői azt vizsgálják, hogyan lehet MapReduce alkalmazásokat futtatni földrajzilag elosztottan elhelyezett adatokon, földrajzilag elosztott számítási erőforrások felett. A cikkben a szerzők a MapReduce fázisait külön ütemezik, ezzel szemben az én munkám során a feladatok erőforrásait egyszerre foglalom le.

A [19] cikk először összehasonlítja az egy klaszteres környezetben működő MapReduce adatelemzést a földrajzilag elosztott több klaszteres verzióval, majd három problémát optimalizál: 1) rész-klaszter tudatos feladatütemezés, 2) rész-klaszter tudatos *reduce* kimenetelhelyezés, és 3) *map* bemenet előtöltés. A cikk szerzői több adatközpont egységét tartják egy klaszternek, amelyen belül a rész-klaszter tudatos feladatütemezés során eldöntik, hogy melyik adatközpontban futtatható a feladat hatékonyan. A rész-klaszter tudatos *reduce* kimenetelhelyezés során a *reduce* feladatok kimenetének a tárolási adatközpontját határozzák meg a szerzők. A *map* bemenet előtöltés futásakor a távoli adatközpontokban található bemenetet tölti át a program az adott adatközpontba még a *map* funkciók indítása előtt.

Big Data alkalmazások ütemezése SDN-képes adatközpontokban

A [20, 21] cikkek SDN (Software-Defined Network) hálózatokban vizsgálják meg a Big Data rendszerek működését. A [20] cikkben a szerzők optikai switch-ekkel és SDN kontrollerrel konfigurálják a hálózatot futási időben, és így növelik a Big Data alkalmazás alatti hálózati teljesítményt, valamint a kihasználtságot. OpenFlow szabályokat használnak a hálózat vezénylésére. Az általuk használt infrastruktúrában továbbá Hadoop MapReduce munkákat futtatnak és ezeken az alkalmazásokon keresztül figyelik meg a hálózat működését. Eredményükként megmutatták, hogy az SDN technológiák használatával jobb hálózati kihasználtság érhető el.

A gyakorlatban használt Hadoop rendszerekben felmerül egy probléma, amely jelentősen

befolyásolja a rendszer teljesítményét. Ez a probléma NP-teljes és a lényege, hogy szeretnénk feladatokat rendelni a dolgozókhoz úgy, hogy a munka a lehető legrövidebb időn belül befejeződhessen. Erre az egyik jelenleg legelterjedtebb Big Data rendszerekben alkalmazott megoldás az adat lokalizáció. A jelenlegi megoldások azonban nem veszik figyelembe a rendelkezésre álló hálózati sávszélességet az ütemezés során. A [21] cikk szerzői egy hálózati sávszélességet figyelembe vevő heurisztikus algoritmust nyújtanak, amely ötvözi a Hadoop és az SDN technológiákat. Eredményeikként bemutatják, hogy a saját algoritmusukkal valóban csökkenteni tudták a munkafolyamat futási idejét. A [20, 21] cikkek ugyan a hálózati kihasználtság javítására törekednek, lényeges eltérés azonban, hogy a saját munkám során klasszikus hálózatokat modellezek, amelyek működésébe a felhasználóknak nincs módosítási lehetőségük.

Földrajzilag elosztott topológiában működő Spark alkalmazások ütemezésével foglalkozó irodalom

A [22, 23, 24, 25, 26] cikkek szintén a földrajzilag elosztott több klaszteres környezetben működő adatfeldolgozást vizsgálják, azonban az előzőektől eltérően a vizsgálataik és megoldásaik Spark-ra irányulnak. A [22] cikk szerzői egy új rendszert implementáltak (Iridium), amely alacsony késleltetést biztosít földrajzilag elosztott topológián működő adatanalízist végző alkalmazásoknak azáltal, hogy optimalizálták az adatok elhelyezésére és a lekérdező logikák megvalósítására szolgáló algoritmusokat. Az algoritmusuk egy online heurisztikus algoritmus, amely újraosztja az adatokat a klaszterek között a lekérdezések prioritásai alapján, ezáltal csökkentik a hálózati korlátokat az alkalmazás futása során.

A [23, 24] cikkekben az algoritmusok hálózattudatosan működnek földrajzilag elosztott topológián Spark felett. A két algoritmus célja az alkalmazás futási idejének csökkentése a funkciók között mérhető sávszélesség figyelembe vételével. Az én algoritmusom általánosságban próbálja elérni az optimális sávszélességet a feladatok között, míg a [23, 24] cikkekben bemutatott algoritmusok a költség kiszámításánál figyelembe veszik a köztes adatok méretét.

A [25] cikkben egy új adatelemző platformot hoztak létre, melyet GeeLytics-nek neveztek el. A GeeLytics dinamikus folyamfeldolgozás támogatására készült úgy, hogy figyelembe vegye a topológia heterogenitását (perem klaszterek és felhő jelenlétét) és az aktuális terheltséget. A cikk szerzői szintén az alkalmazások válaszidejét csökkentik úgy, hogy közben minimalizálják a perem és a felhő közti sávszélesség felhasználását. Az általam elkészített modellben nem csupán egy felhő szerepel, hanem több nagy adatközpont is lehet a topológiában, továbbá a sávszélesség kihasználtságot az én algoritmusom az összes klaszter között minimalizálja.

A [26] cikkben bemutatott probléma szintén Spark felett működik, azonban a szerzők a pénzben kifejezett hálózati költségeket csökkentik. Az én munkám során nem foglalkozok a hálózati erőforrások pénzbeli költségével.

3. fejezet

A HDFS blokkok okos elhelyezése a hálózati hibák okozta kiesések csökkentéséért

Ebben a fejezetben részletesen ismertetem a HDFS hálózati beállításait és ismereteit. Bemutatom a működését egy földrajzilag elosztott, több adatközpontból álló topológia esetén és ismertetem a jelenlegi megoldás limitációit. A limitációk bemutatása során különös figyelmet szentelek a hálózattal, hálózati erőforrásokkal kapcsolatos korlátokra. A bemutatott korlátok alapján megfogalmazom és formalizálom a szolgáltatáskiesést okozó problémákat, melyekre a dolgozatomban megoldást keresek. Ebből a célból egy új algoritmust készítek és írok le, amivel kiküszöbölöm a korábban említett limitációkat egy általános földrajzilag elosztott topológián. Az algoritmusom célja a HDFS-ben a blokkok elhelyezése az adatközpontok között úgy, hogy az a legnagyobb megbízhatóságot nyújtsa. Ezek után bemutatom az implementációmat, mellyel a gyakorlatban is használhatóvá teszem az algoritmusomat, majd különböző tesztekkel bizonyítom a működésének helyességét. Összevetem a saját megoldásomat egy másik már létező implementációval, majd bemutatom az eredményeimet, melyek bizonyítják, hogy a javasolt blokkelhelyező algoritmusom magasabb megbízhatóságot nyújt a Big Data alkalmazásoknak.

3.1. A Hadoop és a hálózat

Ahhoz hogy a legjobb teljesítményt tudjuk elérni a Hadoop ökoszisztémában fontos, hogy a Hadoop főbb komponensei ismerjék a hálózat felépítését, topológiáját. Egy rack általánosságban körülbelül harminc-negyven szervert tartalmaz magában, melyek egymáshoz fizikailag közel vannak elhelyezve és egy közös switch-hez csatlakoznak. Több rack-ből álló topológiák esetén az egyes szervereket rack-ekhez kell hozzárendelni, hogy a HDFS képes legyen intelligensebb logikát használva elhelyezni az egyes blokkokat figyelembe véve a teljesítményt és a megbízhatóságot.

A Hadoop által használt alap hálózati csomópontok (például szerver, rack) reprezentálása egy fával történik, amely tartalmazza a hálózati távolságot az egyes csomópontok között.

A NameNode ezeket a csomópontokat felhasználva határozza meg az egyes replikák helyét, továbbá például a MapReduce ütemező ugyanezeket a hálózati csomópontokat használja ahhoz, hogy meghatározza hol van a legközelebbi bemenet az egyes *map* feladatok számára.

Mit jelent az, hogy két csomópont közel van egymáshoz egy hálózatban? Ha a nagy adatmennyiség feldolgozását vesszük célnak, akkor egy jelentős korlátozó tényező az a sebesség, amellyel az adatokat csomópontok között továbbíthatjuk. Az alapötlet ezért nem más, mint hogy használjuk a sávszélességet a csomópontok között a távolság mérésére. Azonban a csomópontok közötti sávszélesség mérése helyett, ami a gyakorlatban nehéz lehet, a Hadoop olyan egyszerű megközelítést alkalmaz, amelyben a hálózatot faként ábrázolja és két csomópont közötti távolság a legközelebbi közös ősnek a távolsága. A fa szintjei nincsenek előre definiálva, de leggyakrabban az egyes szintek megfelelnek egy adatközpontnak, rack-nek, szervernek, amelyen az adatokat feldolgozó folyamatok futnak. Az ötlet az, hogy a következő adatfeldolgozási szcenáriókhoz rendelkezésre álló sávszélesség csökkenő sorrendben a következő:

- Adatok feldolgozása ugyanazon a szerveren
- Egy rack-ben található különböző szervereken adatok feldolgozása
- Feldolgozás különböző rack-ekben található szervereken
- Adatok feldolgozása eltérő adatközpontokban található szervereken

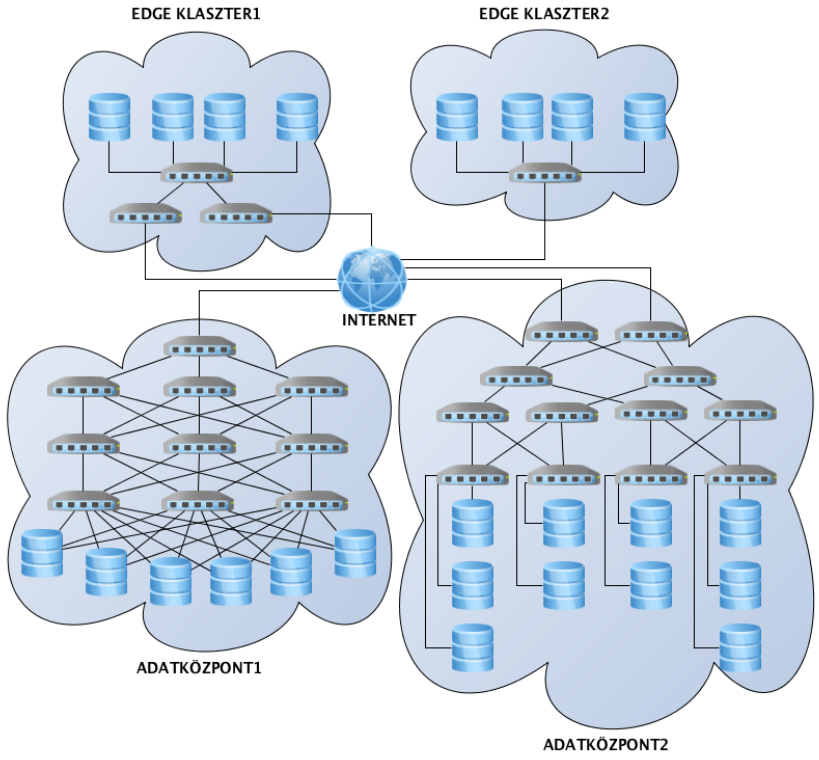
Például legyen egy szerver n1 az r1 rack-ben, ami a d1 adatközpontban található. Ezt a következőképpen reprezentálja a Hadoop: /d1/r1/n1. Ilyen reprezentáció használatával az előbb ismertetett forgatókönyvekhez rendelt távolságok a következőek [4]:

- távolság(/d1/r1/n1, /d1/r1/n1) = 0 (Adatok feldolgozása ugyanazon a szerveren)
- távolság(/d1/r1/n1, /d1/r1/n2) = 2 (Egy rack-ben található különböző szervereken adatok feldolgozása)
- távolság(/d1/r1/n1, /d1/r2/n3) = 4 (Feldolgozás különböző rack-ekben található szervereken)
- távolság(/d1/r1/n1, /d2/r3/n4) = 6 (Adatok feldolgozása eltérő adatközpontokban található szervereken)

Ezek alapján vizsgáljuk meg a Hadoop alapértelmezett blokk elhelyezési stratégiáját. Legyen a replikációs faktor értéke az alapértelmezett, azaz 3. Az első replikát a klienssel [**?**] megegyező szerverre rakja le, amennyiben a kliens a klaszteren belül található. Ha a kliens a klaszteren kívül helyezkedik el, úgy az algoritmus véletlenszerűen választ egy szervert az első replikának. A második replika egy olyan véletlenszerűen választott rack-ben lesz elhelyezve, amely különbözik az első példány rack-jétől. A harmadik replika pedig a második példányt tároló rack-ben lesz elhelyezve, azonban az előzőtől eltérő, véletlenszerűen kiválasztott szerveren. Nagyobb replikációs faktor esetén a további replikákat a klaszterben található véletlenszerűen kiválasztott csomópontokra teszi az algoritmus úgy, hogy megpróbálja elkerülni azt, hogy túlterheljen egy rack-et.

3.2. Egy új hálózati modell a HDFS-hez

Az általam elkészített modellben egy gráffal reprezentálom a fizikai hálózati topológiát. A gráf egyes csomópontjai lehetnek szerverek, switch-ek, gateway-ek. A modellben szerepel továbbá egy központi csomópont, amely az Internetet reprezentálja. Ezek a csomópontok csoportokba vannak szervezve, melyek a fizikai világban az egyes klasztereket reprezentálják. A klaszterekből kettő félét különböztetek meg. Az első típus az adatközpont, amely nagy tárolási kapacitással rendelkezik, azonban hálózati szempontból messze található a felhasználótól vagy általában az adatok keletkezési helyétől, például szenzorok által mért adatoktól. A második típus a hálózat széleihez közel elhelyezett perem klaszter, mely kevés tárolási erőforrással rendelkezik (mindössze néhány szerver), azonban hálózati szempontból közel található bizonyos felhasználókhoz, végberendezésekhez, szenzorokhoz. A blokkok tárolása a szervereken megy végbe. A szerverek a klasztereken belül rack-ekbe vannak továbbá csoportosítva. Minden szerver rendelkezik tárolási kapacitással, mely megadja, hogy hány replika tárolható az adott szerveren. A modell megalkotása során úgy vettem, hogy minden replika ugyanannyi helyet foglal a szervereken, tehát a blokkok mérete egységes. A gráf csomópontjait irányítatlan élek kötik össze, melyek fizikai linkeknek felelnek meg. A 3.1. ábrán egy most leírt elrendezés látható.



3.1. ábra. *Földrajzilag elosztott hálózati topológia HDFS blokkok elhelyezésére*

Ehhez hasonló topológiákban célom a HDFS blokk replikák elhelyezése lehetőleg minél megbízhatóbban elérhető szervereken. A modellemben az éleket egy pozitív számmal súlyozom, melyek megadják az él elromlási valószínűségét. Ezt a súlyt használva számítom ki az egyes szerverek elérhetőségének megbízhatóságát. A 3.3 alfejezetben bemutatom az általam

elkészített polinomiális futási idejű algoritmust a HDFS blokk replikák elhelyezésére.

Legyen az adott topológia egy gráffal reprezentálva, ahol az egyes pontok a topológiában található szerverek, switchek és egyéb csomópontok, az élek pedig az őket összekötő hálózati linkek. Legyen továbbá minden élhez hozzárendelve egy pozitív p valószínűségi érték, amely megadja az él kiesési vagy elromlási valószínűségét. Jelöljünk ki a gráfunkban egy pontot, ami minden esetben a kiinduló pont lesz. Két csomópont között vezető út értéke, amely az úton lévő élek súlyaiból számolandó, megmondja az út elromlási valószínűségét.

Ekkor a feladat meghatározni azt az x darab csomópontot, amelyek rendelkeznek szabad kapacitással és egy adott pontból nézve a legkisebb az elromlási valószínűségük. Jelölje itt x a blokkok replikációs faktorát, ami legyen a mi esetünkben kettő. Tegyük fel, hogy minden élhez tartozó p egyenlő. Keressünk a gráfban kettő olyan pontot, amelyekhez vezető utak éldiszjunktak; két út éldiszjunkt vagy élidegen, ha nincs közös élük. Ekkor mivel az utak függetlenek, a közös elromlási valószínűség értékét a következő egyenlettel közelítjük: $p_1p_2 = p^2l_1l_2$, ahol l_1, l_2 a szerverhez vezető utak hossza. Legyenek az ugyanezekhez a pontokhoz vezető utak nem éldiszjunktak, azaz legyen legalább egy olyan él, ami a kettő útban közös. Ekkor az utak közös elromlási valószínűségének értéke nagyobb egyenlő mint p. Keressük meg azt a p küszöbértéket, amelyre teljesül, hogy a tőle kisebb értékek esetén az azonos csúcsokhoz vezető egyenlő hosszúságú utak közül az éldiszjunkt utak a megbízhatóbbak, azaz teljesül rá a következő egyenlőtlenség: $p^2l_1l_2 < p$.

$$p < \frac{1}{l_1l_2} \tag{3.1}$$

Az l_1, l_2 felülről becsülhető a gráfban található élek számával. Legyen a gráf éleinek száma e, ekkor a következő egyenlőtlenség teljesülése esetén: $p < \frac{1}{e^2}$ biztos, hogy az éldiszjunkt utak lesznek a megbízhatóbbak. p elegendő kis értékeire tehát az éldiszjunkt utak megtalálása feladat egy általános gráfban. Rendeljünk az élekhez kapacitásokat, melyek egységnyi értékűek. Ekkor az éldiszjunkt utak megtalálása megegyezik az egy értékű folyamok megtalálásával, amire a [27] cikk szerzői bebizonyították, hogy már kettő folyam esetén is NP-teljes feladat, amiből következik, hogy az általam definiált probléma is az.

3.3. Hálózati kiesésekre tervezett HDFS blokkelhelyezési heurisztikus algoritmus

Ebben az alfejezetben bemutatom az elkészített algoritmusom két legfontosabb komponensét, valamint meghatározom a függvények bonyolultságát.

A 1. algoritmus a program indulásakor egyszer fut le. Ez a függvény határozza meg az egyes szerverekhez tartozó megbízhatósági értékeket. Első lépésként minden rack-hez meghatározza az élsúly szerinti legrövidebb utat. Mivel az élsúlyok az élek meghibásodásának a valószínűségét mutatják, ezért egy szerver megbízhatósága közelíthető a következő képlettel:

$$1 - \sum_{u,v \in V} p_{u,v} \tag{3.2}$$

ahol $p_{u,v}$ az u és v csomópont közti él súlyát jelöli minden olyan u, v csomópontra, mely megtalálható az „INTERNET” csomópont és a rack közötti úton. Álljon egy út kettő élből, p_i és p_j elromlási valószínűségekkel. Ebben az esetben az úthoz tartozó szerver megbízhatósága: $(1-p_i)(1-p_j) = 1-p_i-p_j+(p_ip_j)$. p_i és p_j elegendően kis értékei esetén a (p_ip_j) tag elhanyagolható, így megkapjuk a 3.2 közelítést.

Látható tehát, hogy ahhoz, hogy egy szerver a legmegbízhatóbb legyen, maximalizálni kell a 3.2 képlet eredményét, tehát minimalizálni kell az élek súlyának összegét. A legrövidebb út megtalálásához használható Dijkstra algoritmusa, melynek komplexitása $\mathcal{O}(e + nlogn)$, ahol n a gráf csúcsainak száma e pedig a gráf éleinek a száma.

A következő lépésként sorba rendezem a klaszterben található szervereket a kiszámított megbízhatósági értékük szerint. A sorba rendezés komplexitása: $\mathcal{O}(nlogn)$. Ezek alapján a teljes inicializáló fázis lépésszámának (melyben meghatározom a megbízhatósági értéket) felső korlátja: $\mathcal{O}(n(e + nlogn))$, melyről belátható, hogy polinomiális.

Algorithm 1 Rack-ek sorbarendezése klaszterekben megbízhatóságuk szerint

```
racks ←DC.get_racks()
for each r ∈ racks do
    path_value ← calculate_shortest_path_from_INTERNET_to_rack(from="INTERNET", to=r)
    reliability ← 1 − path_value
    for each s ∈ r.servers do
        s.reliability ← reliability
ordered_rack_list ← order_by(reliability)
```

A következő fontos mechanizmusa az algoritmusomnak a klaszter kiválasztása az egyes replikáknak. A metódus pszeudokódját a 2. kódrészlet mutatja. A függvényem először sorba rendezi a klasztereket az aktuális kihasználtságuk szerint. A sorba rendezés komplexitása: $\mathcal{O}(nlogn)$. Ezt követően kétféle esetet különböztetek meg a replikációs faktor függvényében. Amennyiben a replikációs faktor kisebb mint három, az első replikát a legkevésbé terhelt klaszterben, a második replikát pedig a második legkevésbé terhelt klaszterben helyezem el. Amennyiben a replikációs faktor nagyobb mint kettő, a következő feltételek szerinti sorrendben próbálom elhelyezni az első két replikációt:

- Egy olyan klaszterben, amely felhő adatközpont és van legalább kettő különböző rack-je, amiben van szabad hely.
- Kettő különböző klaszterben, amelyek felhő adatközpontok, továbbá még nem voltak használva az adott blokk replikáinak elhelyezése során.
- Kettő különböző klaszterben (lehet akár perem klaszter is), amelyek még nem voltak használva az adott blokk replikáinak elhelyezése során.
- Az aktuálisan legkevésbé kihasznált klaszterben.

A fennmaradó replikákat a program megpróbálja „szétszórni” az egyes perem klaszterek között. Ez azt jelenti, hogy amíg van olyan perem klaszter, ami nem volt használva, addig oda helyez el egy-egy replikát, amennyiben már nem áll rendelkezésre ilyen klaszter, úgy először olyan klasztert keres ami még nem volt használva, függetlenül a típusától, ha ilyen sem létezik akkor a legkevésbé terhelt klasztert választja.

A függvény futása során megfigyelhető, hogy legrosszabb esetben háromszor kell megvizsgálnunk az topológiában szereplő klasztereket és a bennük található szervereket. Ezek alapján a klaszter kiválasztásának lépésszáma nem nagyobb mint $\mathcal{O}(4n)$, ahol n a szerverek száma. Ezekből következtethető, hogy a klaszter kiválasztásának a komplexitása: $\mathcal{O}(r(nlogn + 4n))$ ahol r a replikációs faktor.

Algorithm 2 Klaszterek kiválasztása a replikáknak

```
1: if replication_factor >2 then
2:     two_replica_in_one_DC ← False
3:     for (i = 0; i < replication_factor; i++) do
4:         ordered_DC_list ← order_DCs_by_utilization(topology)
5:         while i < 2 do
6:             if any(DC in ordered_DC_list where DC.type! = "edge" and DC.available_racks > 1) and
   two_replica_in_one_DC is False then
7:                 chosen_DC ←first DC from ordered_DC_list where DC.type! = "edge" and
   DC.available_racks > 1
8:                 two_replica_in_one_DC ← True
9:             else
10:                if any(DC in ordered_DC_list where DC.type! = "edge" and DC not used then
11:                    chosen_DC ←first DC from ordered_DC_list where DC.type! = "edge" and DC not used
12:                else
13:                    if any(DC in ordered_DC_list where DC not used then
14:                        chosen_DC ←first DC from ordered_DC_list where DC not used
15:                    else
16:                        chosen_DC ←first DC from ordered_DC_list
17:        if any(DC in ordered_DC_list where DC.type == "edge" and DC not used) then
18:            chosen_DC ←first DC from ordered_DC_list where DC.type == "edge" and DC not used
19:        else
20:            if any(DC in ordered_DC_list where DC not used) then
21:                chosen_DC ←first DC from ordered_DC_list where DC not used
22:            else
23:                chosen_DC ←first DC from ordered_DC_list
24: else
25:     for (i = 0; i < replication_factor; i++) do
26:         chosen_DC ← (ordered_DC_list[i])
```

A klaszter kiválasztása után a szerver kiválasztása következik, ahol a replika ténylegesen elhelyezésre kerül. Egy klaszteren belül az algoritmus ismeri a szerverek sorrendjét a megbízhatóságuk szerint. Ebből a sorba rendezett listából választ szervert úgy, hogy ha ugyanebből a blokkból még nincs replika ebben a klaszterben, akkor a legmegbízhatóbbat választja, ahol természetesen még van szabad hely. Amennyiben azonban ugyanebből a blokkból már található replika ebben a klaszterben, úgy a rendezett listában megkeresi az első olyan szervert, amelynek van szabad erőforrása és az előzőtől eltérő rack-ben található. Ha nincs ilyen, akkor az előzővel megegyező rack-ből azt a szervert választja, ahol a legtöbb szabad hely található. Fontos megjegyezni, hogy egy blokkhoz tartozó kettő replika kerülhet egy rack-en belül két különböző szerverre, azonban egy blokkhoz tartozó két replika soha nem kerülhet ugyanarra a szerverre.

3.4. A javasolt algoritmus szimulációs vizsgálatának eredményei

Az általam elkészített algoritmust és a Hadoop által használt alapértelmezett algoritmust szimulációkkal hasonlítottam össze. Ehhez saját szimulációs környezetet implementáltam. Több topológiát is megvizsgáltam: a különböző topológiákban különböző típusú és számú klaszterek szerepeltek. A 3.1. táblázatban látható a szimulációkban használt klaszter típu-

sok és a bennük található szerverek, switchek és átjárók száma. Korábban említettem, hogy a klaszterek két fajtáját különböztetem meg, a perem klasztereket és az adatközpontokat. Az adatközpontok közül a gyakorlatban legelterjedtebb CLOS [28] és Fat-tree [29] elnevezésű topológiákat használtam. Látható a 3.1. táblázatban, hogy három különböző méretű CLOS adatközpontot használtam. A táblázatból megfigyelhető továbbá, hogy az az egyes klasztereknél variáltam a bennük található átjárók számát. A perem klaszterek esetében egy vagy kettő, míg a CLOS adatközpontok esetében egy vagy három átjárót tartalmazott a klaszter.

	perem klaszter	clos1	clos2	clos3	fat-tree
szerverek	20	800	2000	2800	2560
switchek	1	70	160	200	210
átjárók	1-2	1-3	1-3	1-3	2

3.1. táblázat. *Hálózati topológiát figyelembe vevő szimulációk során használt klaszter típusok*

Minden topológia ötven darab perem klasztert és négy-öt nagy adatközpontot tartalmaz. A szimulációk során minden algoritmus betelíti a szervereket a blokkok replikáival, majd véletlenszerűen elvesz csomópontokat a topológiából addig, amíg adatvesztés nem történik. Adatvesztésről akkor beszélünk, ha egy blokkhoz tartozó összes replika elérhetetlenné válik az „INTERNET" csomópont felől.

Az egyes adatközpontokban a switcheknek és a szervereknek „négy kilences" rendelkezésre állást, míg a perem klaszterekben a csomópontoknak „három kilences" rendelkezésre állást feltételezünk. Ez a szimuláció során azt jelenti, hogy a perem klaszterekben lévő szerverek és switchek tízszer nagyobb valószínűséggel hibásodnak meg, és kerülnek törlésre a szimulációban, mint az adatközpontban lévők. Fontos megjegyezni, hogy a szimuláció során egy csomópont törlése nem csupán az eszköz meghibásodását illusztrálja, hanem az esetlegesen előforduló félre konfigurációkat, hardver cseréket, áramszünetet, stb.

Minden szimuláció két algoritmust hasonlít össze. Először a Hadoop alapértelmezett algoritmusa elhelyezi a blokkok replikáit, majd a szimulátor véletlenszerűen kitöröl csomópontokat amíg adatvesztés nem lesz. A kitörölt csomópontokat a szimulátor sorrendben elmenti. Ezt követően a saját algoritmusom helyezi el a replikákat az eredeti topológián, majd az előző futás során elmentett törölt csomópontokat szintén elveszi a topológiából, ugyanabban a sorrendben. Amennyiben még nem következett be adatvesztés, úgy folytatja a csomópontok véletlenszerű kiválasztását és elvételét az első adatvesztés bekövetkeztéig.

A szimulációk végeztével az egyik legjelentősebb információ a törölt és a kiesett csomópontok száma az adatvesztés bekövetkeztekor. A szimulációkat a több különböző topológiára lefuttattam. A 3.2. ábrán láthatóak a szimulációs eredmények, melyek a 3.2 táblázatban látható topológiák használatával készültek.

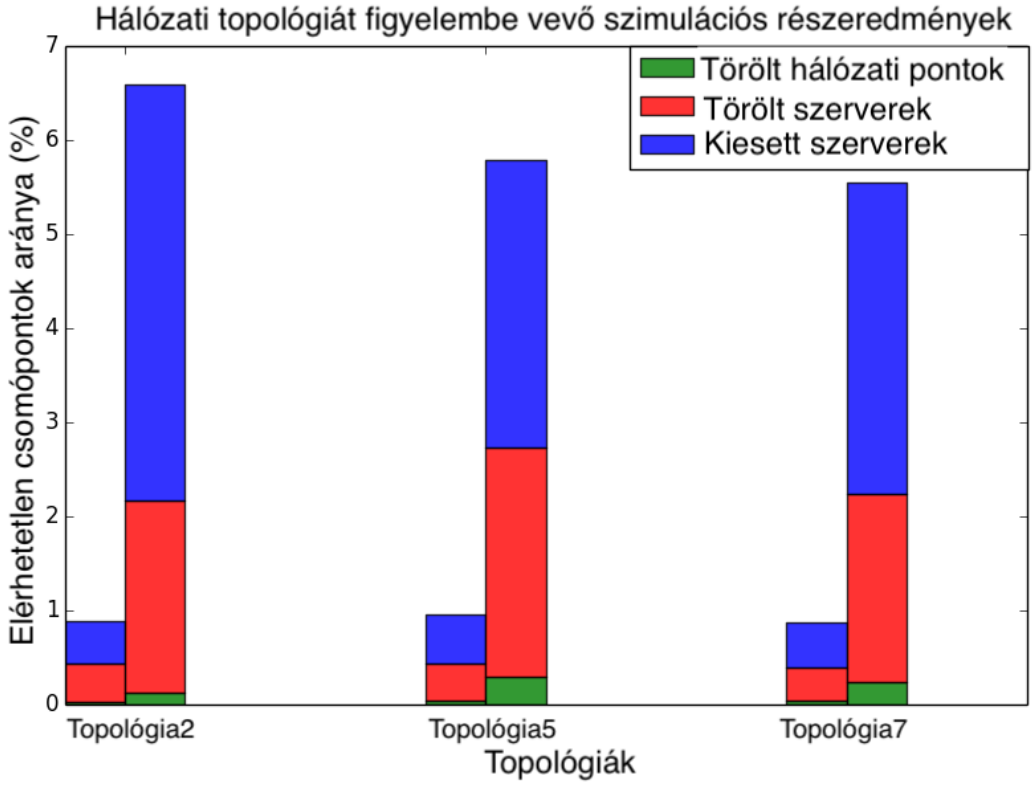
Azért ezt a három topológiát választottam ki, mert ezen eredményekkel tudom a legszemléletesebben bemutatni a különböző típusú adatközpontokra és átjáró számra jellemző eredményeket, ugyanis a Topológia2-ben csak CLOS adatközpontok találhatók, és a perem klaszterekben csak egy átjárójuk van. A Topológia5-ben vegyesen található CLOS és Fat Tree adatközpont, továbbá ebben az esetben a perem klaszterekben már kettő átjárójuk

	perem klaszter	clos1	clos2	clos3	fat-tree
Topológia2	50 (1 átjáró)	-	2 (1 átjáró)	2 (1 átjáró)	-
Topológia5	50 (2 átjáró)	2 (1 átjáró)	-	-	3
Topológia7	50 (2 átjáró)	-	-	-	4

3.2. táblázat. *Hálózati topológiát figyelembe vevő szimulációkban használt klaszterek*

van. A Topológia7-ben pedig csak Fat Tree adatközpont szerepel.

Az ábrán az egyes oszlopcsoportokban a bal oldali oszlop mutatja az alapértelmezett HDFS algoritmus szimulálása során kapott eredményt, míg a jobb oldali oszlop a saját algoritmusom eredményeit mutatja.



3.2. ábra. *Az első adatvesztésig kiesett hálózati pontok aránya az alapértelmezett HDFS, és az ajánlott algoritmus esetén*

A 3.2. ábrán az y-tengely az elérhetetlen csomópontok százalékos arányát mutatja a topológiában található összes csomóponthoz képest, abban a pillanatban, amikor az első HDFS blokk összes replikája elérhetetlenné válik. Az elérhetetlen csomópontok magukban foglalják a törölt és az elérhetetlenné vált pontokat egyaránt.

A grafikonon az első oszlopcsoport a Topológia2-n futtatott szimulációs eredményeket mutatja. Megfigyelhető, hogy mind a három topológia esetén a HDFS alapértelmezett algoritmusát használva elég körülbelül a csomópontok egy százalékát elveszteni ahhoz, hogy adatvesztés következzen be. Ehhez képest az általam javasolt algoritmus használata esetén körülbelül hatszor annyi csomópont elérhetetlenné válását viseli el a rendszer adatvesztés nélkül. Látható, hogy az algoritmusom a Topológia2 esetén teljesített a legjobban, vagyis

amikor csak CLOS adatközpont található a rendszerben. Ez az eredmény azzal magyarázható, hogy a CLOS topológiában általában az egyes pontok fokszáma nagyobb, mint a Fat Tree adatközpontokban található pontoké. Megfigyelhető továbbá a grafikonon, hogy a perem klaszterekben az átjárók számának növelése lényegében nem volt hatással az eredményekre.

4. fejezet

Számítási késleltetés csökkentése az adatelemző komponensek okos elhelyezésével

A fejezet során bemutatásra kerülnek a Hadoop ökoszisztémában használt alapvető erőforrás-allokációs algoritmusok, melyek olyan általános alkalmazások komponenseit helyezik el, amelyek felépítése *map* illetve *reduce* feladatokból áll. Az algoritmusok elemzésével bemutatom azok limitációit több adatközpontból és perem klaszterből álló, földrajzilag elosztott topológia esetén. A korlátozások vizsgálatakor nagy figyelmet szentelek a hálózati késleltetés által okozott teljesítményromlásra. A hálózati késleltetések alapján egy teljesítményjavító problémát fogalmazok meg, melyet formalizálok, és egy megoldást kínálok rá.

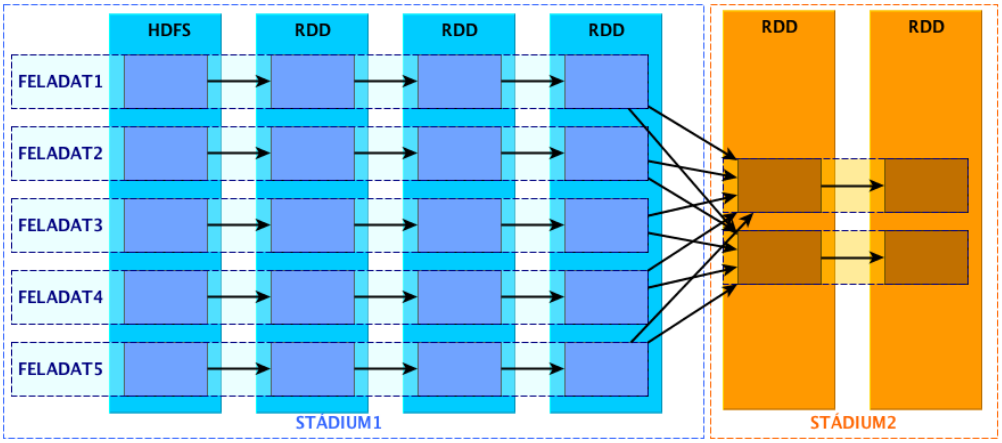
A javasolt megoldásom egy olyan új algoritmus, amivel kiküszöbölöm a korábban említett algoritmusok hiányosságait egy földrajzilag elosztott topológián. A célja az algoritmusomnak az, hogy csökkentsem egy Big Data munkafolyamat végrehajtási idejét egy földrajzilag elosztott topológián úgy, hogy a Big Data komponensek közötti hálózati késleltetést minimalizálom. Ezek után bemutatom az algoritmusom implementációját, numerikus értékelésének környezetét, amelyben szimulációkkal bizonyítom az algoritmus működésének helyességét, gyorsaságát, és a vele elérhető teljesítményjavulást. Az algoritmusomat összehasonlítom egy jelenleg is elterjedt megoldással és bemutatom a szimulációs eredményeimet.

4.1. A Spark feladatelhelyező algoritmusa

Manapság a Big Data rendszerekben legelterjedtebben használt adatfeldolgozó technológia a nyílt forráskódú Spark. A legtöbb ilyen rendszerben a Spark szorosan együttműködik a YARN-nal. Az együttműködésük során a YARN szolgálja ki a Spark által beküldött erőforráskéréseket. A Spark egy beküldött alkalmazásból egy irányított körmentes gráfot (DAG-ot) csinál a benne található DAGScheduler (Directed Acyclic Graph Scheduler - Irányított körmentes gráf ütemező) segítségével.

A DAGScheduler az ütemező rétege a Spark-nak, mely stádium-orientált ütemezést valósít meg. Logikai megvalósítási terveket transzformál át fizikai megvalósítási tervekké. A DAGScheduler a beküldött alkalmazást stádiumokra és feladatokra osztja. Egy stádium a végrehajtás fizikai egysége, vagyis egy lépés a fizikai megvalósítási tervben. A stádium egy, vagy több párhuzamosan futtatható feladatot tartalmaz magában. A feladatok számát a szükséges partíciók száma határozza meg.

Az adatok érkezhetnek a HDFS által eltárolt blokkokból, amelyeket a Spark alkalmazás beolvas és a keretrendszer által kezelt típusra, RDD-re (Resiliend Distributed Dataset) transzformál és használ további műveletekre. Egy ilyen felépítést szemléltet a 4.1 ábra is.



4.1. ábra. *Spark feladat, stádium, és adategység*

Kettő típusú stádiumot különböztetünk meg.

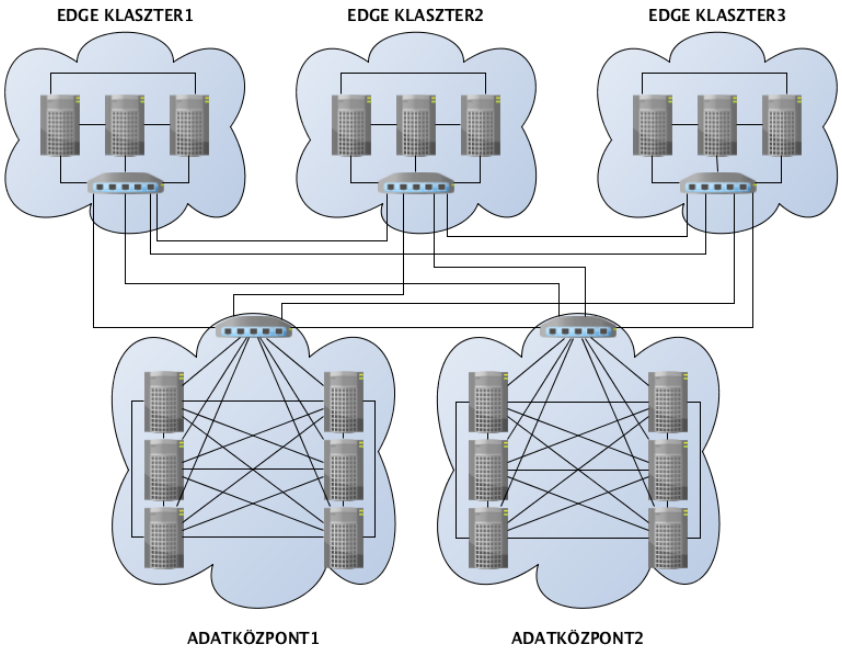
- ShuffleMapStage: egy köztes stádium, amely adatot nyújt más stádiumoknak. Az 4.1 ábrán például stádium1.
- ResultStage: végső stádium, amely Spark akciót hajt végre. Az 4.1 ábrán például stádium2. Az akció egy olyan művelete a Spark-nak, amely egy eredménnyel tér vissza.

Mint látjuk a Spark tehát feladatokat használ az adatok beolvasására és feldolgozására. Azonban a Big Data rendszerekben az adatok elosztottan helyezkednek el a klaszterünkben/klasztereinkben. Ezért a Spark (és az alatta lévő YARN is) az adatok helyére támaszkodik, vagyis a Spark feladatok érzékenyek az adathoz való közelségre. Az ilyen jellegű feladatok az erőforráskérésükben megadják, hogy melyek azok a szerverek, amelyeket preferálják a futásukhoz (amelyeken a feldolgozandó adatrészek találhatóak). Ezt a keretrendszer úgy próbálja biztosítani, hogy amennyiben a kijelölt szerverek közül valamelyik rendelkezik szabad számítási erőforrással, úgy azon indítja el. Ellenkező esetben az algoritmus megkísérli ugyanabban a rack-ben, egy másik szerveren lefuttatni a feladatot. Ha ez sem lehetséges akkor véletlenszerűen választ az egész topológiából egy szabad erőforrással rendelkező szervert a feladat futtatására.

4.2. Big Data alkalmazások végrehajtási idejének modellje hálózati paraméterekkel

A hálózati topológiát egy gráffal reprezentálom. A gráfban a csomópontok lehetnek szerverek, vagy átjárók, melyeket a gráf irányítatlan élei kötnek össze. A gráf csúcsait összekötő élekhez egy pozitív egész szám van hozzárendelve, mely az él két végpontja között mérhető késleltetést mutatja. Minden szerver rendelkezik egy szintén pozitív egész kapacitással, mely meghatározza, hogy hány számítási funkció helyezhető el a szerveren. Feltételezem ebben az esetben, hogy az egyes számítási funkciók, feladatok ugyanannyi, egységes erőforrás használattal futnak a topológián.

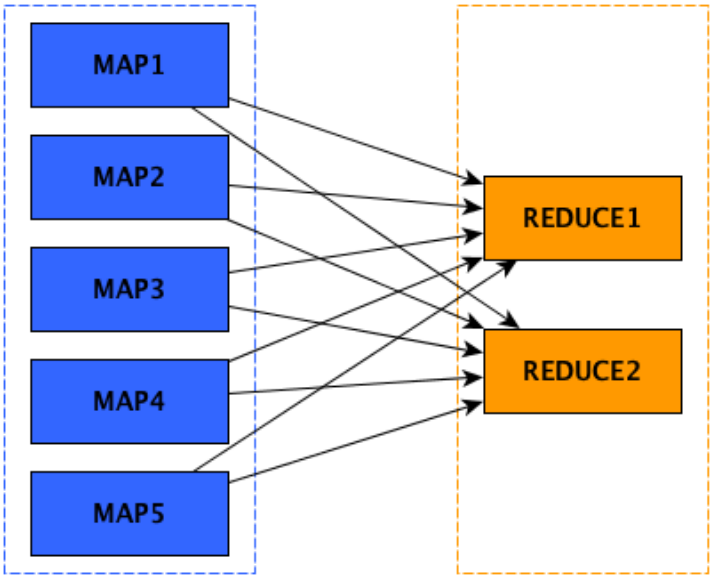
Az egyes feladatok elhelyezése és futtatása a szervereken történik. A szervereket rackekbe, a rack-eket pedig klaszterekbe csoportosítom. Minden klaszter rendelkezik egy átjáróval, amely kapcsolatot nyújt a klaszterben található szerverek és a többi klaszter között. Egy példa ábrázolása a modellemnek a 4.2. ábrán látható. A klaszterekből kettő típust különböztetek meg: az első az adatközpont, amely sok számítási kapacitással rendelkezik, a második típus a perem klaszter, mely kevesebb számítási erőforrással rendelkezik (mindössze néhány szerverrel).



4.2. ábra. *Földrajzilag elosztott Big Data infrastruktúra*

Ezek alapján megfigyelhető a topológiámban, hogy az átjárók által alkotott részgráf egy klikket alkot, továbbá az egyes klaszterek által alkotott részgráfok is klikkeket alkotnak.

Az 4.3. ábrán egy általános nézete látható az általam vizsgált Big Data alkalmazás feladatainak.



4.3. ábra. *Map és reduce feladatokból álló kérések sémája*

Az alkalmazásokat szintén egy gráffal reprezentálom. A gráfban egy feladat vagy funkció egy csomóponttal van reprezentálva, a köztük futó élek pedig a köztes adatok által meghatározott függőségeket jelölik. Ezek a kérés gráfok teljes páros gráfokat alkotnak, amiben az egyik csomópont halmaz a *map* funkciókat, a másik csomópont halmaz pedig a *reduce* funkciókat tartalmazza.

A feladat egy olyan algoritmus készítése, mely *map* és *reduce* funkciókat helyez el a topológián úgy, hogy a *map* funkciók helyétől az egyes *reduce* funkciókhoz vezető utakon a linkek késleltetéseinek legnagyobb összege a lehető legalacsonyabb legyen.

A feladatra egy ismert NP-teljes probléma visszavezethető. A *map* és *reduce* funkciók által alkotott alkalmazás gráf egy teljes páros gráf. Vegyük a topológia gráfnak egy olyan logikai nézetét, amelyben a csomópontok megfelelnek a fizikai topológia csomópontjaival, az élek pedig a fizikai topológiában a két csomópont közötti legrövidebb út késleltetésének az értéke. Jelöljük ezt a logikai gráfot G-vel. Megfigyelhető hogy G egy teljes gráf. Legyen G-ben minden csomópont kapacitása egy. Ekkor a feladatunk a logikai teljes gráfban megtalálni azt a részgráfot, amely megegyezik az alkalmazás gráfban található funkciók leképzése során kapott gráffal. Belátható, hogy ez a leképzett részgráf szintén egy teljes páros gráfot alkot úgy mint az alkalmazás gráf.

Legyen G_x a G gráf részgráfja, mely az első x darab legkisebb késleltetéssel rendelkező élet tartalmazza G-ből. Ekkor a feladatunk, hogy eldöntsük, hogy a leképzett teljes páros gráf része-e a G_x részgráfnak. Egy kiegyensúlyozott teljes páros részgráf megtalálása egy páros gráfban bizonyítottan NP-teljes probléma [30]. Kiegyenlítettnek nevezzük azt a páros gráfot, amelyben mindkét csúcshalmaz egyenlő számú ponttal rendelkezik. Mivel a páros gráfok halmaza részhalmaza az általános gráfok halmazának, így általános gráfokban kiegyenlített teljes páros gráfot keresni is NP-teljes. Ezek alapján mivel a kiegyenlített teljes páros gráf speciális esete a teljes páros gráfoknak, így egy teljes páros gráf megtalálása

általános gráfokban NP-teljes probléma.

Az algoritmusom által megoldandó feladat formális leírásához szükséges jelöléseket a 4.1. táblázat tartalmazza.

Jelölés	**Jelentés**
$V_s(V_m, V_r), E_s$	Az alkalmazás gráf csúcsai és élei
V_t, E_t	Topológia gráf csúcsai és élei
$x_u^i : i \in V_s, u \in V_t$	1, ha az i. feladat az u. csomópontra lett leképezve 0, különben
$y_{u,v}^{i,j} : (i,j) \in E_s, (u,v) \in E_t$	1, ha (u,v)-t tartalmazza az (i,j) alkalmazás gráfbeli él fizikai útvonala 0, különben
r_i	Az i. végrehajtó által megkövetelt erőforrások
ρ_u	A u. csomópontban rendelkezésre álló erőforrások
$\delta_{u,v}$	az (u,v) fizikai link késleltetése
$\beta_{u,v}$	az (u,v) fizikai linken elérhető sávszélesség

4.1. táblázat. *Késleltetést és sávszélességet figyelembe vevő feladatok matematikai jelölései*

A megoldandó feladatot a következő feltételekkel és célfüggvénnyel fogalmazom meg.

$$\forall i \in V_s : \sum_{u \in V_t} x_u^i = 1 \tag{4.1}$$

$$\forall u \in V_t : \sum_{i \in V_s} x_u^i r_i \leq \rho_u \tag{4.2}$$

$$\forall (i,j) \in E_s, \forall u \in V_t : \sum_{v:(u \to v) \in V_t} y_{u,v}^{i,j} - \sum_{w:(w \to u) \in V_t} y_{w,u}^{i,j} = x_u^i - x_u^j \tag{4.3}$$

$$\min_{r \in V_r}(\max_{i \in V_m}(\sum_{(u,v) \in E_t} y_{u,v}^{i,r} \delta_{u,v})) \tag{4.4}$$

A leképzés során teljesül, hogy minden feladat pontosan egy fizikai csomóponthoz van hozzárendelve (4.1). Egy fizikai szerverre elhelyezett virtuális funkciók erőforrásainak összege nem haladja meg a szerver összes fizikai erőforrását (4.2). Minden szerverre teljesül, hogy a hozzá bemenő virtuális élek száma megegyezik a belőle kimenő virtuális élek számával(4.3). Az optimalizálás célfüggvénye az, hogy az összes *map* funkció fizikai helyétől az egyes *reduce* funkciókhoz vezető úton található késleltetés értékek összegének a maximuma a lehető legkisebb legyen (4.4).

4.3. Big Data feladat teljesítési idejét minimalizáló heurisztikus algoritmus

A fent megfogalmazott nehéz feladat megoldására a következő heurisztikát javaslom. Egy Big Data alkalmazás kérés feldolgozása során az algoritmusom végigmegy a benne található

feladatokon, és elhelyezi azokat a hálózati topológián. Először a *map* funkciókat helyezem el a következő eseteket vizsgálva sorrendben:

- A *map* funkcióban definiált preferált szerverek valamelyike rendelkezik-e szabad számítási erőforrással. Amennyiben igen, úgy a feladatot arra a szerverre helyezem el.
- Ha a preferált szerverek egyike sem rendelkezik szabad erőforrással, akkor megpróbálom a preferált szerverekkel megegyező rack-ekbe elhelyezni egy másik, szabad erőforrással rendelkező szerveren.
- Ha a preferált rack-ekben sincs szabad erőforrás, akkor minden klaszterből a legkevésbé terhelt szerverhez kiszámítom a késleltetést a preferált szerverektől és azt a szervert választom, amelyikhez a legkisebb a kiszámított késleltetés.

Algorithm 3 Hálózati késleltetést figyelembe vevő SPARK adatelemző feladat elhelyezése

```
1:  for each task ∈ request.map_tasks + request.reduce_tasks do
2:      if task has locality_constraint then
3:          for each constraint_server ∈ locality_constraint do
4:              if constraint_server has available resource then
5:                  task.place_to(constraint_server)
6:                  break
7:          if task not placed then
8:              for each constraint_server ∈ locality_constraint do
9:                  if constraint_server.rack has available resource then
10:                     task.place_to(least_utilized_server_from_constraint_rack)
11:                     break
12:         if task not placed then
13:             min_delay ← ∞
14:             available_servers ←least utilized available server from each cluster
15:             for each server ∈ available_servers do
16:                 for each constraint_server ∈ locality_constraint do
17:                     delay ← get_delay_between_nodes(server, constraint_server)
18:                     if delay < min_delay then
19:                         min_delay ← delay
20:                         chosen_server ← server
21:             task.place_to(chosen_server)
22:     else
23:         min_delay ← ∞
24:         available_servers ←least utilized available servers from each cluster
25:         for each server ∈ available_servers do
26:             delay ← get_max_delay_between_nodes(server, map_places)
27:             if max_delay < min_delay then
28:                 min_delay ← max_delay
29:                 chosen_server ← server
30:             task.place_to(chosen_server)
```

A *map* funkciók elhelyezése után a *reduce* funkciók lerakása következik. A *reduce* funkciók nem rendelkeznek preferált szerver megkötéssel. Így az elhelyezésükkor az algoritmus arra törekszik, hogy hálózati késleltetés szempontjából minél közelebb legyen a már elhelyezett *map* funkciókhoz. Ehhez első lépésként minden klaszterből a legkevésbé terhelt szervereket mentem el egy listába. Ezen a listán végigmenve az összes szerver esetén eltárolom a maximális késleltetés értékét az aktuális szerver és azon szerverek között, ahova a *map* funkciók lettek leképezve. Majd a legkisebb maximális késleltetéssel rendelkező szervert választom ki a *reduce* funkció leképzési helyéül.

Az algoritmusomnak szüksége van egy inicializáló fázisra, amely során meghatározzuk az egyes csomópontok között található késleltetéseket. Ennek a komplexitása n darab cso-

mópont esetén $\mathcal{O}(n^2)$. Ezeket a méréseket az inicializálás után csak akkor kell újrafuttatni, ha a topológia változik. A feladatok elhelyezésére szolgáló metódusban egy szerver vagy rack megtalálása konstans idő alatt lehetséges. A legkevésbé terhelt szerverekből egy lista készítésének lépésszáma: $\mathcal{O}(n)$. A lista hossza megegyezik a klaszterek számával, amit jelöljünk c-vel. Legyen k az egy *map* funkcióhoz tartozó preferált szerverek száma. Két szerver között a késleltetés szerinti legrövidebb út megtalálásához Dijkstra algoritmusát használva a komplexitás: $\mathcal{O}(e + nlogn)$, ahol e a topológia éleinek számát n pedig a csomópontok számát jelenti. Egy *map* funkció elhelyezésének lépésszáma egy olyan szerveren, amely nem szerepel a preferált szerverek által meghatározott rackekben, felülről becsülhető $\mathcal{O}(n+ck(e+nlogn)))$. Legyen t az alkalmazásban szereplő feladatok száma, így a feladatokat elhelyező algoritmusom komplexitása felülről becsülhető: $\mathcal{O}(n^2+t(n+ck(e+nlogn)))$.

4.4. Szimulációs beállítások és eredmények

Saját szimulátort implementáltam, melyben egy szimulált nagy topológián hasonlítottam össze az általam elkészített algoritmust és a Hadoop ökoszisztémában használt alapértelmezett algoritmust. A szimulációk során egy topológián futtattam az alkalmazások által generált különböző erőforrás kéréseket. A szimulált topológiában ötven perem klaszter és öt adatközpont található. A szimulációk során a topológiában használt késleltetés értékeket a 4.2. táblázat mutatja. Az azonos rack-en belül lévő szerverek között található a legkisebb késleltetés, amely a szimulációk során 1 milliszekundum. Egy klaszteren belül két különböző rack-ben található szerver vagy egy szerver és az átjáró között 5 milliszekundumos késleltetést határoztam meg. A klaszterek átjárói között található a legnagyobb késleltetés, hiszen előfordulhat, hogy ezek fizikailag távol vannak egymástól. A szimulációim során 20 és 200 milliszekundum között véletlenszerűen választottam késleltetés értéket az ilyen éleknek.

kiinduló csomópont típus-cél csomópont típus	késleltetés
szerver-szerver (egy racken belül)	1ms
rack-rack (egy klaszteren belül)	5ms
szerver-átjáró (egy klaszteren belül)	5ms
átjáró-átjáró (két klaszter között)	20-200 ms

4.2. táblázat. *Szimuláció során használt késleltetések*

A szimulációk során minden kérés öt darab *map* feladatból és kettő darab *reduce* feladatból áll. Az egyes *map* feladatokhoz kényszereket definiálok, melyek megadják azokat a fizikai szervereket, amelyen a feldolgozandó adat található, tehát ahova a *map* funkciót célszerű elhelyezni. Minden *map* funkcióhoz három fizikai szervert rendelek. Ezek határozzák meg a *map* funkció preferált helyeit a topológiában. Azért három szervert definiálok, mert például a HDFS alapértelmezett replikációs faktora is három.

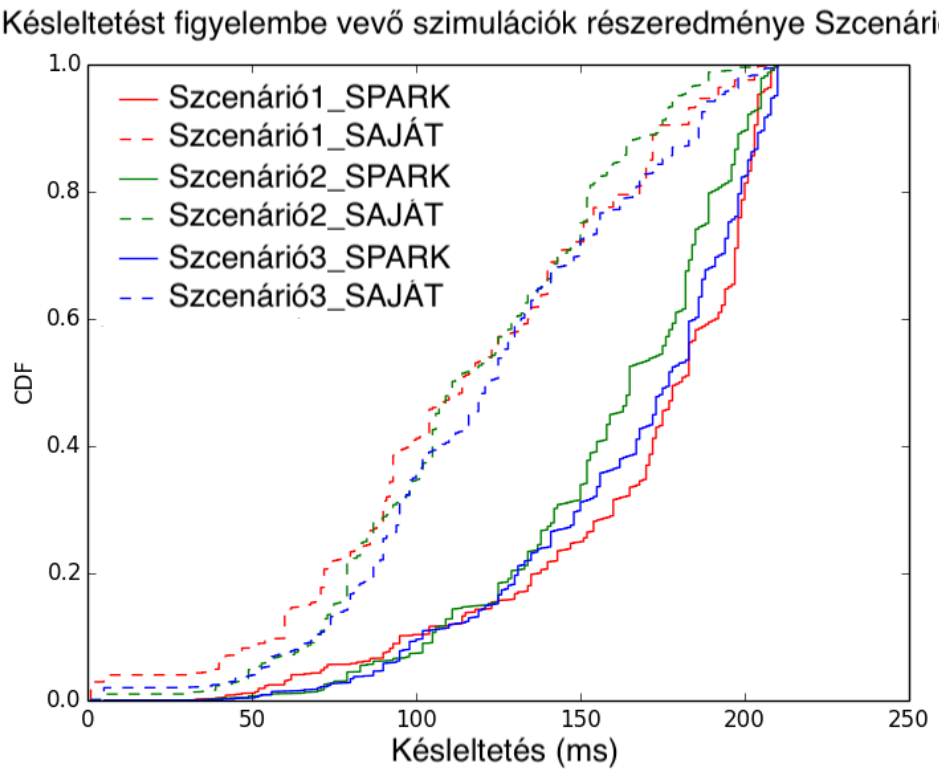
Két szimulációfutást a generált Big Data kérések típusai, azon belül is a *map* funkciók kényszereinek a kiválasztása különbözteti meg. Ezek alapján a következő szimulációs szcenáriókkal vizsgáltam algoritmusokat:

- Szcenárió1: A *map* funkciókhoz tartozó kényszerek a perem klaszterekből kerülnek kiválasztásra véletlenszerűen.
- Szcenárió2: A *map* funkciókhoz tartozó kényszerek az adatközpontokból kerülnek kiválasztásra véletlenszerűen.
- Szcenárió3: A *map* funkciókhoz tartozó kényszerek az egész topológiából kerülnek kiválasztásra véletlenszerűen.
- Szcenárió4:A *map* funkciókhoz tartozó kényszerek egy meghatározott perem klaszter szerverei közül kerülnek kiválasztásra véletlenszerűen.
- Szcenárió5: A *map* funkciókhoz tartozó kényszerek kettő meghatározott perem klaszter szerverei közül kerülnek kiválasztásra véletlenszerűen..
- Szcenárió6: A *map* funkciókhoz tartozó kényszerek három meghatározott perem klaszter szerverei közül kerülnek kiválasztásra véletlenszerűen.
- Szcenárió7: A *map* funkciókhoz tartozó kényszerek négy meghatározott perem klaszter szerverei közül kerülnek kiválasztásra véletlenszerűen.
- Szcenárió8: A *map* funkciókhoz tartozó kényszerek öt meghatározott perem klaszter szerverei közül kerülnek kiválasztásra véletlenszerűen.

Minden szimuláció alkalmával addig generálok kéréseket amíg van szabad számítási erőforrás a topológiában. Egy ősosztályból leszármaztatok két új szimulátor osztályt: a Hadoop alapértelmezett algoritmust használó szimulátort, és a saját algoritmust használó szimulátort. A két szimulátor két izomorf topológián fog dolgozni ugyanazokkal a kérésekkel. Miután mind a kettő szimulátor betelítette a saját topológiáját a kérésekkel, összehasonlítom a két algoritmust a kérések alapján elhelyezett *map* és *reduce* komponensek közötti hálózati késleltetések okozta végrehajtási idő nyúlását.

A szimulációk végén megvizsgálom az elhelyezett *map* funkciók és *reduce* feladatok között lévő maximális késleltetés értékét. Minden szimulációt tízszer ismételtem meg. A korábban felsorolt összes szcenárióra lefuttattam a szimulációkat, és összehasonlítottam a kapott eredményeket. A 4.4 CDF (Cumulative distribution function) diagrammon az első három szcenárió eredményei láthatóak. Az x-tengelyen a késleltetések szerepelnek milliszekundumban. A szaggatott vonalak mutatják a saját algoritmusom által adott eredmények eloszlását, míg a folytonos vonalak reprezentálják a SPARK ütemező algoritmusa által adott eredményeket. A piros vonalak szcenárió1 eredményeit, a zöld vonalak a szcenárió2, a kék vonalak pedig a szcenárió3 szimulációi során kapott eredményeket mutatják.

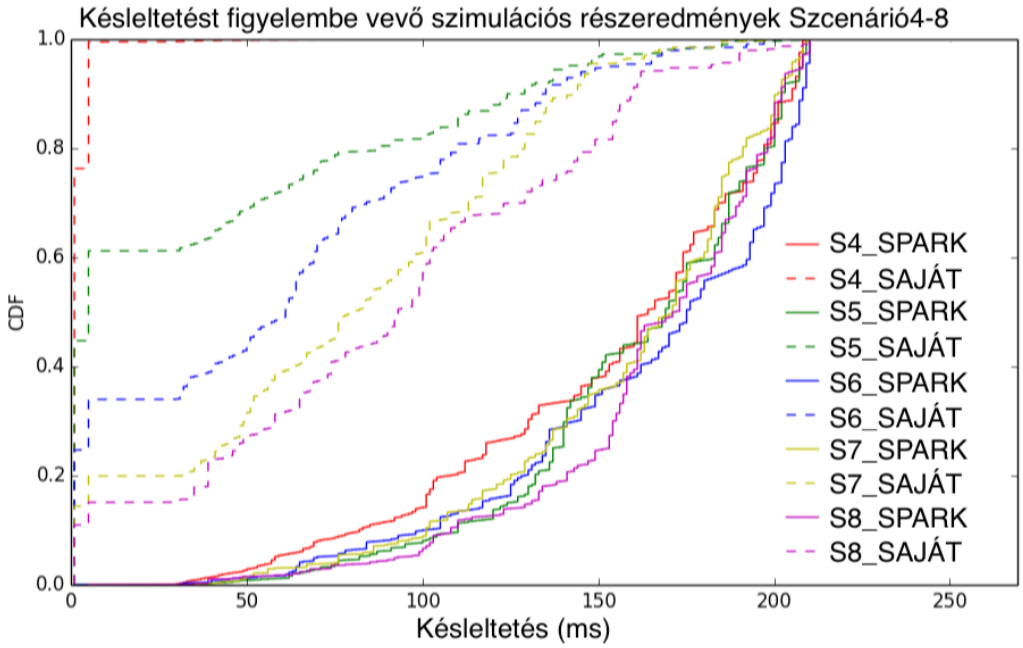
Látható, hogy az általam ajánlott algoritmus mind a három esetben jobb teljesítményt ért el, ugyanis a lerakott *map* funkciók általánosan kevesebb késleltetéssel képesek elérni a *reduce* funkciókat. Megfigyelhető, hogy az algoritmusom által elhelyezett funkciók 80 százalékában volt 150 milliszekundum, vagy az alatt a kapott késleltetési érték, míg a SPARK algoritmusa esetén az elhelyezett funkciók 80 százalékánál 200 milliszekundum vagy az alatt van a kapott késleltetési érték.



4.4. ábra. *Hálózati késleltetést figyelembe vevő, Big Data komponenseket elhelyező algoritmusok teljesítési idő eredményei*

A 4.5. CDF diagrammon, a szcenárió4-szcenárió8 szimulációk által kapott eredmények láthatóak. Itt is az x-tengely mutatja a késleltetéseket milliszekundumban. A szaggatott vonalak mutatják a saját algoritmusom által adott eredmények eloszlását, míg a folytonos vonalak reprezentálják a SPARK ütemező algoritmusa által adott eredményeket. A különböző vonal színek a különböző szcenáriók eredményeit különböztetik meg.

Jól megfigyelhető az ábrán, hogy ezen szcenáriók esetében is mindig jobb teljesítményt ért el az algoritmusom, mint a SPARK algoritmusa. Látható továbbá, hogy a megkötéseknek kiválasztható perem klaszterek számának növelésével csökken az algoritmusom teljesítménye, azaz általánosságban nőnek a *map* és a *reduce* feladatok között az elért késleltetési értékek. Ez teljes mértékben elvárt működés, ugyanis minél több perem klaszterbe helyezhetjük el a *map* feladatainkat, annál nagyobb valószínűséggel és annál több *map-reduce* kapcsolat fog átívelni a klaszterek közötti hálózaton, ahol az utaknak elkerülhetetlenül magas a késleltetési értéke.



4.5. ábra. *Hálózati késleltetést figyelembe vevő, Big Data komponenseket elhelyező algoritmusok teljesítési idő eredményei: Szcenárió4-8*

Megfigyelhető továbbá az is, hogy a SPARK algoritmusára nincs nagy hatással a kiválasztható perem klaszterek számának növelése. Ennek oka az, hogy a SPARK nem veszi figyelembe a hálózati erőforrásokat, mint például a késleltetést, így a véletlenszerű *reduce* helyének kiválasztásánál nem befolyásolja a *map* funkciókhoz tartozó megkötéseknek a kiválasztható perem klaszterek számának növelése.

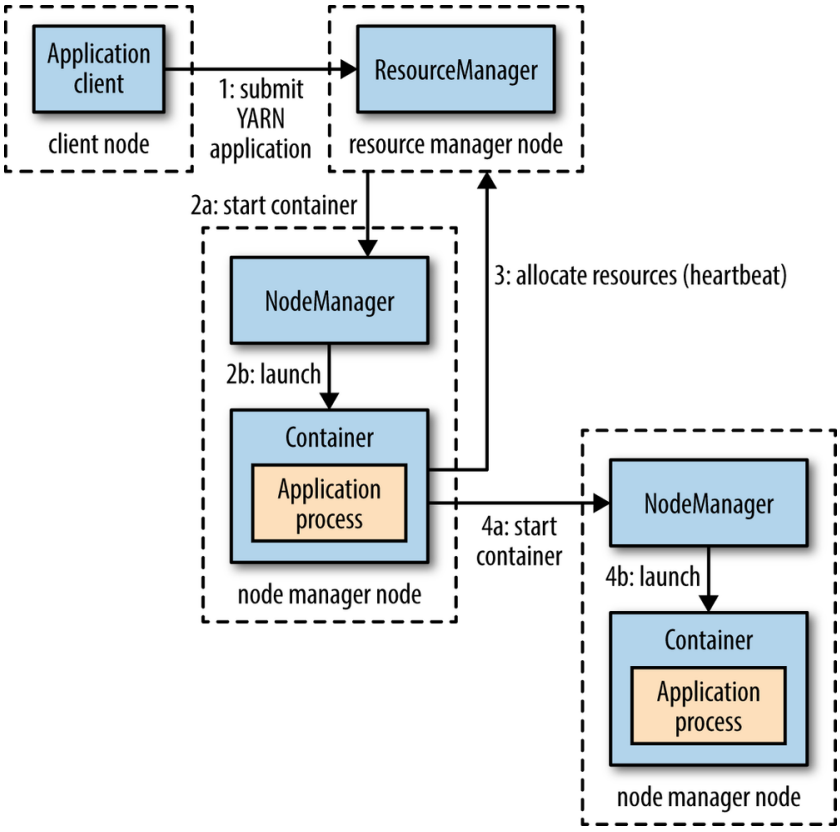
5. fejezet

YARN végrehajtók által lefoglalt WAN sávszélesség csökkentése

Ebben a fejezetben ismertetem a YARN alapértelmezett algoritmusát, amelytől egy tetszőleges alkalmazás erőforrásokat igényel a számítási komponenseinek elhelyezésére. Tanulmányozom az alkalmazott elhelyező algoritmusának jelenlegi hiányosságait a hálózati sávszélesség kihasználálásank szempontjából. A jelenlegi megoldást egy földrajzilag elosztott, több adatközpontból álló topológián szimulálom. A hiányosságok vizsgálata után megfogalmazok és formalizálok egy problémát, melyre dolgozatomban megoldást is nyújtok: egy olyan új algoritmust mutatok be, amivel javítok az alapértelmezett algoritmus működésén egy földrajzilag elosztott topológián a sávszélesség kihasználása szempontjából. Az algoritmusom célja, hogy a YARN a Big Data végrehajtók elhelyezésekor úgy kerüljenek kiválasztásra a szerverek, hogy a hálózati sávszélesség kapacitásokat figyelembe vegye az erőforrás-vezénylő algoritmus. Bemutatom az implementációmat és a szimulációs környezetet, mellyel bizonyítom a megoldásom működésének helyességét. Összevetem a jelenlegi megoldással a saját implementációmat és bemutatom az eredményeimet.

5.1. A YARN a végrehajtókat elhelyező algoritmusa

A 5.1. ábrán megfigyelhető a YARN működése erőforrásfoglalás közben [4].



5.1. ábra. *YARN erőforrás foglalása*

Vizsgáljuk meg részletesebben az 5.1. ábrán látható erőforrásfoglalás folyamatának lépéseit.

1. Ahhoz, hogy egy Big Data alkalmazást futtatni tudjon, a felhasználó első lépésben az erőforrás menedzserhez fordul és megkéri, hogy indítson egy alkalmazás mestert a számára.

2. Az erőforrás menedzser keres egy fizikai szervert, amelyen van elég számítási erőforrás az alkalmazás mester elindítására. A kiválasztott hoszton üzenetet küld a hoszt menedzsernek (2a), ami az üzenet hatására elindítja az alkalmazás mestert (2b). Az alkalmazás mester feladata a futtatott applikációtól függ. Elképzelhető, hogy egy egyszerű számítást végez a számára indított konténerben.

3. Előfordulhat azonban az is, hogy az alkalmazás mester több végrehajtót kér az erőforrás menedzsertől.

4. Az alkalmazás mester a megkapott plusz konténer erőforrásokat a hoszt menedzsereken keresztül lefoglalja és elindítja a szükséges virtuális környezeteket (4a, 4b).

A végrehajtóknak lehetnek olyan erőforrás követelményei, melyek a hozzájuk tartozó bemenet helyén alapulnak. Ezekkel a követelményekkel maximalizálja a rendszer az adat lokalitást. Ezek az követelmények a folyamat során erőforráskérésekké alakulnak, melyeket az alkalmazás mester elküld az ütemezőnek (ami az erőforrás menedzseren belül található). Minden erőforráskérésben megtalálhatóak a következő paraméterek: preferált erőforrások,

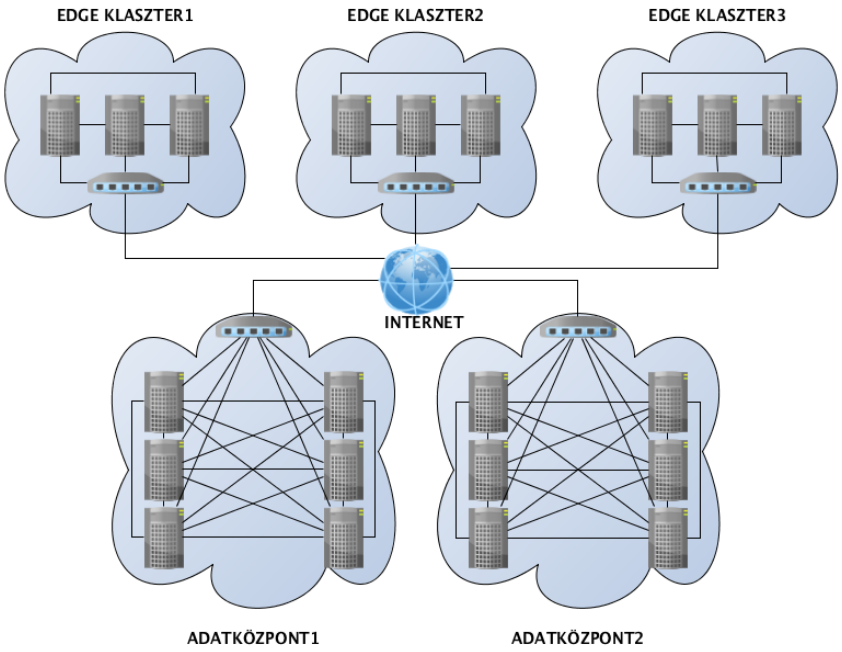
a konténerek száma erőforrásonként, lokalitás preferencia és prioritás. Mivel az erőforrás-menedzsernek globális rálátása van az elérhető erőforrásokra, válaszul megadja az alkalmazás mesternek azokat az erőforrásokat, ahol az egyes végrehajtók elhelyezhetők.

Az alkalmazás mester menedzseli az alkalmazás életciklusát is, beleértve a végrehajtók számának növelését, csökkentését is [31]. Három lokalitás preferencia szint van amit egy ütemező használni tud.

- Csomópont lokalitás: A legeffektívebb elhelyezés, amely során az egyes funkciókat azon a csomóponton indítja, amely a bemeneti adat egy blokkját tárolja. Nincs szükség távoli adat eléréséhez.
- Rack lokalitás: Ha a csomópont lokalitás nem lehetséges, mert egyik preferált csomópontnak sincs elérhető erőforrása, ezért az ütemező a funkciót egy olyan csomóponton helyezi el, ami egy közös rack-ben található valamelyik preferált szerverrel.
- Off-switch lokalitás: A legrosszabb eset. Ilyenkor nem található szabad erőforrás a megegyező rack-eken belül sem, tehát a funkciót valahova máshova kell elhelyezni.

5.2. A javasolt sávszélesség-tudatos erőforrás-vezénylő algoritmus

A modellemben a topológiát egy gráfként reprezentálom, ahol az egyes csomópontok a fizikai környezetből vett szerverek vagy átjárók, továbbá a topológia rendelkezik egy központi csomóponttal, amely az Internetet jelenti. A csomópontok klaszterekbe vannak csoportosítva; a szerverek továbbá rack-ekbe vannak szervezve a klasztereken belül. A csomópontokat irányítatlan élek kötik össze, melyek rendelkeznek egy pozitív egész kapacitással, amely a két csomópont között elérhető sávszélességet mutatja. Egy általános felépítést reprezentál a 5.2. ábra.



5.2. ábra. *Földrajzilag elosztott topológia a sávszélességet figyelembe vevő modellben*

A végrehajtók a szervereken vannak elhelyezve. Minden klaszternek van egy átjárója, amely az „INTERNET" csomópontba van bekötve. Ez a csomópont reprezentálja a kapcsolatot a klaszter és a külvilág között. Kettő típusú klasztert modelleztem. Az első a perem klaszter, amely kevés erőforrással rendelkezik és az Internet felé is alacsony elérhető sávszélessége van. A második típus az adatközpont, amely jóval nagyobb számítási erőforrással bír, továbbá az átjárója és az Internet között is nagyobb a sávszélesség mint a perem klaszter esetén.

A modellemben használt kérések megegyeznek a gyakorlatban is használt erőforrás kérésekkel. Egy alkalmazásnak van egy alkalmazás mestere, ami további végrehajtókat igényel a rendszertől. A végrehajtóknak egy bizonyos része megjelöl a futásukhoz preferált szervereket.

Ezen modell felhasználásával a célom a végrehajtók elhelyezése úgy, hogy az alkalmazásban szereplő köztes adatok áramlása a lehető legnagyobb sávszélességen történjen. A cél eléréséhez saját heurisztikus algoritmust készítettem, melyet 5.3 alfejezetben mutatok be.

A feladatom az egyes alkalmazások végrehajtóinak elhelyezése úgy, hogy a *map* funkcióktól a *reduce* funkciókat megvalósító végrehajtókhoz vezető úton a lehető legnagyobb legyen a minimális sávszélesség.

Legyen a topológia egy gráffal reprezentálva, amelyben a csúcsok megegyeznek a topológiában található szerverekkel, switchekkel és egyéb pontokkal. A csúcsokat összekötő élek pedig a topológiában megtalálható linkek, utak. Az élekhez továbbá hozzá van rendelve egy pozitív egész kapacitás, amely megmutatja az adott élen található aktuálisan elérhető sávszélességet.

Az egyszerű kettő termékes egészértékű folyam probléma egy NP-teljes probléma [27]. A

problémában kettő kiindulópontból (lehetnek azonosak is) egy-egy egységnyi egészértékű folyamot keresünk kettő nyelőbe (lehetnek azonosak is) úgy, hogy a gráf éleinek a kapacitása egy. Ez a saját problémámnak egy leegyszerűsített változata, amelyben a források a *map* funkciók helyei, a nyelők pedig a *reduce* funkciók. Minden *reduce* funkciónak létrehozunk egy új pontot a gráfban, amelyet összekötünk azokkal a pontokkal, ahová a *reduce* végrehajtó elhelyezhető. Ezek az élek szintén egy kapacitással rendelkeznek. Ez alapján belátható, hogy az én problémám is NP-teljes.

Az algoritmusom által megoldandó feladat formális leírásához szükséges jelöléseket a 4.1. táblázat tartalmazza. A probléma formalizálása során használt célfüggvény alább látható.

$$\max_{r \in V_r}(\min_{i \in V_m}(\min_{(u,v) \in E_t} y_{u,v}^{i,r} \beta_{u,v})) \tag{5.1}$$

A feladatra érvényesek a már korábban bemutatott feltételek (4.1, 4.2, 4.3).Az optimalizálás célfüggvénye az, hogy az összes *map* funkció fizikai helyétől az egyes *reduce* funkciókhoz vezető úton található minimális sávszélesség a lehető legnagyobb legyen (5.1).

5.3. A rendelkezésre álló sávszélességet növelő heurisztikus algoritmus

Az algoritmusom célja a végrehajtók elhelyezése úgy, hogy az elhelyezés során a hálózati sávszélesség maximális legyen a köztes adatfolyamhoz. Az algoritmus pszeudokódját a 4. kódrészlet mutatja.

Az algoritmus először az alkalmazás mestert helyezi el a topológiában. Ehhez mivel még nem tud semmit az alkalmazás igényeiről, véletlenszerűen választ egy szervert. Az alkalmazás mester elhelyezése után a végrehajtók következnek. Először azokat a végrehajtókat rakja le az algoritmus, amelyek rendelkeznek preferált szerverekkel. Az ilyen végrehajtók elhelyezése a következők szerint történik.

- A végrehajtók által definiált preferált szerverek valamelyikén helyezem el, amelyik rendelkezik szabad számítási erőforrással.
- A preferált szerverekkel megegyező a rackekbe helyezem el egy másik, szabad erőforrással rendelkező szerveren.
- Minden klaszterből a legkevésbé terhelt szerverhez kiszámítom a minimális sávszélességet a preferált szerverektől és azt a szervert választom, amelyikhez a legnagyobb a kiszámított minimális sávszélesség.

Algorithm 4 Elérhető sávszélességet figyelembe vevő végrehajtók elhelyezésére szolgáló algoritmus

```
1:  request.application_master.place_to(random_server)
2:  for each executor ∈ request.executors do
3:      if executor has locality_constraint then
4:          for each constraint_server ∈ locality_constraint do
5:              if constraint_server has available resource then
6:                  executor.place_to(constraint_server)
7:                  break
8:          if executor not placed then
9:              for each constraint_server ∈ locality_constraint do
10:                 if constraint_server.rack has available resource then
11:                     executor.place_to(least_utilized_server_from_constraint_rack)
12:                     break
13:         if executor not placed then
14:             max_bw ← 0
15:             available_servers ←least utilized available server from each cluster
16:             for each server ∈ available_servers do
17:                 for each constraint_server ∈ locality_constraint do
18:                     bw ← get_min_bw_between_nodes(server, constraint_server)
19:                     if bw > max_bw then
20:                         max_bw ← bw
21:                         chosen_server ← server
22:             executor.place_to(chosen_server)
23:     else
24:         max_bw ← 0
25:         available_servers ←least utilized available servers from each cluster
26:         for each server ∈ available_servers do
27:             bw ← get_min_bw_between_nodes(server, executor_places_with_locality_constraint)
28:             if bw > max_bw then
29:                 max_bw ← bw
30:                 chosen_server ← server
31:             executor.place_to(chosen_server)
```

A következő lépésként az olyan végrehajtók lerakása következik, melyek nem rendelkeznek helyi megkötésekkel. A lerakásuk során az algoritmus maximalizálni próbálja a sávszélességet a megkötéssel rendelkező végrehajtók és a többi végrehajtó között. Ehhez első lépésként minden klaszterből a legkevésbé terhelt szervereket keresem meg, majd ezen a szervereken végigmenve kiszámítom a minimális sávszélesség értékét az aktuális szerver és azon szerverek között, ahova a megkötéssel rendelkező végrehajtók lettek leképezve. Majd a legnagyobb minimális sávszélességgel rendelkező szervert választom ki a végrehajtó leképzési helyéül.

Az algoritmusom inicializáló fázisában meghatározom az egyes csomópontok között található sávszélességeket. Ezen művelet komplexitása n darab csomópont esetén $\mathcal{O}(n^2)$. Ugyanezeket a kéréseket az inicializálás után csak akkor kell újrafuttatni, ha a topológiában változás történik. A végrehajtók elhelyezése során egy szerver vagy rack megtalálása konstans idő alatt lehetséges. A legkevésbé terhelt szerverekből egy lista készítésének lépésszáma: $\mathcal{O}(n)$. A lista hossza megegyezik a klaszterek számával, amit jelöljünk c-vel.

Legyen k az egy *map* funkciót megvalósító végrehajtóhoz tartozó preferált szerverek száma. Két szerver között a legrövidebb út megtalálásához Dijkstra algoritmusát használva a komplexitás: $\mathcal{O}(e + nlogn)$, ahol e a topológia éleinek számát n pedig a csomópontok számát jelenti. Egy *map* feladatot megvalósító végrehajtó elhelyezésének lépésszáma egy olyan szerveren, amely nem szerepel a preferált szerverek által meghatározott rack-ekben felülről becsülhető $\mathcal{O}(n + ck(e + nlogn)))$. Legyen t az alkalmazásban szereplő végrehajtók

száma, így a végrehajtókat elhelyező algoritmusom komplexitása felülről becsülhető: $\mathcal{O}(n^2 + t(n + ck(e + nlogn)))$.

5.4. Szimulációs beállítások és eredmények

Ahhoz, hogy az algoritmusomat összehasonlítsam a YARN által használt algoritmussal egy nagy méretű földrajzilag elosztott topológiában, saját szimulátort fejlesztettem. A szimulátoromban az 5.2 alfejezetben bemutatott modellt használtam mind az alkalmazások, mind pedig a topológia szimulálására. A szimulációk során ugyanazon a topológián futtattam az algoritmusokat. A topológia ötven perem klasztert és öt adatközpontot tartalmazott. A szimulációk során használt sávszélesség értékeket a 5.1 mutatja. A táblázatban definiált sávszélesség értékek definiálásakor törekedtem a mai technológiák által elérhető az iparban is megtalálható értékekhez való közelítésre.

kiinduló csomópont típus-cél csomópont típus	sávszélesség
szerver-szerver (egy racken belül adatközpontban)	100Gb/s
szerver-szerver (egy racken belül perem klaszterben)	50Gb/s
rack-rack (egy klaszteren belül)	10Gb/s
szerver-átjáró (egy klaszteren belül)	10Gb/s
átjáró-Internet (adatközpont)	5Gb/s
átjáró-Internet (perem klaszter)	1Gb/s

5.1. táblázat. *Szimuláció során használt sávszélességek*

Két szimulációs példányt a használt kérések típusai, azon belül is a lokalitás kényszereknek a száma és a kiválasztása különböztet meg. Ezek alapján a következő szimulációs szcenáriókkal vizsgáltam algoritmusokat:

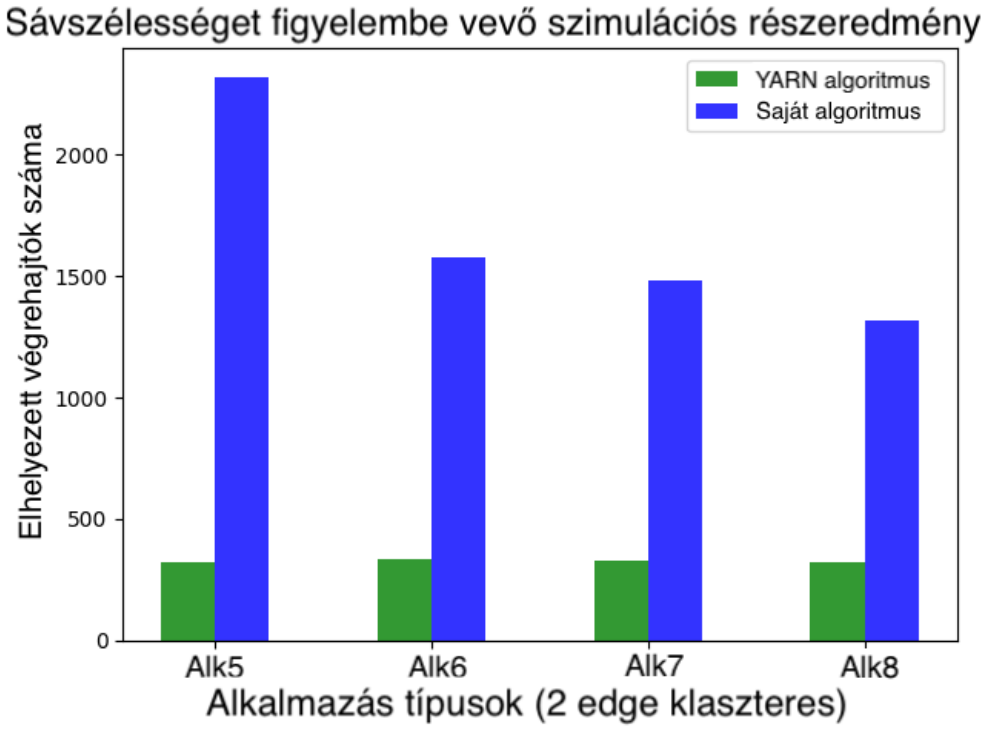
- Alkalmazás1: Az egyes végrehajtókhoz tartozó kényszerek a perem klaszterekből kerülnek kiválasztásra véletlenszerűen.
- Alkalmazás2: Az egyes végrehajtókhoz tartozó kényszerek az adatközpontokból kerülnek kiválasztásra véletlenszerűen.
- Alkalmazás3: Az egyes végrehajtókhoz tartozó kényszerek az egész topológiából kerülnek kiválasztásra véletlenszerűen.
- Alkalmazás4:Az egyes végrehajtókhoz tartozó kényszerek egy meghatározott perem klaszter szerverei közül kerülnek kiválasztásra véletlenszerűen.
- Alkalmazás5: Az egyes végrehajtókhoz tartozó kényszerek kettő meghatározott perem klaszter szerverei közül kerülnek kiválasztásra véletlenszerűen.
- Alkalmazás6: Az egyes végrehajtókhoz tartozó kényszerek három meghatározott perem klaszter szerverei közül kerülnek kiválasztásra véletlenszerűen.
- Alkalmazás7: Az egyes végrehajtókhoz tartozó kényszerek négy meghatározott perem klaszter szerverei közül kerülnek kiválasztásra véletlenszerűen.

- Alkalmazás8: Az egyes végrehajtókhoz tartozó kényszerek öt meghatározott perem klaszter szerverei közül kerülnek kiválasztásra véletlenszerűen.

Az egyes alkalmazás típusoknál változtattam továbbá a kényszerekkel rendelkező végrehajtók számát is, tehát minden alkalmazás típust szimuláltam úgy, hogy a bennük található *map*-szerű végrehajtók száma egytől nyolcig lehetséges egy alkalmazás példányban. A szimulációk során az egyes alkalmazások által igényelt erőforrásokat egyszerre foglalom le. Az egyes alkalmazások elhelyezését követően amennyiben a WAN (Wide Area Network) hálózaton ment át folyam egy *map* és egy *reduce* végrehajtó között, úgy az adott klaszterekhez tartozó WAN él elérhető sávszélességét csökkentettem 10 Mb/s-os értékkel. Az egyes szimulációk addig tartottak, amíg az összes számítási erőforrás vagy egy klaszterhez vezető rendelkezésre álló sávszélesség el nem fogyott.

A szimulációk lefutása után a két legfontosabb információ az elhelyezett végrehajtók száma, valamint az egyes kérések által keletkezett klaszterek közötti folyamok száma. Minden szimulációt tízszer futtattam. Az egyes szimulációk esetében a korábban felsorolt alkalmazás típusokat használtam úgy, hogy minden típus esetén a nyolc különböző számú *map* funkciós változatot is megvizsgáltam. A saját algoritmusom által elhelyezett kérések kevésbé használták a WAN hálózatot a köztes adatok továbbítására, így nem meglepő, hogy több alkalmazás példányt és több végrehajtót tudtak elhelyezni a topológiában, mint a YARN algoritmusa.

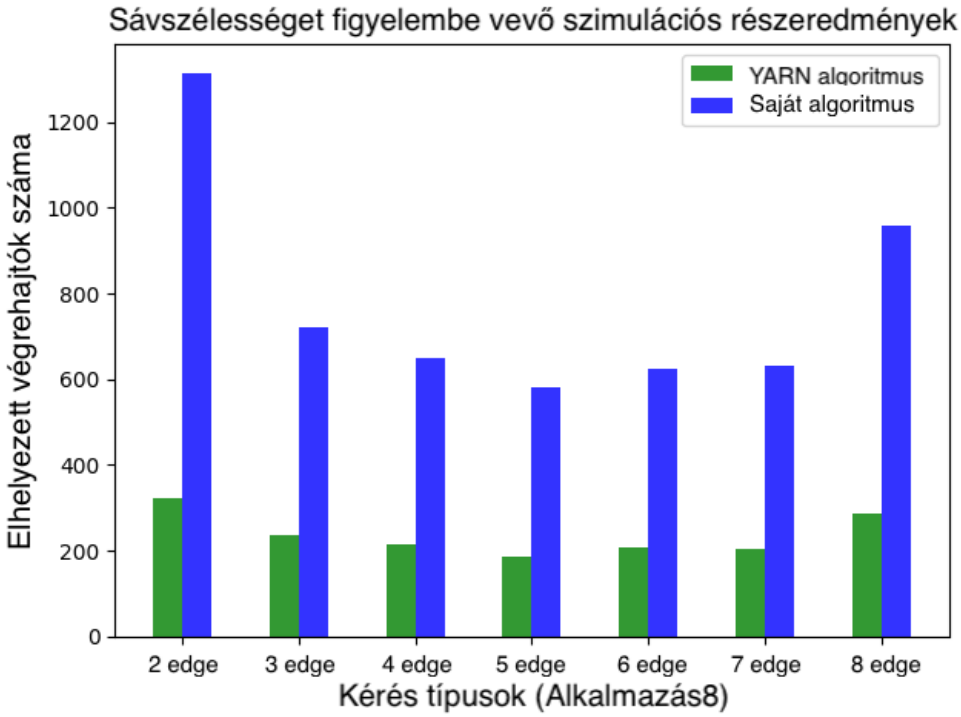
A 5.3. grafikonon a szimulációs eredmények egy része látható. Az egyes oszlopcsoportok a szimulációs szcenáriók eredményeit mutatják, melyekben minden alkalmazásban kettő végrehajtó van, aminek volt definiált helyi megkötése. Az egyes oszlopcsoportok által futtatott szcenáriók balról-jobbra a következők: Alkalmazás5, Alkalmazás6, Alkalmazás7, Alkalmazás8. Az y-tengelyen a topológiába elhelyezett végrehajtók száma látható.



5.3. ábra. *Sávszélességet figyelembe vevő szimulációs részeredmények kettő map funkció esetén*

Jól megfigyelhető, hogy a választható perem klaszterek számának növekedésével csökken a lerakott végrehajtók száma is. Ez könnyen magyarázható azzal, hogy minél több klaszterben helyezzük el a *map* funkcióinkat, annál több folyam fog átmenni a WAN hálózaton a *map* és *reduce* funkciók között. Látható az 5.3. grafikonon, hogy míg a YARN algoritmusa csupán körülbelül 300 végrehajtót volt képes sikeresen elhelyezni, addig az én megoldásom a kiválasztott alkalmazás típusok esetén legjobb esetben ennek több mint hétszeresét, körülbelül 2200 végrehajtót helyezett el. Továbbá az is megfigyelhető, hogy a 5.3. grafikon által mutatott eredményekben a legrosszabb esetben is körülbelül négyszer annyi alkalmazást tudott elhelyezni az én algoritmusom, mint a YARN algoritmusa.

Az 5.4. grafikonon szintén a YARN és a saját algoritmusom teljesítményét hasonlítom össze az elhelyezett végrehajtók által felhasznált számítási erőforrások függvényében. Az egyes oszlopcsoportok szintén a szimulációs szcenáriók eredményeit mutatják, melyeknél minden alkalmazásban az egyes végrehajtókhoz tartozó kényszerek öt meghatározott perem klaszter szerverei közül választódnak véletlenszerűen (Alkalmazás8). Az egyes csoportok a bennük található *map* funkciók számában tér el (kettőtől nyolcig). Az 5.4. grafikon y-tengelyén az elhelyezett végrehajtók száma látható.



5.4. ábra. *Sávszélességet figyelembe vevő szimulációs részeredmények Alkalmazás8 típusú kérések esetén*

Az 5.4. grafikonon látható, hogy az általam elkészített algoritmus az Alkalmazás5 esetében mindig jobb megoldást nyújtott mint a YARN hagyományos algoritmusa. Látható az ábrán, hogy a *map* funkciók számának növelésével egy bizonyos értékig romlik az algoritmusok teljesítménye, majd aztán megint javulni kezd. Ez elvárt működés. Legyen összesen tíz végrehajtónk, amiben x darab *map* és y darab *reduce* funkció van ($x + y = 10$). Tegyük fel, hogy egyik *reduce* funkció sem került egy klaszterbe egy *map* funkcióval. Ekkor minden *map* és minden *reduce* funkció között a folyam átmegy a WAN hálózaton, vagyis mind a két klaszter kimenő/bemenő élén csökken az elérhető sávszélesség. Mivel minden *map* funkció küld adatot minden *reduce* funkciónak, a felhasznált WAN folyamok száma: $fn = 2xy$. Ez alapján a 5.2 táblázat megmutatja, az egy alkalmazás által forgalmazott maximális folyam mennyiséget a *map* funkciók és *reduce* funkciók számának függvényében.

x	y	fn
2	8	32
3	7	42
4	6	48
5	5	50
6	4	48
7	3	42
8	2	32

5.2. táblázat. *Maximális WAN-on áthaladó folyamok száma*

A 5.2 táblázatból és a 5.4 grafikonból egyaránt látható, hogy az én esetemben a WAN hálózaton áthaladó folyamok száma öt *map* funkcióig növekszik, amiből következik, hogy

csökken a lerakható alkalmazások száma. Szintén megfigyelhető az ábrán, hogy az elért eredményeim a YARN eredményeihez képest körülbelül háromszor-négyszer jobbak, vagyis az én algoritmusom háromszor-négyszer több végrehajtót tudott elhelyezni a topológiában, mint a YARN algoritmusa.

6. fejezet

Összefoglalás

A munkám során megismerkedtem a Big Data fogalmával, illetve a Big Data rendszerek ökoszisztémájával. Megvizsgáltam az ökoszisztémában található egyes logikai rétegeket, és a bennük található komponenseket, szoftvereket. Különös figyelmet fordítottam a Big Data erőforrásokat vezénylő rétegére és a benne található algoritmusokra, megoldásokra. A jelenlegi megoldásokat összehasonlítottam egymással, majd mélyebben megvizsgáltam az általam választott erőforrás-vezénylő technológiát, a YARN ütemezőt.

A jelenlegi Big Data megoldásokat megvizsgáltam földrajzilag elosztott, több klasztert tartalmazó topológiában hálózati működés szempontjából. Megismertem a gyakorlatban elterjedt megoldás gyengeségeit az általam vizsgált újszerű környezetben. Mások eredményeiknek megismerése után azonosítottam a felmerülő problémákat, amelyek illeszkednek mind a Big Data rendszerek felhasználhatóságára, mind pedig az általam elképzelt földrajzilag elosztott topológiákra.

Látható, hogy a gyakorlatban használt Big Data technológiák jól működnek egy klaszteres környezetben, azonban működésük közel sem optimális több klaszteres környezetben. A dolgozatban a három probléma megfogalmazása során igyekeztem lefedni a hálózati teljesítmény legfontosabb kérdéseit. Ezzel a céllal fogalmaztam meg egy hálózati megbízhatósággal, egy késleltetéssel, és egy sávszélesség kihasználtsággal kapcsolatos problémát.

A Hadoopban a HDFS szolgál az adatok elosztott tárolására úgy, hogy nem veszi figyelembe a hálózat felépítését. Ez egy adatközpontos topológia esetén nem jelent hátrányt, azonban több klaszter esetén lehetnek problémák a működésben. A dolgozatomban ezeket a hiányosságokat vizsgálom a megbízhatóság szempontjából. Megfogalmazom és formalizálom a problémát, amely a hálózati megbízhatósággal függ össze. Megmutatom, hogy ez a probléma NP-nehéz, és egy heurisztikus algoritmust nyújtok, mely megoldja a feladatot és polinomiális időben működik. Az algoritmusomat a HDFS algoritmusával szimulációk segítségével összehasonlítom. Az eredmények alapján látható, hogy az általam nyújtott blokkelhelyező algoritmus az elérhetőség szempontjából körülbelül hatszor jobban teljesít, mint a HDFS algoritmusa.

A SPARK az egyik legelterjedtebb adatfeldolgozó technológiája a Big Data rendszereknek. Egy klaszterben meglehetősen hatékonyan működik, azonban több klaszter esetén amikor nagy mértékű késleltetés található a rendszerben a SPARK teljesítménye romlik.

Ennek kiküszöbölésére hoztam létre egy algoritmust, amely polinomiális idő alatt megtalálja azon szervereket a kiterjedt topológiában, ahová a *reduce* feladatokat elhelyezve a *map* funkcióktól vett késleltetés értéke lényegesen kevesebb lesz, mint a SPARK alapértelmezett elhelyező algoritmusa esetében.

A YARN a Hadoop ökoszisztéma egyik széles körben használt erőforrás-vezénylő technológiája. Működése során azonban ez a technológia nem veszi figyelembe a hálózati erőforrásokat, mint például a sávszélességet. Szimulációk során bemutatom, hogy az általam elképzelt topológiában a sávszélesség figyelembe vétele az erőforrások ütemezése során igenis jelentős tényező. A saját algoritmusom polinomiális idő alatt határozza meg az egyes alkalmazások által kért végrehajtók helyét úgy, hogy a WAN hálózatot kevésbé használja, mint a YARN algoritmusa. A szimulációk során látható, hogy az általam ajánlott algoritmus teljesítménye akár ötszöröse is lehet a YARN algoritmusának.

A dolgozatom során három nehéz problémára kínálok megoldást. A problémák technológiai szempontból relevánsak, hiszen olyan szolgáltatások nyújtása kapcsán merülnek fel, melyek a legújabb, legelterjedtebb megoldásokat felhasználva működnek. Létrehoztam három olyan polinomiális lépésszámú algoritmust, mely a fentebb bevezetett problémákra megoldást nyújtanak és jobb teljesítményt érnek el, mint a jelenleg elterjedt megoldások. A szimulációk során megállapítottam, hogy a megoldásaim jól skálázódnak, hiszen mind nagy topológián, mind sok kiszolgált alkalmazás után is megtartják a gyors működésüket és helyes eredményeket.

Köszönetnyilvánítás

Köszönöm mindenekelőtt konzulensemnek Dr. Toka Lászlónak, hogy konzultációs tevékenységével folyamatosan irányt mutatott munkám során és beszélgetéseinkkel hozzájárult a témában való szemléletmódom formálásához. Továbbá köszönettel és hálával tartozok Vass Balázsnak, aki segítségével hozzájárult a dolgozatom sikeres elkészüléséhez. Szintén köszönettel tartozok a BME TMIT tanszékének, hogy rendelkezésemre bocsájtotta a munkámhoz szükséges eszközöket és infrastruktúrát.

Irodalomjegyzék

[1] Sachin P Bappalige. An introduction to apache hadoop for big data. `https://opensource.com/life/14/8/intro-apache-hadoop-big-data`, May 2018.

[2] Shubham Sinha. Apache hadoop ecosystem. `https://www.quora.com/What-is-a-Hadoop-ecosystem`, May 2018.

[3] Apache. Apache hadoop distributed file system. `http://hadoop.apache.org/docs/stable/hadoop-project-dist/hadoop-hdfs/HdfsDesign.html`, May 2018.

[4] Tom White. *Hadoop: The Definitive Guide, Fourth Edition.* O'Reilly Media, Inc., 2015.

[5] Apache. Apache spark architecture. `https://spark.apache.org/docs/latest/cluster-overview.html`, May 2018.

[6] Apache. Apache yarn architecture. `https://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html`, May 2018.

[7] Charles J. Colbourn. *The Combinatorics of Network Reliability.* Oxford University Press, Inc., 1987.

[8] Ilya B. Gertsbakh, Yoseph Shpungin. *Models of Network Reliability: Analysis, Combinatorics, and Monte Carlo.* Boca Raton: CRC Press., 2012.

[9] János Tapolcai, Balázs Vass, Zalán Heszberger, József Bíró, David Hay, Fernando A. Kuipers, Lajos Rónyai. A tractable stochastic model of correlated link failures caused by disasters. *IEEE INFOCOM*, 2018.

[10] Justin Meza, Tianyin Xu, Kaushik Veeraraghavan, Onur Mutlu. A large scale study of data center network reliability. *ACM IMC*, 2018.

[11] Andreas Fischer, Juan Felipe Botero, Michael Till Beck, Hermann de Meer, Xavier Hesselbach. Virtual network embedding: A survey. *IEEE Communications Surveys and Tutorials*, 2013.

[12] Minlan Yu, Yung Yi, Jennifer Rexford, Mung Chiang. Rethinking virtual network embedding: substrate support for path splitting and migration. *ACM SIGCOMM*, 2008.

[13] N. M. M. K. Chowdhury, M. R. Rahman, R. Boutaba. Virtual network embedding with coordinated node and link mapping. *IEEE INFOCOM*, 2009.

[14] David Haja, Marton Szabo, Mark Szalay, Adam Nagy, Andras Kern, Laszlo Toka, Balazs Sonkoly. How to orchestrate a distributed openstack. *IEEE INFOCOM*, 2018.

[15] Balázs Németh, Márk Szalay, János Dóka, Matthias Rost, Stefan Schmid, László Toka, Balázs Sonkoly. Fast and efficient network service embedding method with adaptive offloading to the edge. *IEEE INFOCOM*, 2018.

[16] Marty Humphrey Arkaitz Ruiz-Alvarez. Toward optimal resource provisioning for cloud mapreduce and hybrid cloud applications. *IEEE/ACM BDCAT*, 2014.

[17] Marco Cavallo. H2f: a hierarchical hadoop framework to process big data in geo-distributed contexts, 2018.

[18] Benjamin Heintz, Abhishek Chandra, Ramesh K. Sitaraman, Jon Weissman. End-to-end optimization for geo-distributed mapreduce. *IEEE Transactions on Cloud Computing*, 2014.

[19] Qi Zhang, Ling Liu, Kisung Lee, Yang Zhou, Aameek Singh, Nagapramod Mandagere, Sandeep Gopisetty, Gabriel Alatorre. Improving hadoop service provisioning in a geographically distributed cloud. *IEEE CLOUD*, 2014.

[20] Guohui Wang, T. S. Eugene Ng, Anees Shaikh. Programming your network at run-time for big data applications. *ACM SIGCOMM*, 2012.

[21] Peng Qin, Bin Dai, Benxiong Huang, Guan Xu. Bandwidth-aware scheduling with sdn in hadoop: A new trend for big data. *IEEE Systems Journal*, 2015.

[22] Qifan Pu, Ganesh Ananthanarayanan, Peter Bodik, Srikanth Kandula, Aditya Akella, Paramvir Bahl, Ion Stoica. Low latency geo-distributed data analytics. *ACM SIGCOMM*, 2015.

[23] Ge Zhang, Haozhan Wang, Zhongzhi Luan, Weiguo Wu, Depei Qian. Improving performance for geo-distributed data process in wide -area. *IEEE CIT*, 2017.

[24] Zhiming Hu, Baochun Li, Jun Luo. Flutter: Scheduling tasks closer to data across geo-distributed datacenters. *IEEE INFOCOM*, 2016.

[25] Bin Cheng, Apostolos Papageorgiou, Flavio Cirillo, Ernoe Kovacs. Geelytics: Geo-distributed edge analytics for large scale iot systems based on dynamic topology. *IEEE WF-IoT*, 2015.

[26] Lin Gu, Deze Zeng, Song Guo, Yong Xiang, Jiankun Hu. A general communication cost optimization framework for big data stream processing in geo-distributed data centers. *IEEE Transactions on Computers*, 2016.

[27] S. Even, A. Itai, A. Shamir. On the complexity of tmetable and multi-commodity flow problems. *SIAM Journal on Computing*, 1976.

[28] Charles Clos. A study of non-blocking switching networks. *IEEE The Bell System Technical Journal*, 1953.

[29] Mohammad Al-Fares, Alexander Loukissas, Amin Vahdat. A scalable, commodity data center network architecture. *ACM SIGCOMM*, 2008.

[30] Michael R. Garey, David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman and Company, 1979.

[31] Mohammad Hammoud, Majd F. Sakr. Locality-aware reduce task scheduling for mapreduce. *IEEE CLOUDCOM*, 2011.