# Muppet: Adatfolyamok közel valós idejű feldolgozása egy a MapReduce-hoz hasonló módszerrel

Müller Dávid (DFL93A)
Torma Balázs (KPLHJ5)
Tornóczky Zoltán (E7RZGC)

Ismertető Wang Lam, Lu Liu, STS Prasad, Anand Rajaraman, Zoheb Vacheri és AnHai Doan
Muppet: MapReduce-Style Processing of Fast Data c. cikkéhez

## Rövid összefoglalás

Ezen tanulmány célja ismertetni a Wang Lam, Lu Liu, STS Prasad, Anand Rajaraman, Zoheb Vacheri és AnHai Doan által írt Muppet: MapReduce-Style Processing of Fast Data c. cikkben leírtakat.

A MapReduce modell egy ma már népszerű módja nagy mennyiségű adatok feldolgozásának. A szociális hálók széles körű elterjedésével viszont további követelmények léptek fel, amelyeknek a MapReduce modell már nem képes megfelelni. Ez a modell ugyanis eredetileg nagy méretű adatbázisokon végzett műveletekre lett tervezve, melyek tartalma a feldolgozás közben nem változik. Például a Twitter üzenetek nyomon követésekor a folyamatosan beérkező tweeteket is időszerűen kell tudni kezelni. Ezért itt már nem pusztán adathalmazokról, hanem adatfolyamokról beszélhetünk, amiket valós időben, illetve minimális késleltetéssel kell feldolgozni.

Az ismertetendő cikk szerzői a Kosmix és a WalmartLabsnél dolgozva a MapReduce-ból kiindulva megalkották az ún. MapUpdate keretrendszert ami képes ezen probléma megoldására. A szerzők készítettek is egy implementációját a modelljüknek, a Muppetet, amit a gyakorlatban kipróbálva sikeresen tudtak szociális hálókkal és e-kereskedelemmel kapcsolatos adatfolyamokon (megfelelő válaszidő mellett) műveleteket végezni.

## Bevezetés

A MapReduce [1] modell rendkívüli módon leegyszerűsíti a nagy adatmennyiségek feldolgozására képes alkalmazások fejlesztését, ezért az utóbbi években egyre nagyobb népszerűségnek örvend. Ezt a módszert használva a fejlesztőnek nincsen más dolga, mint írnia néhány mapper és redukáló függvényt, majd a rendszer automatikusan beállítja a feldolgozásra szánt gépek klaszterét. Ezen túl gondoskodik a klaszterben található számítógépek közötti egyenletes teherelosztásról, felügyeli a végrehajtást, és kezeli a meghibásodásokat.

A map és reduce függvények a funkcionális programozási paradigmából ismertek. A map függvény az adatok egy listáján végrehajt egy átalakító műveletet, amit szintén paraméterként kap meg. A visszatérési értéke egy másik lista lesz. Ezzel szemben a reduce függvény összegzi a bemeneti lista elemeit, vagyis redukálja azt egy adott művelet szerint. Értelemszerűen a lista redukciójának végeredménye nem egy átalakított lista, hanem a listából kinyert adat. Például egész számok egy véges listájának elemeit ha összeadjuk, azt szintén a lista redukálásnak lehet tekinteni,

és a végeredmény egyetlen szám lesz. A MapReduce modellben ehhez hasonlóan viselkedik a map és reduce művelet. Látható, hogy ennek a kétféle műveletnek a kombinálásával lefedhető az összes jellemző adatbázisokon elvégzendő átalakítás, információ lekérdezés stb.

Viszont vannak olyan adatfajták, amelyekre a MapReduce modellben található megközelítés nem felel meg, és ezek közül kitűnnek a szociális hálókban található, folyamatosan változó és bővülő adattömegek. Ezek nem ábrázolhatóak elemek véges listájaként, ezért nem is lehet őket véges idő alatt redukálni. Ebben az esetben pillanatfelvételt kellene készíteni az adattömegről, mielőtt azt redukálással lehetne feldolgozni. Már emiatt is szükség van egy új keretrendszer kidolgozására, mely képes kezelni az ilyen típusú alkalmazásokat. Emellett listák helyett adatfolyamokkal kell ábrázolni az említett adattömegeket és azok időbeli alakulását.

Az elmúlt évtizedben az olyan szociális háló szolgáltatások elterjedésével, mint a Facebook, Twitter, iWiW, vagy a Google+, nagyon fontos lett a bennük rejlő információk hatékony és gyors kezelése. Ezeken a weblapokon történő események (ismerős bejelölése, üzenőfalra kiírás, üzenet írása stb.) szintén egy az idő függvényében bővülő folyamot alkotnak, amikre szükséges a rendszereknek másodperceken belül, vagyis szinte valós időben reagálniuk. A cikk írói az ilyen típusú adatokat nevezik „gyors adatnak”. Nem szabad elfelejteni, hogy ebben az esetben a rövid késleltetés mellett nagyon fontos a rendszer skálázhatósága is, mivel az előbb említett közösségi weblapok forgalma egyáltalán nem egyenletes. Tehát a rendszernek képesnek kell lenni megbirkózni a csúcsforgalom okozta többlet terheléssel. Hasonlóan 'gyors' adatnak tekinthető pl. egy érzékelő által generált mérési adatfolyam.

A MapUpdate modell az eddig említett kihívásoknak próbál maradéktalanul megfelelni. Redukálás helyett az adatfolyamokon végzett update műveleteket végezhetünk el, ami annyiban különbözik a reduce-tól, hogy működés közben lekérdezhető az összegző művelet addig akkumulált részeredménye. Ezeket az összegyűjtött adatokat az update művelet az ún. slate-en (lapon) tárolja. Az update műveletnek tehát nem is kell leállnia, mint ahogy az adatfolyam sem biztos hogy véget ér, és folyamatosan tudja fogadni az újabb beérkező eseményeket. A map művelet működése hasonlóan átalakítható folyamokra, vagyis nem véges listákra.

## Konkrét példa

Adott egy alkalmazás ami a Twitter Firehose-t elemzi és kiválogatja a legtöbbet említett témákat. A Firehose [2] magában foglalja a Twitter rendszerében megjelenő napi átlag 50 millió üzenetet [3] és egy hatalmas adatfolyamot alkot. Az alkalmazás minden felbukkanó témának fenntart egy számlálót, majd adott időközönként (pl. percenként) megfigyeli, hogy hány üzenet vág bele egy-egy témába.

Egyúttal átlagolja is naponta az előbb említett értékeket témánként. Minden témánál ezeket a napi átlagokat több napra vetítve is átlagolja az alkalmazás. Ha ezt az összesített átlagot a percenkénti témába vágó üzenetszámláló egy bizonyos mértékben túllépi (küszöbérték eleve adott), akkor azt kigyűjti az alkalmazás és eltárolja a <téma, perc> párt. Ez a pár azt jelöli, hogy melyik téma a nap melyik percében lett az átlagosnál többször említve.

Ezt az alkalmazást a MapReduce keretrendszer segítségével több okból sem célszerű megírni. Mivel az adatfolyam kb. másodpercenként 580 üzenettel bővül folyamatosan, ezért nagyon gyakran kellene pillanatfelvételt készíteni a teljes adatmennyiségről, hogy viszonylag gyorsan észre lehessen venni, ha egy téma népszerűvé válik. A pillanatfelvételek készítése elengedhetetlen az alkalmazásnak, hogy legyen mindig egy véges adatszerkezet amin a map és reduce műveletek végig tudjanak iterálni. Minden egyes pillanatfelvételnél újra és újra fel kell dolgozni olyan adatokat, amik az előző felvételben már benne voltak, ezért fölöslegesen dolgozik ilyenkor az alkalmazás. Ráadásul ha több hónapig, vagy akár évig akarjuk a MapReduce keretrendszerben írt alkalmazásunkkal figyelni a népszerű témákat, akkor elég hamar kezelhetetlen méretűre duzzadnak ezek a pillanatfelvételek. Ekkor már elfogadhatatlanul sokáig tart egyet is végig fésülni, így a rendszer késleltetési ideje a végtelenségig nőhet. Hirtelen forgalomnövekedéskor a helyzet tovább romlik, ezért az alkalmazásunk skálázhatóságáról sem lehet beszélni.

A végeredmény valójában az lesz, hogy már az első nap folyamán kifogyunk a tárhelyből, és

meg kell szakítani a működést. De amíg az alkalmazás működött is, csak nagyon lassan kaptunk vissza pillanatnyi információkat a Firehose-ról.

Ezzel szemben, ha a MapUpdate keretrendszert használtuk volna fel, akkor az update műveleteknek és a módosult mappereknek köszönhetően mindig csak az újonnan beérkezett üzeneteket kell feldolgoznia az alkalmazásunknak, és a memória és számítási idő pazarló pillanatfelvételekre sincsen szükség, mivel az update függvények eleve feljegyzik az addig összegyűjtött részeredményeket a megfelelő lapokba.

Ekkor a rendszer reakció ideje állandóan rövid marad, és nem növekszik az idő múlásával. Sokkal inkább képes lesz elviselni a hirtelen forgalomnövekedéseket, és időszerűbben kaphatóak meg az adott perc népszerű témáinak listája.

Végül is egyértelmű hogy egy adatfolyamokon értelmezett keretrendszerre van szükség, aminek a következő követelményeknek kell megfelelnie:

- A modellje alapvetően egyszerű és átlátható legyen, könnyű legyen rajta alkalmazásokat programozni. Minél inkább hasonlítson használatában a MapReduce keretrendszerhez, hogy a fejlesztőknek könnyen menjen az átszokás.
- A keretrendszer jól tudja kezelni a dinamikus adatszerkezeteket, mint pl. az adatfolyamokat.
- Alacsony késleltetésű számítást tegyen lehetővé, ami közel valós idejű, és menet közben is le lehessen kérdezni az addig kiszámított részeredményt.
- Az elkészült alkalmazás jól skálázható legyen közönséges személyi számítógépekből felépített klasztereken is.

# Kapcsolódó munkák

A Muppet első változata még a Kosmix alatt lett bevezetve 2010 közepén. Azóta több fejlesztésen esett át és most már kiterjedten használják a WalmartLabsben. Kezdetben a Muppet a Twitter Firehose folyamának és a Foursquare bejelentkezési folyamának feldolgozására lett használva. 2011 elejére már több mint 100 millió Twitter üzenetet, és naponta több mint 1,5 millió Foursquare bejelentkezést dolgozott fel. Eközben több mint 30 millió Twitter felhasználóhoz, és 4 millió Foursquare találkozóhelyhez tartozó lapot tárolt. Több tíz személyi számítógépből álló klaszteren futott, ami képes volt kevesebb mint 2 másodperces átfutási idővel dolgozni. Muppet keretrendszer működtette a Kosmix legfőbb termékét, a TweetBeatet [5], és az utóbbi időben a WalmartLabsnél a ShopyCat [6] alkalmazás működtetéséért felel.

Körülbelül 16 különböző fejlesztőcég használta már 15 alkalmazás elkészítéséhez a Muppet keretrendszert, melyek közül többen visszatérő ügyfélnek számítanak. 2012 júniusáig a Cassandra klaszter ezeket az alkalmazásokat kiszolgálva több mint 2 milliárd lapot tartott fenn.

# Alapfogalmak

## Platform

A MapUpdate keretrendszer hardveres követelményei a MapReduce-éhoz nagyban hasonlítanak, kisebb különbségekkel. A klasztert szintén elő lehet olcsón állítani boltban megvásárolható személyi számítógépek hálózatba kötésével. Viszont, a MapUpdate-nél a klasztert alkotó gépeknél a hangsúlyt a központi memória méretére, ill. teljesítményére kell helyezni, nem pedig a háttértáréra. Ez annak tudható be, hogy valós idejű számításokat szeretnénk végezni, ahol az update függvények által használt lapok miatt nem kell egyszerre olyan sok adatot tárolni a háttértárban. Az elérési idő további csökkentése érdekében forgótányéros merevlemezek helyett SSD-ket érdemes beszerelni a klaszter tagjaiba.

## Események és folyamok

Az események nem mások, mint az adatfolyamok legkisebb egységei, ahol a folyamok események időben változó sorozatai. Ilyen lehetn pl. a Twitterben egy üzenet, vagy az iWiW-en egy ismerősnek jelölés. Az eseményeket egy rendezett 4-essel jelöljük: *<sid, ts, k, v>*

- *sid (Stream ID)*: Azon adatfolyam azonosítója, amelyhez a szóban forgó esemény tartozik.
- *ts (TimeStamp):* Az esemény beérkezésének időpontja.
- *k (Key)*: Olyan kulcs, ami nem az eseményt különbözteti meg. Ehelyett azonosíthatja pl. Twitterben egy üzenet esemény kiváltóját, vagyis azt a felhasználót aki küldte. A kulcsokkal tehát a teljes rendszeren belül, vagy akár csak egy adott folyamon belül csoportosíthatóak az események.
- *v (Value)*: Az esemény értéke, ami lehet tetszőleges típusú adat, pl. JSON objektum.

Ezentúl feltesszük, hogy a folyamban sorban következő események időpont szerinti növekvő sorrendbe vannak állítva, vagyis az újonnan beérkező események a folyam végére kerülnek, miközben az alkalmazás balról-jobbra dolgozza fel a folyamot.

## Map függvények

A mapperek elsődleges célja az adatfolyamok kezelése, összefésülése, átirányítása, illetve a teljes folyamra (annak minden eseményére) valamilyen művelet elvégzése. A bemenetén több folyam is szerepelhet, ugyanúgy mint a kimeneten. Egy mapper elemenként dolgozza fel a kapott eseményeket.

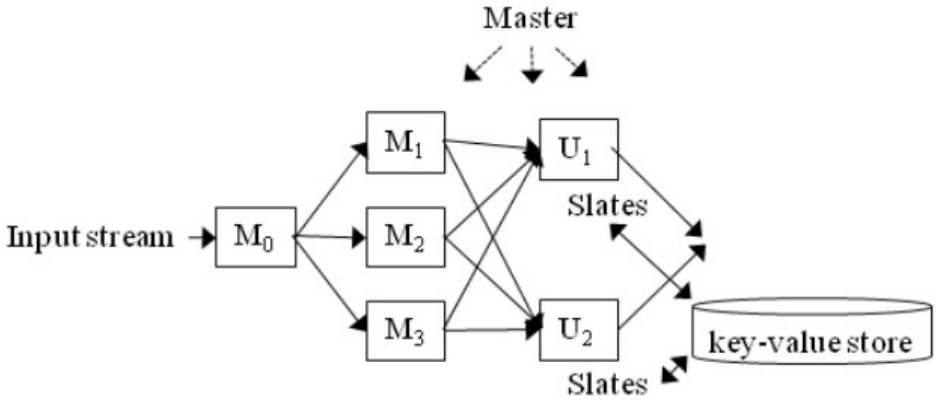
Ha egynél több folyamról kapja az inputot, akkor a bemenő folyamokról érkező eseményeket időpont szerinti növekvő sorrendben fűzi össze egy közös folyammá. Időpont egyezés esetén determinisztikusan dönt, hogy melyik esemény kapjon elsőbbséget. Eztán az összes átmenő eseményre elvégez egy adott műveletet (ami lehet akár identitás, vagy szűrő is), majd szükség szerint egy vagy több kimenő folyamba vezeti ki a feldolgozott eseményeket. A kimenő események időpont mezőjét a feldolgozás időpillanata szerint állítja át.

## Update függvények és lapjaik

Az update függvény hasonlóan viselkedik mint a map függvény: akár több bemenő folyamból több kimenő folyamot is tud eredményezni. A bemenő események összefűzésének módja megegyezik a mapnél leírtakkal. A mappel ellentétben viszont rendelkezik saját memóriával, ahol a működésének addigi részeredményeit képes eltárolni. Ennek a memóriának a részeit hívják az update függvény slate-jének, azaz lapjának. Egy updaterhez nem csak egyetlen lap tartozik, hanem eseménykulcs szerint akár több is. Vagyis egy lapot a hozzá tartozó update függvény nevével és eseménykulcsának párjával lehet egyértelműen meghatározni. Ezért egy lap $S_{U,k}$-val jelölendő, ahol U az update függvény, k a kulcs.

Az updater működése közben a lapokat valós időben, folyamatosan frissíti, ahol is az adott kulcshoz tartozó addigi részeredményt tárolja el. Ilyen lehet pl. egy Twitter felhasználó által írt üzenetek száma. Egy lap, csak lokális adatokat tartalmazhat, globális változókra nem hivatkozhat az ütközések elkerülése érdekében (így nem lesz szükség szinkronizációra). Nem kötelező egy lapot határozatlan ideig tárolni, különben könnyen telítődhetne a klaszter gépeinek központi memóriája. Erre szolgál a Time To Live (TTL) paraméter, ami meghatározza, hogy milyen sokáig maradhat a memóriában egy olyan lap amit nem írt felül a hozzá tartozó update függvény. Ennek ellenére van lehetőség a lapokat a háttértáron megőrizni az ún. kulcs-érték tárban. Ebből a tárból, de akár egy éppen használatban levő lapból is lekérdezhetőek a részeredmények bármikor a rendszer működése közben.

Fontos megjegyezni, hogy mind a map, mind az update függvények képesek akár teljesen új események létrehozására is.




1. ábra: MapUpdate alkalmazás

## MapUpdate alkalmazás

Egy teljes alkalmazás már összeállítható az ebben a fejezetben eddig tárgyalt elemekből: lényegében egy map és update függvényeket folyamokkal, mint élekkel összekötött irányított gráfról van szó, ahol az irányított körök megengedettek (ld. 1. ábra). Egy ilyen alkalmazás eredménye a kimenő folyamokból és az összes lap tartalmából olvasható ki. Az ábrán a „Master" egy hibakezelő programot jelöl, ami a MapUpdate modell konkrét implementációjának, a Muppetnek a leírásában lesz bővebben kifejtve.

A MapUpdate-et használó fejlesztőnek alkalmazás írásakor csak a map és update függvényeket kell implementálnia, és egy konfigurációs fájlt megírnia, ami meghatározza ezek összekötését.

# Eredmények

Ebben a fejezetben először a MapUpdate keretrendszer Muppet nevezetű implementációjának 1.0-ás verzióját, majd annak továbbfejlesztését írjuk le. Végül néhány, gyakorlatban is bizonyított MapUpdate alkalmazás teljesítményéről is szó lesz.

## Muppet 1.0

Az első változatot még a Kosmix berkein belül fejlesztették ki, mielőtt a WalMart felvásárolta volna a céget, ami után a 2.0-ás változat fejlesztése elindult.

## Elosztott végrehajtás

Az 1. ábra demonstrálja, hogyan épül fel egy Muppet alkalmazás. Az implementációban az update vagy map függvények különálló programokban futnak, amiket együttesen dolgozóknak (worker) nevezünk. Mindig kell lennie egy kezdő map függvénynek ($M_0$ az ábrán), ami elosztja a többi dolgozó között a bejövő adatfolyamot. Ez az elosztás egyetlen hasító függvénnyel megoldott, melyet a dolgozók egymás közt megosztanak.

A szóban forgó hasító függvény értelemszerűen az események kulcsai alapján adja meg, hogy melyik dolgozónak kell az eseményt a következő lépésben megkapnia. Ebből következik, hogy minden egyes dolgozó csak meghatározott kulccsal rendelkező eseményekkel foglalkozhat. Azt sem szabad elfelejteni, hogy nem feltétlenül működik minden összekötött dolgozó egyetlen gépen: a hasító függvény akár olyan dolgozónak is adhat egy eseményt, ami a klaszteren belül másik gépen fut. Minden dolgozó rendelkezik egy saját végrehajtási sorral (queue), amibe a frissen megkapott eseményt el tudja helyezni. A dolgozók ennek a sornak a végéről veszik le egyesével a feldolgozandó eseményeket. Egy dolgozónak magának kell gondoskodnia arról, hogy a feldolgozott eseményeket a következő dolgozó végrehajtási sorba helyezze, miután megtalálta azt a hasító függvény segítségével.

Ez a végrehajtás szöges ellentétben áll a MapReducer modellben alkalmazottal: ott egy vezérlőprogramnak szólnak a dolgozók (ott egy dolgozó nyilván map vagy reduce függvényt futtathat) ha az eredményüket tovább akarják küldeni, illetve a bemenetet is onnan várják. Vagyis a rendszer teljes vezérléséért felel ott az 1. ábrán „Master" névvel jelzett program. A Muppet rendszerben ugyanez a program csak a hibakezelésért felel. A Muppet-féle megközelítés nagyban rövidíti a rendszer válaszidejét azzal, hogy kiiktatja a dolgozók kommunikációjából a közvetítőket.

## Lapok kezelése

Mint ahogy az alapfogalmaknál említésre került, egy adott U update függvényhez és k kulcshoz egyértelműen tartozó lap elsősorban az U-t futtató gép központi memóriájában létezik. Ha egy ilyen lapban tárolt adatokra hosszútávon is szükség van (pl. ebből szeretnénk később kiolvasni az eredményt), akkor lehetőség van annak perzisztálására egy kulcs-érték tárban. Az ilyen lapok nem vesznek el a klaszter gépeinek meghibásodásakor, ezen kívül a RAM túlcsordulását is el lehet kerülni a kulcs-érték tárba helyezéssel.

A Muppet kulcs-érték tárnak a szintén saját fejlesztésű Cassandrát használja. A Cassandra szintén egy közönséges számítógépekből felépített klaszteren fut, melyek mindegyike futtatja a Cassandra program egy példányát. Ez a klaszter kívülről nézve egy teljesen egységes tárhelyet biztosít. A Cassandra kulcsok intervallumának (key space) egy halmazát tartja karban, melyek egyenként „oszlopcsaládok" (column family) halmazaiból áll. Egy ilyen oszlopcsalád nem más, mint <kulcs, oszlop> párok gyűjteménye, ahol a kulcs az eseményhez tartozik, az oszlop pedig egy adott updaterhez.

Az értékek ilyen szervezése azt szolgálja, hogy egy Cassandra klaszter egyszerre ne csak egy Muppet klasztert, vagy másmilyen alkalmazást tudjon csak kiszolgálni: minden alkalmazásnak megvan a saját kulcs intervalluma és azon belül az oszlopcsaládja. Tehát ha a Cassandrából le akarjuk kérdezni az $S_{u,k}$ lapot, akkor az alkalmazásunknak fenntartott oszlopcsaládon belül a k-adik sor U-adik oszlopában található a kívánt lap. A Cassandrából lekérdezés általában a TCP protokoll segítségével történik meg.

Végül egy Cassandrát használó Muppet klaszterben egy lap lekérdezése a következő lépések szerint történik:

- Először a gyorsítótárban, vagyis a Muppet klaszter központi memóriájában keresi a lapot.
- Ha ott nem találta, akkor próbálja a Cassandrából lekérni.
- Ha ez is sikertelen, akkor új lapot kell létrehozni.

A gyakorlatban a Cassandra többnyire tömörített JSON objektumok formájában tárolja a lapok tartalmát.

## SSD-k használata és lapok gyorsítótárazása

Tovább rövidíthető a Cassandrában tárolt lapok elérési ideje, ha a háttértároláshoz SSD-ket használunk: az SSD-kre jellemző, hogy akármelyik adatblokk ugyanannyi, milisecundumos nagyságrendű elérési idővel rendelkezik (random access). Így feleslegessé válik a Cassandra klaszter gépeinek központi memóriájában az olvasások gyorsítótárazása. Az ennek köszönhetően fel szabadult memóriaterületet teljes mértékben a lapok írásának gyorsítótárazására tudjuk szentelni. Innentől a lapok olvasását elég már a Muppet klaszter memóriájában cache-elni. Ez az elrendezés fontos előnyökkel jár:

- Az alkalmazás indulásakor amikor a Muppet cache-e teljesen üres, a legtöbb laplekérdezés a Cassandrához jut. Az SSD-k relatíve magas IOPS értékének és áteresztőképességének köszönhetően gyorsan és könnyedén megbirkózik az egyszerre beérkező sok laplekérdezéssel.
- Az SSD-k magas IOPS értéke és véletlen elérése lehetővé teszi, hogy a Cassandra képes legyen párhuzamosan eleget tenni a laplekérdezésnek és egyben periodikusan tömöríteni a JSON objektumokból álló laptárat.
- Mivel a legtöbb alkalmazás hajlamos néhány lapot nagyon sűrűn módosítani, jól jön az, hogy a Cassandrában a lapírások sokáig a központi memóriában vannak gyorsítótárazva. Így egy módosítás még gyorsabban tud megtörténni, és az SSD-k szektorainak korlátozottabb számú írási ciklusait sem használja el olyan gyorsan a tárhely (nő az SSD-k élettartama).

## Flush, Quorum, Time To Live

Alapvetően 3 paraméterrel lehet beállítani egy Muppet alkalmazás várható teljesítményét, robusztusságát és biztonságosságát:

A Flush azt határozza meg, hogy a módosított lapok milyen időközönként ürüljenek ki a gyorsítótárból, és íródjanak a kulcs-érték tárba (véglegesítés). Ezt minél ritkábban teszi meg az alkalmazás, annál jobb teljesítményt ér el, feltéve hogy nem csordul túl gyakran a központi memória. A ritka lapürítés másik hátránya, hogy meghibásodás esetén a kulcs-érték tárból csak sokkal régebbi másolatokat lehet helyreállítani, tehát kevésbé biztonságos a rendszer.

A Quorum, vagyis a sikeresnek tekinthető lapíráshoz/-olvasáshoz szükséges minimális másolatok száma szétosztva a Muppet klaszter gépei közt. Ez az érték 1-től a klasztert alkotó gépek számáig növekedhet. Ha egy gépnél több helyen is létezik másolat egy lapról, akkor egy esetleges meghibásodás kisebb valószínűséggel okozhat adatveszteséget.

A Time To Live paraméter azt az időmennyiséget adja meg, ameddig egy kihasználatlan lapot bent kell tartani a Cassandrának a memóriában vagy a háttértáron. A kihasználatlanság ebben az esetben a frissítések hiányát jelenti. Ezzel a beállítással kordában lehet tartani a kulcs-érték tár telítettségét. Ez a beállítás update függvényenként állítható, így sokkal hasznosabb, hiszen amellett hogy el szeretnénk kerülni a tárhely betelését, lehet hogy határozatlan ideig szeretnénk egyes lapokat perzisztálni. Például egy olyan update függvény, ami az utóbbi fél óra legaktívabb Twitter felhasználóinak aktivitását jegyzi, fölöslegesen tárolná el egy órákkal korábban aktívnak számító felhasználó adatait.

## Hibakezelés

Egy Muppet klaszterben a meghibásodásnak leggyakrabban 2 oka lehet: az egyik gép meghibásodik, illetve egy dolgozó végrehajtási sora túlcsordul.

Az alkalmazáson belül minden dolgozó önállóan tartja nyilván a meghibásodott gépek listáját, azaz a nem elérhető dolgozók listáját. Mivel az is a dolgozókra van bízva, hogy egymással kommunikáljanak, és tegyék egymás végrehajtási sorába az eseményeket (ld. Elosztott végrehajtás), ezért ha nem sikerül kapcsolatot létesíteniük egy másik dolgozóval, akkor szólnak a vezérlőprogramnak (ld. 1. ábra: Master), hogy a nem válaszoló dolgozót futtató gép valószínűleg meghibásodott. Ha ez valóban így van, akkor a vezérlőprogram küld egy üzenetet a klaszterben levő összes még működő dolgozónak, hogy melyik gép nem működik, és módosítja a közös hash függvényt. Tehát, ha egy A dolgozó hiába próbálta menetrendszerűen B-nek átadni egy *e* eseményt, és a B-t futtató gép meghibásodott, akkor a hash függvény onnantól fogva mindig egy C dolgozóhoz fogja irányítani *e*-t.

A B-nek már sikertelenül küldött esemény ilyenkor elveszik, a B végrehajtási sorában levő eseményekkel, és a kulcs-érték tárban B-hez tartozó még nem véglegesített lapokkal együtt. Jelenleg a Muppet ilyenkor nem kísérli meg az elveszett események helyreállítását, hiszen sokkal nagyobb a hangsúly a valós idejű végrehajtáson és a rövid válaszidőn, mint az események maradéktalan feldolgozásán.

Ha a B végrehajtási sora telítődött, és az A akarna egy újabb eseményt beletenni, ilyenkor a B elutasítja az eseményt. Az elutasításra az A által meghívott hibakezelő mechanizmus többféleképpen reagálhat:

- Eldobja az elutasított eseményt és feljegyzi azt. A feljegyzés debuggolás, illetve későbbi újrafeldolgozás szempontjából hasznos lehet.
- Egy külön a túlcsordult események számára kialakított folyamba teheti, ahol speciális dolgozók kezelik a folyamot. Ezek a speciális dolgozók az eredeti átalakító műveletek helyett azok egyszerűsített változatát alkalmazzák, ami kevésbé számításigényes és gyorsabb átfutást biztosít az eseményeknek. Például az eredeti művelet helyett annak egy közelítő algoritmusát hajtják végre.
- Egy kicsit lassít az egész Muppet alkalmazás működésén, hogy B dolgozó győzze feldolgozni a végrehajtási sorába kerülő eseményeket.

## Lapok olvasása

Ahhoz, hogy a korábban már megemlített menet közbeni lap kiolvasást meg lehessen oldani a Muppet klaszteren kívülről, el kell indítani egy kis HTTP kiszolgálót a klaszter minden gépén ami fogadni tudja a bejövő kéréseket. Ily módon egészen 'friss' eredményekhez lehet hozzájutni a rendszer működése közben, ahelyett hogy a Cassandrából kellene lekérdezni egy nem annyira friss példányt. A HTTP szervereknek küldött URI-ben benne kell lennie az update függvény nevének és az esemény kulcsának ami alapján a lap beazonosítható.

## A Muppet 2.0-ás verziójának fejlesztése

A Muppet első változatában a dolgozók úgy vannak implementálva, hogy az számos hátrányt okoz. Egy dolgozó 2 szorosan együttműködő folyamatból áll:

- Perlben megírt vezénylő (conductor) folyamat: ez felel a logisztikai feladatok ellátásáért. Ez magában foglalja a végrehajtási sorok kezelését, az események más dolgozókhoz küldését, és a feldolgozó folyamat bemenetének és kimenetének működtetését stb.
- JVM-en futó feldolgozó (task processor) folyamat: kizárólag a Muppet alkalmazás fejlesztője által megírt update/map függvény kódját hajtja végre.

Ennek a tervezési döntésnek, és még más tényezőknek köszönhetően 4 komolyabb hibája van az 1.0-ás Muppetnek.

- A klaszter egy gépén futó összes dolgozónak be kell tölteni egy saját másolatot a feldolgozó folyamathoz szükséges JVM-en futtatandó kódból. Ezek a kód duplikációk a memória pazarlásához vezetnek.
- Túl költséges a vezénylő folyamatok által lebonyolított kommunikáció, pl. az események ide-oda küldözgetése.
- A klaszter egyes gépein minden dolgozónak külön lapjai vannak. Emiatt a gépen a lapok olvasási gyorsítótára teljesen külön kezelt lapokból áll, ahol mindegyik lapért csak a hozzá tartozó dolgozó felel. Ez komoly memóriapazarlást jelent. Például, ha egy gépen van 100 gyakran frissített lap, amit egyetlen update függvény kezel, akkor hatékonyan van kihasználva a memória. Ha viszont 5 függvény kezeli ezt a 100 lapot, a Muppet rendszer hasító függvénye egyenetlenül oszthatja szét a lapokat a függvények között. Ehhez hozzá tartozik az is, hogy egy gépen globálisan lehet csak beállítani a dolgozóknak jutó lap gyorsítótár méretét. Tehát ha az egyik dolgozónak 25 lap jut, a többi 4 közt pedig a maradék 75-öt kell elosztani, akkor is 25-25 lap méretű kell legyen egyenként a gyorsítótárak méretének. Vagyis összesen az optimális 100 helyett 125 lapnyi méretű az adott gépen a gyorsítótár mérete.
- Nehéz teljesen kihasználni a gépek processzorainak magját (már ha többmagúak a processzorok, de általában azok). Ugyanis a dolgozók száma nem a magok számához van igazítva, hanem a feladathoz, ill. az alkalmazás konfigurációjához.
  Egy sokmagos gépben ha erőltetjük, akkor sem éri meg annyi dolgozót futtatni ahány mag van, mert a fent említett 3 jelenség miatt a memória gazdálkodási problémák tovább súlyosbodnak. Illetve azt is figyelembe kell venni, hogy mivel minden dolgozó eleve 2 folyamatból áll, ilyenkor a processzorok gyorsítótárának tartalmát gyakran kell kicserélni a szálak közti váltogatáskor (context switch).

A Muppet újabb változata egy sor lényeges változtatással próbálja kiküszöbölni a fent említett 4 problémát. Itt a fejlesztés már a Java és a Scala programozási nyelven történt, ami azért előnyös, mert a Scala nagyon jól támogatja az adatfolyamok feldolgozását külön Stream adatszerkezettel és a hozzá fűződö műveletekkel [4].

Az első fontos szerkezeti változtatás, hogy a klaszter egyes gépein egy szálkészlet tervezési minta (thread pool) lett alkalmazva. Az így kezelt összes szál egy dolgozó, ami tetszés szerint bármelyik map vagy update függvényt végre tudja hajtani és menet közben akár váltani is tud

közöttük. Innentől kezdve annyi dolgozó szál futhat egy gépen amennyit a feladat, a gép processzora, vagy egy korlátozottan (párhuzamosan) hozzáférhető erőforrás megenged.

A dolgozó szálakon kívül, a klaszter minden tagján a háttér I/O műveletek (Cassandrába perzisztálás stb.) elvégzésére is fenn van tartva egy külön végrehajtási szál, illetve egy második szálkészlet a HTTP szerver működtetéséhez. A külön I/O szál biztosítja, hogy a kulcs-érték tárba írás nem blokkolja valamelyik dolgozó szálat és teljesen párhuzamos lehessen a rendszer működése.

A pazarló lap gyorsítótár allokálások elkerülése végett minden lapot a klaszter egységei egy központi gyorsítótárba helyeznek el.

A következő módosítás a klaszter elosztott végrehajtását érinti: minden dolgozó szálnak saját (az eseménykulcsokat tekintve) heterogén végrehajtási sora van. A dolgozó szál aszerint változtatja a viselkedését, hogy milyen esemény következik a sorában (válthat update és map függvény végrehajtása között).

Amikor egy újabb *e* esemény érkezik a klaszter egyik gépébe, egy hasítófüggvény az *e* kulcsa és a feldolgozásához szükséges map/update függvény alapján 2 dolgozó szálhoz tartozó végrehajtási sort jelöl ki. Az egyik lesz az ún. elsődleges sor, a másik pedig a másodlagos sor. Innentől 2 eset lehetséges:

- Ha a 2 sor közül akármelyikhez egy olyan dolgozó szál tartozik, ami már egy olyan függvényt hajt végre ami *e* feldolgozásához kell, akkor abba a sorba kerül az esemény. Ha ez mindkettőre igaz, akkor az elsődleges sor van előnyben részesítve.
- Ha az előző állítás egyik sorra sem igaz, akkor automatikusan az elsődleges sorba tesszük *e*-t, feltéve hogy a másodlagos sor nem sokkal rövidebb (kevésbé telített).

Az a viselkedés, hogy egy eseményt 2 sor közül csak az egyikbe utalunk, ahelyett hogy potenciálisan az összes sor valamelyikébe utalnánk, komoly előnyökkel jár. Ekkor a bejövő *e* esemény miatt nem kell 2-nél több végrehajtási sort zárolni amikor is meg kell vizsgálni, hogy melyik sorba is lehet tenni *e*-t. Mindezt ahelyett hogy az összes sort egy rövid időre zárolni kéne, ugyanis akkor az egész rendszer működése megakadna egy rövid időre (versengés alakul ki a sorok használatáért).

A másik, hogy az ugyanahhoz a kulcshoz és ugyanahhoz a függvényhez tartozó események nem hajlamosak a dolgozó szálak közt szétszóródni, ehelyett csak egy-néhány szál fog a legtöbb (ha nem is az összes) ilyen adattal foglalkozni. Ez pedig az adott $S_{U,k}$ lapért folyó versengést minimalizálja a szálak közt, nagyban javítva az átfutási időt (az ütemezési költség kisebb).

Ha a bejövő *e* eseményhez hasított elsődleges sorban már nagyon sok *e*-nek megfelelő esemény várja hogy feldolgozzák, akkor a másodlagos sorba kerül. Tehát elkerülhető az, hogy hirtelen sok azonos kulcsú, azonos függvényt igénylő bejövő esemény túlterhelje a néhány erre specializálódott szálat. És ilyenkor a rendszer alkalmazkodik, és több szál közt dobja szét ezeket az adatokat. Nem alakul ki torlódási pont.

Tehát a legfőbb oka annak hogy az 1.0-ás változattal ellentétben a 2.0-ás Muppet megengedi, hogy egy laphoz egy dolgozónál több is hozzáférhessen, a torlódási pontok kialakulásának elkerülése. Ha csak néhány szál verseng ugyanazért a lapért az elfogadható azzal szemben hogy túlcsorduljon az egyik szál várakozási sora és megakassza az egész rendszert. A dolgozók szálakká alakítása azért hasznos, mert így nem kell feleslegesen lemásolni a függvények algoritmusát leíró kódot, hiszen mindegyik szál osztozik ezeken. Vegyük észre, hogy mivel a szálak tetszőleges függvény kódját végre tudják hajtani, nincsen arra szükség, hogy a klaszter egyik gépében lévő dolgozó egy másik gépbe küldje az eredményét. Ezzel komoly kommunikációs költségeket lehet megspórolni. Végül a dolgozó szálak most már maximálisan képesek kihasználni a hardver nyújtotta párhuzamos számítási teljesítményt.

A továbbfejlesztett Mupettel elért eredményekért ld. a Kapcsolódó munkák részt.

# További kutatási terv

A vizsgált cikkben több ötlet is felmerül a Muppet keretrendszer továbbfejlesztésére. Ezek közül az egyik, hogy az alkalmazás futása közben lehessen le- és felcsatolni további gépeket a klaszterre. Ha ez megvalósulna, akkor a rendszer sokkal rugalmasabban tudná kezelni a meghibásodó gépek kiesését a klaszterből, amiket az alkalmazás újraindítása nélkül pótolni lehetne miután a meghibásodás oka ki lett javítva az adott gépeken.

Szintén a meghibásodás kezelésénél a meghibásodott gépen ragadt, végül eldobott eseményeket helyre lehetne állítani, hogy ha ezekre az eseményekre úgymond vissza lehetne forgatni a rendszer működését egy olyan pontra, amikor még egy olyan dolgozó végrehajtási sorában volt ami még mindig működőképes. Ehhez legalábbis arra lenne szükség, hogy minden dolgozó valahova feljegyezze a hozzá került események állapotát, ami a módosítás mellett egy eredeti másolat megtartását is jelentheti, pl. egy egészen rövid TTL értékkel. Ennek ellenére ez a megoldás nagy terhet róna az egész rendszerre, és valószínűleg jelentősen csökkentené az alkalmazás áteresztőképességét, csupán azért hogy néhány nem kritikus adat helyreállítható legyen. Egy fokkal hatékonyabb módszer lenne, ha ezeket az extra másolatokat a Cassandra kezelné. Például egy sűrűn módosított lapot ne is írjon felül közvetlenül a háttértárolón, majd a már elavulttá vált biztonsági másolatokat a tömörítő algoritmus periodikus futásakor egyúttal ki lehetne törölni. De ezt már korábban, az írások gyorsítótárazásához használt RAM-ban meg kell lépni, ami akár meg is tízszerezheti a használt RAM mennyiségét az egyes szeletek állapotaink őrzése, ha azt 10 lépésig jegyzi fel.

Ha képes lenne ez a mechanizmus érzékelni, hogy melyik szeletet a Muppet klaszter melyik gépéről módosítanak, akkor gépenként tudná csoportosítani a szeletekről készített biztonsági másolatokat. Ha figyelembe venné a dolgozók számát ezekre a lapcsoportokra, egy kicsit memóriatakarékosabban tudná eltárolni a biztonsági másolatokat.

# Irodalomjegyzék


[1] Jeffrey Dean and Sanjay Ghemawat: „MapReduce: Simplified Data Processing on Large Clusters” - http://research.google.com/archive/mapreduce.html
[2] Kin Lane: The Twitter Firehose - http://apivoice.com/2012/07/12/the-twitter-firehose/
[3] Twitter Blog: #numbers - http://blog.twitter.com/2011/03/numbers.html
[4] Scala API: Stream - http://www.scala-lang.org/api/2.7.5/scala/Stream.html
[5] TweetBeat Wants To Kill Hashtags On Twitter By Making Them Obsolete - http://techcrunch.com/2010/09/29/tweetbeat/
[6] ShopyCat hivatalos webhelye - http://www.shopycat.com/