Budapest Műszaki és Gazdaságtudományi Egyetem
Villamosmérnöki és Informatikai Kar
Adatbázisok Oktatási Labor

Adatbázisok haladóknak – VITMAV12

Oszlopcsaládok

Cassandra és HBase

Szerző:
Gábor Bernát

Vezető tanár:
Dr. Gajdos Sándor

2012. november 2.

Tartalomjegyzék

Ábrák jegyzéke

Táblázatok jegyzéke

Kivonat

A Google a 2000–es években azt tapasztalta, hogy nagy adat mennyiségek hatékony eltárolására és feldolgozására a relációs adatbázisok már küszködnek. Nem az adat mennyiség feltétlen a gond, hanem annak megbízható eltárolása és gyors feldolgozása.

Ennek orvosolására megalkották a saját fájl (Google File System), adattároló (BigTable) és adatfeldolgozó (MapReduce) rendszerüket. A problémával az Amazon is hamarosan szembesült, és saját rendszerük megkötései és prioritásai szerint létrehozták a Dynamo adat tároló rendszert.

E rendszerek közös és központi tulajdonságai az elosztottság (sok gép együtt, mintsem kevés erős) és a magas redundancia szint biztosítása (adat nem veszhet el). Ugyanakkor az adatokat főleg oszlopokba rendezve tároljuk el. Az ár, amit fizettünk ezért, hogy a régi relációs adatbázis megközelítést, elvek egy részét, mögöttünk kell hagynunk, és új adattárolási paradigmákat kell megismernünk.

Az oszlop orientált adat tárolás koncepció nem újak. OLAP rendszerekben már több évtizede jelen vannak relációs adatbázisokban. Az oszlop család megfogalmazást főleg e rendszerekre illenek. Az iparba a MonetDB, Sysbase avagy Vertica implementációi tűntek ki az évek során.

Az angol szakirodalomba úgymond „Wide" (azaz széles) oszlopcsaládok formájába utalnak arra, amit a Google és Amazon teremtett meg. Felépítésük alapján viszont találóbb a táblázatos tárhely, avagy a strukturált kulcs–érték tárrendszerek kifejezés.

Úgy a Google mint az Amazon rendszere is zárt, belsőleg használt. Kutatási beszámolóik alapján viszont elkészültek nyílt forráskód változatai is. A Google felépítéséből nőtte ki magát az Apache HBase (amely maga is épít az Apache Hadoop–ra). Az Amazon cikk meg a Cassandra rendszernek nyújtott referencia pontot. Napjainkra mindkettő már ipari standarddá vált és oly cégek használják szolgáltatásaik üzemeltetésére, mint a Facebook avagy a Twitter.

Jelen beszámoló indít a hagyományos oszlopcsaládok rövid ismertetésével, majd bemutatja, hogy mit is jelent általánosan egy „széles" oszlopcsalád. Egy–egy fejezet erejéig végül elmerül a Cassandra és a HBase adat modell és adat partíciós stratégiáinak ismertetésére. Az irat fő célja, hogy egy jó áttekintő képet nyújtson arról, hogy mikor érdemes és szükséges használni őket, és nem utolsó sorban melyik implementációt kapjuk le a polcról ilyenkor.

1. fejezet

Oszlopcsaládok

Amennyiben adatbázisainkat Edgar F. Codd cikke[1] nyomán táblákba rendezve képzeljük el két nagy irány tárul elénk: sorba vagy oszlopba rendezve. A tábla egy bejegyzése egy adott objektum attribútumait tárolja el. A sor orientált elképzelésben egy–egy bejegyzés, mint egy attribútum csomag van eltárolva. Tehát az adott bejegyzés attribútumai többnyire egymást követve, egy helyen lesz eltárolva. Ezzel szemben az oszlop orientált megalkotásban az objektum példányok attribútumai vannak egy helyre gyűjtve.

A 20.–ik század folyamán a gyakorlatban a sor orientált adatbázisok terjedtek el.Az oszlop orientált megközelítés erőssége a sor orientálttal szemben az, hogy az aggregációs műveletek rövidebb idő költséggel rendelkeznek. Ennek oka, hogy amennyiben csupán egy (vagy kevés) attribútum szerint akarunk elvégezni egy műveletet, akkor nem kell a lassú merevlemezről beolvasnunk a tábla bejegyzéseinek többi attribútumát is, mivel az érdekelt információ halmazt szekvenciálisan egy helyen tároljuk. Ilyen műveletek főleg az OLAP rendszerekben gyakori.

1.1. Oszlopba avagy sorba rendezve?

A sor orientált adatbázisok esetében egy összetartozó adat csomag attribútumaival egy összetartó egységben van tárolva. Ezzel szemben az oszlop orientált adatbázisok mögött az alap ötlet, hogy egy–egy adat csomó adatai bejegyzések, amely értéke egy attribútum halmaz valamely eleme lehet[2]. A halmaz elemeit egy helyen tároljuk.

Például, legyen egy cikkeket tároló adatbázisunk. Minden cikkhez létezik három attribútum: név, típus és ára. Az 1.1 táblázat szemléltet egy sor orientált tárolást. Ezzel szemben az 1.2 egy oszlop alapú megoldásra mutat rá.

Az OLAP (Online Analytical Processing) és adattárház rendszerek általában oszlop orientáltan tárolják az információt. E mögött az ok az, hogy a szükséges

Név	Típus	Ár
Élet–halál harc	Technológia	1
Vízilabda Kupa	Sport	2
A Prada növekszik	Divat	3

1.1. táblázat: Sor orientált adat tárolás

Név	Élet–halál harc	Vízilabda Kupa	A Prada növekszik
Típus	Technológia	Sport	Divat
Ár	1	2	3

1.2. táblázat: Sor orientált adat tárolás

aggregációs műveleteket gyorsabban végre lehet hajtani, mert ha például átlag árat kell kiszámolnunk, a szükséges adatokat szekvenciálisan könnyedén kiolvashatjuk érintetlenül hagyva a többi attribútumot. Továbbá a kulcs–érték rendszerhez hasonlóan az adatcsomók között laza a kapcsolat.

Ugyanakkor tárhelybe is hatékonyabbak, hiszen az oszlopok entrópiája általában kicsi, e miatt meg nagy hatékonyságú tömörítéseket végezhetünk el. Elterjedt oszloporientált relációs adatbázis implementációk: a Vertica, Sysbase avagy a MonetDB rendszerek.

1.2. A „széles” oszlop családok

A NoSQL világban viszont az oszlopcsalád fogalma mást rejt, mint csupán az 1.1 fejezetben definiált tulajdonságok.

1.2.1. Történelem

Megjelenésüket a 2000–es évek első feléhez köthetjük, amikor is eleget fejlődött a világ ahhoz, hogy mint tároló kapacitás és hálózati elérhetőség szempontból megoldható legyen tömérdek adat eltárolása. Az eltárolás mellet felmerült az igény, hogy azt feldolgozzuk és információkat nyerjünk ki belőle, akár közel valós időben is. E problémával az internet világ akkori legnagyobb gigásza, a Google, is szembesült.

Hamarosan rá kellet döbbenniük, hogy a relációs adatbázisok által biztosítót számos szolgáltatás inkább útjában állt az adat gyors és hatékony tárolásának, illetve feldolgozásának, mint sem segít benne. Mielőtt teljesen lemondunk viszont a relációs adatbázisokról azért érdemes lehet elvégezni pár optimalizálási műveletet. A rendelkezésünkre álló eszközöket az 1.3 fejezetben ismerhetjük meg.

Amennyiben viszont mindezeket elvégeztük és alkalmazásunk oly sikeres, hogy még mindig tovább növekszik nincs más lehetőségünk, mint alternatív tárolás után nézni. Sajnos az optimalizálási lépések többsége nem más, mint újabb és újabb szolgáltatást kikapcsolása, arról való lemondás. Ahogy elvégezzük ezeket, hamarosan eljutunk azon ponthoz, hogy a relációs adatbázisunk már inkább hasonlít egy kulcs–érték tárolóhoz. Ekkor jogosan gondolhatunk arra, hogy ennél biztos jobbat is tudunk csinálni.

A Google is hasonlóan cselekedett és mivel kor legnagyobb teljesítményű számítógépe sem volt képes az Google által generált adatmennyiség egy gépen való tárolására, így nem meglepő az elosztott architektúra felé való orientáció. Egy olyan rendszer után kutattak, amely könnyedén tud tárolni és feldolgozni nagy mennyiségű adatott. Mind e mellet a skálázhatósága se jelentsen gondot, és lehetőleg könnyen beszerezhető és karbantartható hardver egységekre építsen. Itt olyan elemekre gondoljunk amelyek akár a helyi számítástechnika szaküzletben is megtalálhatóak.

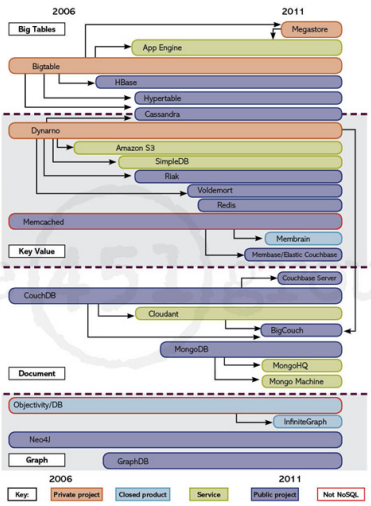
Hamarosan rá kellet döbbenniük, hogy ilyen még a piacon nincs, és mint minden magára adó nagy cég, a megoldás evidens volt: építetek egyet saját maguk. Kutatásaik eredményét több cikkben is publikálták, mint például a Google File System[3], a Google MapReduce[4] avagy a Google BigTable[5]. A BigTable a NoSQL alapú oszlopcsaládok atyát jelenti. Ez a Google File System és a Google MapReduce–ra épít, futás közben azt felhasználja.

A problémával hamarosan számos más cég is szembesült, és így a Google cikkeit felhasználva 2006–ban a *Nutch* a Google cikkeit felhasználva megírta saját implementációjukat a Google fájl rendszernek. Eredményük nagysága, hogy mindezt nyílt forráskód formájában tették. Innen nőtte ki magát a ma Hadoop egyik építő elemének számító HDFS. Majd erre alapozva 2007–ben a PowerSet cég épített egy ugyancsak nyílt forráskód adat tárolót: a HBase–t (*H*adoop Data*base*).

Ezzel egy időben az Amazon is megvalósított egy hasonló rendszert, amely oly követelmény rendszerből indult ki amely a saját cégükön belül volt jellemző. A Google–hoz hasonlóan ők is cikkben publikálták megoldásuk, tanulságaikat Amazon Dynamo néven[6]. A Facebook ebből kiindulva építette meg 2008–ban a Cassandra oszlop alapú NoSQL adatbázisukat.

E beszámoló írásakor bátran kijelenthetjük, hogy a NoSQL világ reneszánsz korszakát éli. Napról, napra újabb és újabb típusú adatbázisok jelenek meg, amelyek elvetik a relációs adatbázis sémát és valamilyen alternatív adattárolási struktúra, illetve adat modellezést használva, bizonyos esetekre ideális adatbázisokat valósítnak meg. Ezek ősei maradnak a BigTable és Dynamo, ahogy az 1.1 ábra szemlélteti is.

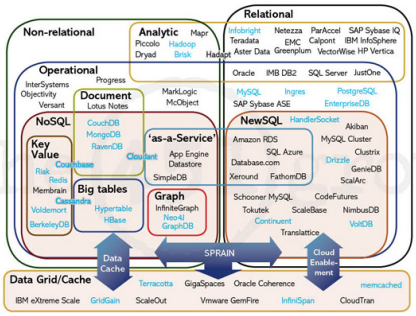
Az utóbbi években megfigyelhető, ahogy a NoSQL adatbázisokból származó tanulságok visszaömlenek a relációs adatbázisok világába megalkotva fél hibrid



1.1. ábra: A NoSQL világ felépítése

Forrás: `http://blogs.the451group.com/information_management/2011/04/`

rendszereket. Ezeket a szakirodalomban *NewSQL* rendszerekként említik, kiteljesítve az aktuális adatbázis világképet, ahogyan azt az 1.2 ábra is szemlélteti.



1.2. ábra: Az adatbázis világ térképe

Forrás: `http://www.the451group.com`

1.2.2. CAP

Ha adatbázisok architektúrája után kutatunk, hamar szembesülünk a CAP hármassal. A CAP az angol *Consistency* (konzisztencia),*Availability* (elérhetőség) és *Partition Tolerence* (partíció tolerancia) szavak rövidítéséből származik. E három fogalmat együtt először 2000–ben Eric Brewer vitaindító beszédben használta,

mint egy sejtés[7]: szerinte amint alkalmazásaink minél inkább web orientáltak lesznek, annál kevésbé szabadna az adat konzisztenciára gondolnunk. Az ok, hogy a magas elérhetőségi szint nem valósulhat meg konzisztencia megkötések betartásával.

Két évvel később Seth Gilbert és Nancy Lynch az MIT–ról ezt formálisan is bebizonyítót[8], igazat adva Brewernek. De mit is jelent mind ez? Julian Brown bejegyzésében többet tudhatunk meg[9]. E szerint, vegyünk egy mindennapi példát. Legyen egy online könyvesboltban egy könyvből csupán egyetlen példány, továbbá legyen egy A és egy B kliens.

A érkezik először az oldalra, kosárba teszi a könyvét és tovább nézelődik. B is megérkezik időközbe, kosárba rakja a könyvet és egyből folytatja is a megrendelési oldallal. A kérdés, hogy hogyan kezeljük a helyzetet, hiszen evidens, hogy vagy A vagy B nem fogja tudni megvenni a könyvet.

Definiáljuk először a CAP három betűjét:

Konzisztencia Egy konzisztens rendszerben vagy az egész rendszer működik vagy semmi. Gilbert és Lynch ezt atomicitás–ként nevezik meg, és ez érhető is, hiszen azt garantálja, hogy a rendszer soha sem lesz egy átmeneti állapotban. Bármely művelet egységnyi idő alatt hajtódik végre. Példánk esetében a könyvet vagy megvesszük, vagy nem. Félig könyvet nem lehet megvásárolni.

Ha mindkettőjük megveheti a könyvet rögtön szembesültünk is egy konzisztencia szabály megsértésével: a raktárban levő könyv mennyiség, és az eladott mennyiség különbözik, nincs szinkronban. Erre megoldást jelenthet például, ha a kosárba tevéskor a raktáron levő könyvek számát csökkentjük, így B már nem tudja a vásárlás műveletet végrehajtani.

Elérhetőség E szerint az oldal elérhető, azaz betölthető és nem egy hiba üzenettel tér vissza a böngésző. Ez legtöbb esetben kritikus, hiszen egy nem elérhető szolgáltatás semmire sem jó.

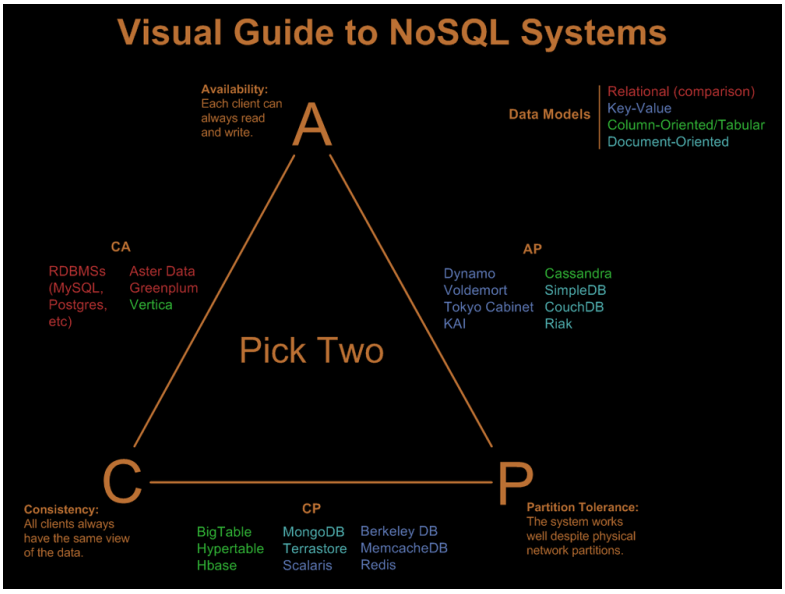
Partíció tolerancia Amíg az összes alkalmazás logikát egyetlen gépen tároljuk addig minden szép és jó. Az elérhetőség szerint vagy fut a rendszerünk vagy nem. Persze, ez nem mindig egy opció, hiszen így könnyen leterhelhető rendszerünk és hardver meghibásodások ellen sem vagyunk védve. Viszont amint a terhet több egységre is leosszuk, partíciók alakulnak ki.

Tegyük fel, hogy két egységre osztottuk le. Amint megszűnik a kommunikációs lehetőség köztük lehetetlen lesz szinkronban tartani a kettőt. Valahogy azonban orvosolnunk kell ezt, hiszen a rövid idejű kommunikációs hibák mindennapiak az elosztott rendszerek világában.

A CAP tétel szerint a fenti három meghatározásból egy időben csak kettőt lehet megvalósítani. A relációs adatbázisok általában a konzisztencia és az elérhetőséget

válasszák. De a partíció tolerancia forrása a skálázhatóság megoldása: elosztott rendszerek összmunkája. Ma már tudjuk, egy skálázható rendszerhez nem minél erősebb, hanem minél több gépet használunk.

Hogy a valós életbeli analógiával éljek, hidat nem egy szuper erős emberrel húzzunk fel, hanem sok átlagember összmunkájával. A NoSQL világban az, hogy mely betűt hagyjuk háttérben már nem egyértelmű. Az 1.3 ábrán láthatjuk, hogy egy–egy adattároló implementáció a háromszögben hol helyezkedik el.



1.3. ábra: A CAP hármas

Forrás: `http://blog.nahurst.com/visual-guide-to-nosql-systems`

Az elérhetőség fontossága miatt logikusnak tűnhet, hogy a partíció toleranciáért a konzisztenciáról mondunk le, de a helyzet nem ennyire komor. A fenti szabály akkor él, ha az elemek fenti definícióját használjuk. Csupán a konzisztencia definíciónkon kell kicsit módosítani (lazítani) és akár mindhárom elemet az asztalon hagyhatjuk. A CAP háromszögre, mint háromszög érdemes gondolni, a csúcsok a határesetek, amik magukban használhatatlan. Viszont a belsejében rengeteg átmeneti pont létezik, ami feladatunkra elégséges lehet.

1.2.3. Általános struktúra

A szakirodalomba, főleg Edlich, német professzor, nyomán[10][11], azon NoSQL adatbázisok, amelyek adat modelljük főleg oszlop alakban rendeződik „széles” oszlopcsalád adatbázisnak nevezzük. A név viszont cseppet megtévesztő, mert bár

való igaz, hogy oszlopokat használunk fel, az lényegesen eltér a relációs oszlop adatbázisok struktúrájától.

Sokkal inkább találó lenne rájuk a táblázatos tároló, avagy a strukturált kulcs–érték tároló[12]. A korábbi név oly szinten állja meg helyét, hogy a hagyományos oszlop alapú adatbázisokkal szemben itt lehetséges (sőt, sokszor ajánlott), hogy egy–egy úgynevezett táblában több millió oszlop egyidejű létezése is.

1.3. Relációs adatbázisok skálázása

Legyen a következő feladat: tervezzünk egy valamilyen szolgáltatást nyújtó rendszert, amely központi eleme valamely adat tárolása és feldolgozása. Kezdetben felírjuk az adatbázisunk modellét, az adatokat normalizáljuk táblákban és a köztük levő kapcsolatokat idegen kulcsok bevezetésével modellezünk. Továbbá a táblákra indexeket vezetünk be a gyors keresés és szűrés érdekében. Ha több táblából kiszámítható adat érdekel valószínűleg végre fog kelleni hajtanunk egy vagy több JOIN művelet az SQL lekérdezés kiértékelésére.

Továbbá az adatbázisunk robusztussága érdekében felhasználjuk a relációs adatbázisok olyan funkcionalitásait mint a *tárolt eljárások*. Ennek segítségével garantálni tudjuk, hogy ha még több forrásból is egy időben frissítünk adatott, adatbázisunkban a különböző táblák adatai mindig megegyeznek, sose mondanak ellent egymásnak. A *tranzakciók* lehetővé teszik, hogy akár több tábla adati frissítését is mind egy atomos, egységnyi művelet hajtsuk végre.

A relációs adatbázisok biztosítják az ACID tulajdonságokat, amelyek egy erős konzisztenciát tesznek lehetővé az adatbázis keretén belül. Továbbá az *SQL* nyelvvel deklaratív módon általános lekérdezéseket fogalmazhatunk meg. Az *alkalmazás séma* eltakarja, elölünk, hogy a háttérben pontosan hogyan is van tárolva az adat. Lehetővé teszi, hogy az adataink felhasználására összpontosítsunk a tárolási adatstruktúrák helyet.

Rendszerünk futtatásában mindez elég hosszú ideig jól szolgál. Azonban, amint egyre több és több felhasználó csatlakozik szolgáltatásunkra ara leszünk figyelmesek, hogy adatbázisunk egyre jobban le van terhelve. Az első lépés e terhelés csökkentése felé, hogy szolga szervereket iktatunk be. Ezeket csak párhuzamos olvasásra használjuk. Továbbra is egyetlen mester adatbázis van, amely fogadja az összes beérkező írást. Az alkalmazások nagy részében az írások száma jócskán alacsonyabb az olvasások számával, így ez egy könnyű kiút lehet a problémából.

Viszont, ha a szolgáltatásunk még népszerűbb lesz, eljutunk azon pontra, hogy már ez sem segít. Ekkor beiktathatunk egy cache–elő rendszert, mint például a Memcached. Most már az olvasást egy nagyon gyors, memória alapú rendszer szolgálja ki, de cserében elveszítünk az erős konzisztencia garanciánkat, mert a memória és a háttér tároló között eltérések fordulhatnak elé. De ez nem add

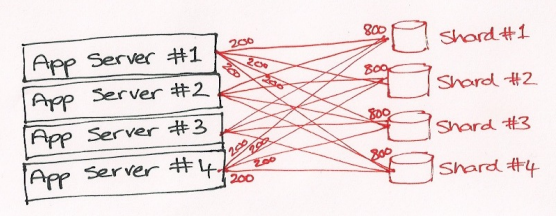
megoldást a sok párhuzamos írásra.

A függőleges skálázás az egyetlen biztos eszközünk: több magos processzor, több memória és gyorsabb háttér-tároló. De ez igencsak drága lehet, főleg ha a korábbi mester–szolga architektúrát már alkalmaztuk, hiszen a mester és a szolgák egyforma erősnek kell lennie. Ellenkező esetben előfordulhat, hogy a szolga nem képes tartani a lépést a mesterrel.

A szolgáltatás fejlődésével egyre több új funkció iránti igény merül fel, amelyek kiszolgálása valószínűleg akár komplex JOIN műveletek végrehajtását is igényeli. Ami ugyanakkor még több adatbázis lekérdezés megjelenését jelenti. E pontban a JOIN műveletek már igen költségesek, és a tábláinkat denormalizálni kell, hogy elkerüljük e költséget.

Ha minden szakad a tárolt eljárásokat is ki kell kapcsolnunk, hiszen akár már ezek is túl sok ideig futnak. E fázisban már az adatott nem úgy tároljuk, hogy azt kényelmes legyen feldolgozni, hanem a használt lekérdezésekhez optimalizálva.

A következő lépés, hogy a gyakran bejövő komplex lekérdezéseket idő előtt lefuthatjuk (korai materializáció). Végezetül az indexek vezetését is kikapcsoljuk, mivel már ezek karbantartása is igen költséges művelet a leterhelt adatbázisunk számára. Ha forgalmunk még tovább növekedhet muszáj lesz adatbázisunk részekre leosztani (sharding), de ez egy adminisztrációs fejfájás és igencsak nehéz jó, könnyen skálázható módon kivitelezni. Már az 1.4 ábrán szemléltet egyszerű leosztás megvalósítása is probléma dús és hosszas művelet lehet.



1.4. ábra: Adatbázis feltördelése darabkákra

Forrás: `http://www.dbshards.com/articles/database-sharding-configuration/`

Mindez azt mutatja, hogy a relációs adatbázisok felépítésük miatt nehézkesen skálázhatóak. Ne értsük félre, a relációs adatbázisok még mindig a legjobb és legkényelmesebb megoldást szolgálják egy adott szintig. A kérdés az, hogy ha tudjuk, hogy ezt a szinten túl lépjük egy idő után, nem lenne értelmesebb egy olyan architektúra mentén elindulni, amely mindezt könnyedén megvalósítja? Egy ilyen megoldás lehet a NoSQL oszlop alapú adatbázisok.

2. fejezet

Implementációk

A NoSQL alapú oszlopcsaládok között két implementációnak sikerült széles körű ipari használhatott elérni: a Cassandra és a HBase. Mindkettő Apache projekt. Architektúrájukban megfigyelhető kiinduló pontjuk, a Google BigTable és az Amazon Dynamo közötti különbség. Míg az első a meglévő Google fájl rendszere épít és kihasználja az általa nyújtott absztrakciós szintet (szolgáltatásokat), addig a későbbi önállóan is működő képes rendszer. Ennek megfelelően az összes problémát saját maga próbálja megoldani.

2.1. Apache Cassandra

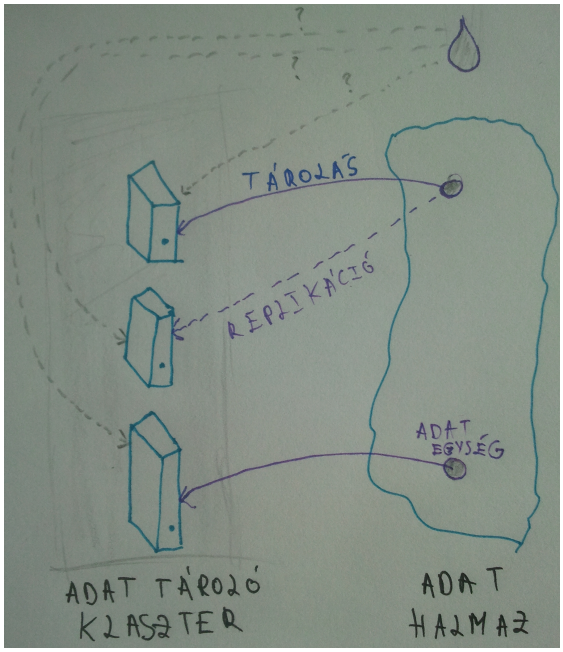
A Cassandrát a Facebook fejlesztette, első körben, mint skálázható adattároló a szociális hálójához. Az első fejlesztők között volt az Amazon Dynamo megalkotója, Avinash Lakshman[13], így nem meglepetés, hogy főleg annak architektúráját és elképzelését követi. 2008 júliusában a forráskód nyílttá vált, a Google Code web felületen keresztül. 2009 márciusában a projekt az Apache projekté alakul át. 2010 februárjában pedig fő szintű projekté alakul át az Apache szervezet keretében. Licensze ennek megfelelően az Apache Licensz 2.0[14].

Két kulcs pontja van egy Cassandra rendszernek: az adat partíció és az adat modell.

2.1.1. Adat partíció

A Cassandra egy elosztott alapú adattároló rendszer. Ez azt jelenti, hogy egyszerre több számítógépen is futhat ugyanazon adattároló példány. Ebből következik,

hogy az egyik feladat az adatok leosztása a rendszer egy-egy gépére. Induljunk ki egy adathalmazból, amint tárolni szeretnénk. Tegyük fel, hogy ez már valamilyen logika szerint el van osztva. Beérkezik egy új adategység a klienstől, a rendszer egy adott gépéhez. Kérdés: hova rakjuk az új adategységet, hol tároljuk el, hogy később könnyedén megkapjuk? A helyzetet ábrázolja a 2.1 ábra.



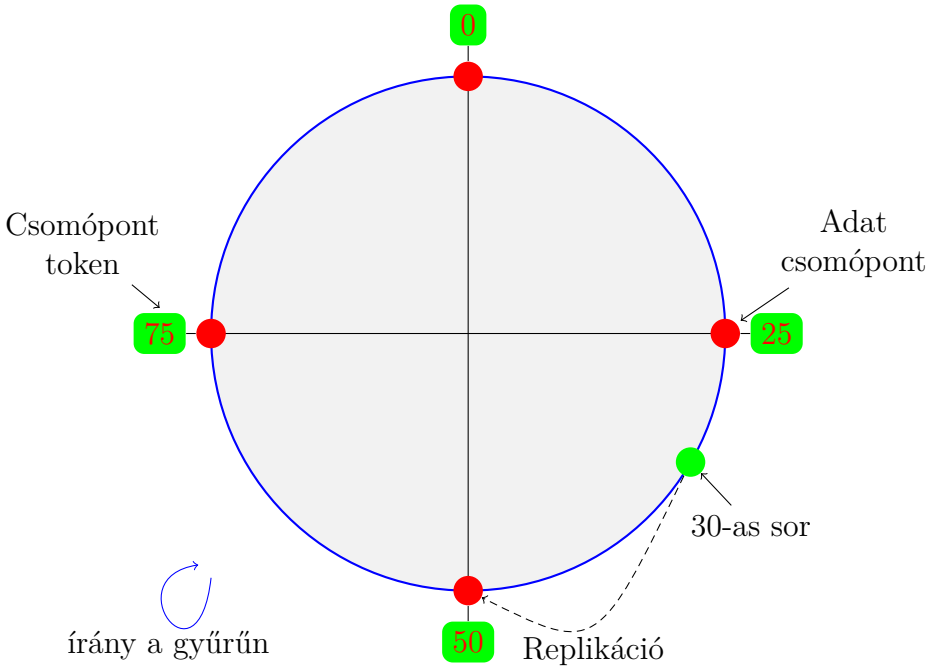
2.1. ábra: Adat halmaz tárolása klaszter felett

A Cassandra válassza erre az *adat gyűrű*. Ennek egy sematikus felépítését a 2.2 ábra szemlélteti. Minden adat egységnek van egy sor azonosítója, ez alapján számolódik rá egy *MD*5 hash érték. A gyűrű e hash értelmezési intervallumot osztja le a gépekre. Minden rendszerbe résztvevő gép kap egy értéket a hash intervallumból. Így a gépek egy kört alkotnak, mivel a legmagasabb hash értéket kapó gépet a legkisebb értékű követi.

Egy adott gép azon adat bejegyzéseket tárolja, amik az ő és előtte levő hash érték intervallumba tartozik. A helyzet kicsit tovább bonyolódik, amikor redundancia is bejön a képbe; hiszen most már nem csak egy helyre, hanem több gépre is ki kell írnunk ugyanazon adatott.

A Cassandra ebben az esetben a gyűrűn tovább haladva, az éppen beállított replikációs szint szerint, következő n fizikai egységre is kiírja az adatott. A konkrét egységek kiválasztásakor számításba vehet az adattárházak fizikai struktúráját is. Így biztosítva, hogy amennyiben még az egész tárház is kiesik egy másikba egy gépen mindig elérhető lesz a szükséges adat.

A gyűrűből az írás és olvasás a lehető legjobb teljesítmény elérésének érdekében



2.2. ábra: A Cassandra adat gyűrű

futás időben konfigurálható. Rendelkezésünkre állnak oly módszerek, mint *egy*, *kettő*, *három*, *bármely*, *mindegyik* avagy *lokális kvórúm*. Az utolsó mód akkor lesz létfontosságú mikor földrajzilag egymástól távoli adat tárházakat rakunk egy rendszerbe. Legyen például egy online üzletház, amely világ szinten üzemel és szállít. Természetes elvárás, hogy a magyar országi klienseknek a szolgáltatás ugyanolyan jól működjön, mint az Egyesült Államokban.

Viszont ennek megvalósításába beleütközünk a fénysebességből adódó adat átvivő késleltetési időbe, amely 100ms feletti egy óceánon keresztüli útvonalon. Alkalmazás és adatbázis szervereink nem lehetnek sem csak Magyarországon, sem csak az Egyesült Államokban, hiszen a túloldalon a szolgáltatás jóval lassabb lenne, és ennek megfelelően gyengébb élményt nyújthat. Megoldás, hogy mindkét kontinensen elhelyezünk egy–egy klasztert és egy adott kérés kiszolgálásához lehetőleg csak a földrajzilag helyit használjuk fel.

Ilyenkor a lokális döntő képesség (kvórum) opcióval azt biztosíthatom, hogy a lekérdezett adat a helyi adattárház szintjén konzisztens. A kvórum a létező egységek felét plusz egy számát jelenti, tehát egy 5 egységből álló Cassandra hálózat esetében legalább 3 ugyanazzal az érték válasszal tér vissza. A lokális kvórum ennek megfelelően például a magyarországi Cassandra gépek fele plusz egytől várja el ugyanazt a választ. Amint ez megtörtént a kliensnek visszatéríti a kapott adatott.

Ez gyakran elégséges, hiszen elég kis arányban fordul elő, hogy egy–egy kliens

kiszolgálására globális nézetre van szükségünk, hiszen például az amerikai rendelés kiszolgáláshoz csupán az amerikai készlet szükséges. Ugyanakkor, ha globális jelentéseseket, elemzéseket kívánnunk futtatni a lehetőség megvan. Mivel rendszerünk a globális olvasási kéréssel még egyként látható.

2.1.2. Adat modell

Az adat modell lényegben eltér a relációs adatbázisoktól. Mi több, szerintem e rendszerekben megfordul a hagyományos relációs adatbázisok használati munkamenete. A relációs adatbázisokra jellemző, hogy strukturáljuk és rendszerezzük a rendelkezésünkre álló adatot, majd különböző főleg JOIN műveletek használatával a kért információt SQL nyelv segítségével kérjük le. Mikor egy Cassandra rendszert használunk a JOIN és az SQL–t is felejtsük el. Habár létezik egy SQL–re emlékeztető CQL (Cassandra Query Language) ez nem támogat JOIN műveleteket.

Ebből adódóan, egy jól működő rendszerbe nem az adatbázisunkban levő adatott rakjuk össze, hogy kinyerjünk valamilyen információt, hanem úgy tervezzük meg adatbázisunkat, hogy könnyű legyen azt lekérdezni. Még akkor is, ha emiatt egy–egy adatott többször is el kell tárolni. A cél a gyors és hatékony lekérdezés, amely érdekében az olcsó merevlemez tár mennyiséggel vagyunk hajlandóak fizetni. Egy Cassandra adatbázis megalkotása nem úgy kezdődik, hogy milyen adatunk van, hogy tároljuk el; hanem, hogy milyen lekérdezéseket akarunk majd elvégezni?

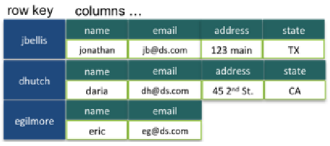
A relációs adatbázisok belli sémák kulcs téré módosulnak, a táblák oszlop családokká. A hasonlat viszont itt megáll, mert nincs idegen kulcs, nincs kapcsolat a családok között és olvasás közben nem lehetséges a családokon JOIN művelet. Ennek megfelelően, ha valamilyen adat szükséges egy lekérdezéshez, akkor annak ott kell lennie az oszlop családban, mint egy oszlop.

Az oszlop családok oszlopokból állnak. Egy oszlopcsalád lehet *statikus* vagy *dinamikus*. Az oszlopok meg a következő típusok közül vehetnek fel egyet: *standard*, *összetett*, *lejáró*, *számláló* avagy *szuper*.

Oszlop család típusok

Statikus Az egyszerűbb, többé–kevésbé megfelel a relációs adatbázisok tábláinak, itt is az oszlopcsalád oszlopait előre definiáljuk és a futás időben beszúrt értékeknek e struktúrát követnie kell. A 2.3 ábra például egy felhasználó adatait tároló oszlopcsaládot mutat.

Abban viszont eltér a relációs táblláktól, hogy nincs szükség NULL elemre a hiányzó adat mezők (oszlopok) jelzésére. A Cassandra oszlopcsaládok natív módon képesek úgy nevezett ritka tárolásra. Amely oszlop nem létezik az oszlop családból, azt egész egyszerűen nem vesszük fel a sor beszúrásakor.



2.3. ábra: Statikus oszlopcsalád a Cassandrában[14]

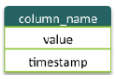
Dinamikus Ez esetben akár az oszlop név maga is jelentheti az adatott, és így ezt a kliens futásidőben szolgáltatja, a Cassandra meg dinamikusan hozza létre. Funkcionálisan úgy lehet felfogni, mint egy előre materializált nézetet, amelyet előre kiszámolunk, és hatékony módon tárolunk el. Ez esetben csak az oszlop név és tartalmának típusát kell előzetesen definiálni. A 2.4 ábra például felhasználók közti kapcsolatot modellez egy oszlop családba foglalva mindezt.

2.4. ábra: Dinamikus oszlopcsalád a Cassandrában[14]

Oszlop típusok

Az oszlop a legkisebb adat elem a Cassandrában: egy név, érték és egy időbélyeg páros.

Statikus Az oszlopnak egy névvel kell rendelkeznie, amely lehet: statikus, avagy dinamikus (attól függően, hogy előre definiált, avagy futás közben hozzuk létre). Egy oszlopnak az értéke fakultatív. Az időbélyeg megadható, avagy amennyiben hiányzik a beszúrás pillanatbeli időbélyeggel lesz ellátva. Olvasáskor mindig csak a legújabb érték lesz visszatérítve (hacsak explicit nem kérünk egy régebbit).



2.5. ábra: Oszlop struktúra a Cassandrában[14]

Összetett Az összetett csoportok egy újabb absztrakciós szintet vezetnek be és segítségükkel klaszterezett sorokat lehet tárolni. Az összes logikai sor, amely partíciós kulcsa ugyanaz lesz, egyetlen széles sorként lesz eltárolva. Így egy sorba akár 2 milliárd oszlop is lehet. Az összetett oszlop segítségével a sorokat teljesen denormalizálható egy előre definiált összetett kulcsot felhasználva.

Legyen egy tweet alkalmazás, ahol az adathalmazhoz tartozik a bejegyzés szövege, felhasználó név és azok követő listái. A feladat, hogy gyűjtsük össze egy adott felhasználó követőinek bejegyzéseit. Ehhez hozzuk létre először egy *tweet* táblát, amelybe bejegyzéseket tárolunk felhasználókhoz. Generáljunk automatikusan minden bejegyzéshez egy azonosítót. A 2.1 tábla egy példa bejegyzést mutat.

```
CREATE TABLE tweets (
tweetid uuid PRIMARY KEY,
author varchar,
body varchar
);
```

tweet_id	author	body
1	A	XY
2	B	YZ
3	C	ZT

2.1. táblázat: Egy tweet oszlopcsalád

A tweet táblát denormalizáljuk, az összetett oszlopok az összetett fő kulcs szerint valósul meg:

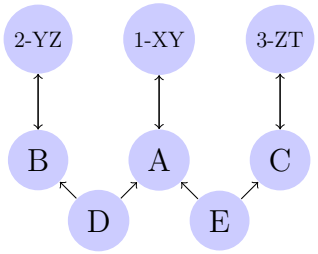
```
CREATE TABLE timeline (
userid varchar,
tweetid uuid,
author varchar,
body varchar,
PRIMARY KEY (userid, tweetid)
);
```

A két azonosító együtt azonban szükséges is, hiszen magára egyik sem lenne egyedi. A 2.2 tábla ad erre is egy példát. A két tábla felépítését a 2.6 ábra szemlélteti. A háttérben továbbra is csak egy fő kulcs lehetséges. Ezért, az összetett kulcs első eleme lesz a partíciós kulcs. Az dönti el, hogy az adat gyűrűben a bejegyzés hova kerül. A kulcs maradék része szerint klasztereződnek az illető sorba bejegyzett elemek.

A klasztereződés abban valósul meg, hogy a tároló motor létrehoz egy indexet, és mindig a szerint tárolja az adatott. Mivel a felhasználó név az adat partíciót megvalósító kulcs, a maradék kulcs (a tweet id) szerint lesz rendezve a többi bejegyzés. A 2.2 példa esetében a háttérben a 2.3 módon lesz eltárolva.

user_id	tweet_id	author	body
D	2	B	YZ
D	1	A	XY
E	4	C	ZT
E	1	A	XY

2.2. táblázat: Egy timeline oszlopcsalád



2.6. ábra: A tweet és timeline tábla felépítése

D	[2,author]:B	[2,body]:YZ	[1,author] : A	[1,body] : XY
E	[4,author]:C	[4,body]:ZT	[1,author] : A	[1,body] : XY

2.3. táblázat: Egy összetett oszlop példa tárolása

Ezen táblákkal egy adott felhasználó követőjének a tweet listájának lekérdezése CQL nyelvben:

```
SELECT * FROM timeline WHERE userid = D
         ORDER BY tweetid DESC LIMIT 50;
```

Láthatjuk, hogy habár a háttérben a Cassandra alapjában eltér a relációs adatbázisoktól, formailag a CQL igyekszik követni az SQL nyelvet.

Lejáró oszlopok Bármely oszlophoz hozzácsatolhatunk élettartamot, angolul: Time to live - *TTL*, amelyet beszúráskor lehet beállítani. Amint ez lejár az oszloptörlésre megjelölve lesz. Ez azt jelenti, hogy lekérdezéskor már nem számit. Törölve viszont még nem lesz. Az csak a periodikus csomagolási műveletek elvégzésekor történik meg.

A Cassandra rendszerben egy elem beszúrása beszúrás amennyiben az oszlop elem még nem létezik, egyébként meg frissítés. Oszlop élettartamot frissíteni nem

lehet, de hasonló hatást érhetünk el, ha kiolvassuk az elemet, majd ismét beszúrjuk új lejárati idővel. A lejárati idő bekapcsolása természetesen további 8 bájt merevlemez és memória költséggel jár.

Számláló oszlopok Egy számláló oszlop speciális oszlop családot igényel, e miatt gyakorlatilag saját oszlopcsaládot kell létrehozni. Ellenben, amint ez megtörtént egy–egy számláló érték frissítésére csupán a változási értéket kell megadnunk.

- ☺ Működik elosztott rendszerekbe.
- ☹ A Cassandra szerverek órái szinkronban kell, hogy legyenek a helyes működéshez.
- ☹ Íráskor elégséges az egy írási parancs, de mivel a rendszer a számlálókat konzisztensnek kell, hogy tartsa a háttérben egy globális olvasási művelet is végre fog hajtódni.

Szuper oszlopok Ezek egy extra absztrakciós szintet vezetnek be, az oszlop és az oszlop család közzé. Sajnos csak kis oszlop szám mellet működik hatékonyan ezért helyettük az összetett oszlopok használata ajánlott.

2.1.3. Adat típusok

Típusa lehet az oszlop értékének (érvényesítő) és az oszlop nevének (komparátor).

Érvényesítő Meghatározza, hogy az érték beszúrható e vagy sem ebbe az oszlopba. Bármikor újra definiálható.

Komparátor Meghatározza, hogy az oszlop név helyes e (dinamikus oszlopcsalád esetében) és az oszlop családban az oszlopok eszerint kerülnek rendezésre. Definiálás után többet nem változtató meg, tehát gondoljuk meg alaposan, hogy mit választunk.

2.1.4. Oszlopcsalád tervezési tanácsok

Kezdjük azzal, hogy megfogalmazzuk, hogy milyen információk érdekelnek. Gondoljuk el, hogy hogyan tudjuk ezeket lekérdezni. Minden adat, ami egy lekérdezéshez szükséges tegyük egy oszlop családba, de ne rakjuk mindent egy oszlop családba. Ha egyes adatott többször is eltárolunk az nem gond, a Cassandra nézőpontjából a tárhely olcsó. Cserébe gyors lekérdezéseket kapunk.

Gondoljuk meg, ha szükségünk van generált egyedi azonosítóra, avagy az természetesen adódik. A természetes kulcs tulajdonságai:

☺ A tárolt adatot szabad szemmel is könnyedén olvasható.

😐 Elkerüljük a denormalizációs műveleteket és az index alkalmazásának szükségességét.

☹ Nem lehet megváltoztatni a kulcsot. Tehát például a felhasználó nevet miután létrehoztuk többet nem lehet megváltoztatni.

☹ Az alkalmazás garantálja a kulcs egyediségét. Ha ez valamiért is sérülne felülírhatjuk az adatott.

A Cassandra rendszer egységnyi (aqtomikus) sor műveletek szintjén. Íráskor és olvasáskor is specifikálhatjuk, hogy milyen konzisztencia szintet óhajtunk. Általában a Cassandra egy előbb–utóbb konzisztens rendszer, ami szerint, ha egy ideig nem történek új elem beszúrások a rendszer automatikusan beáll konzisztens állapotba. De a konzisztens állapot sosem garantált, tehát gondoljuk meg, hogy milyen szintű lekérdezéseket kérünk.

2.2. Apache HBase

A HBase a Cassandrához hasonlóan ugyancsak felső szintű Apache projekt. Gyökerei 2003–ban indult a Google–nál a Google File System[3] megalkotásával. Célja volt egy olyan rendszer megalkotása amely:

- Egy olcsó és könnyen beszerezhető hardver klasztere támaszkodik.
- Képes óriás mennyiségű adat eltárolására.
- Belsőleg megoldja az adat replikációt a klaszter pontjai között.
- Adat folyam áramlatú olvasásra optimalizált.

Miután mindezt sikerült készült el csakugyan a Google–nál a MapReduce[4] amely egy *egyszerűsített adatfeldolgozó* amely képes nagy klaszter farmok felett is hatékonyan működni. A GFS + MapReduce páros most már:

- Az adat feldolgozáshoz hasznosítjuk az adat tárolásra. már úgyis meglevő fürt processzor készletét.
- Hatékony pár nagyméretű fájl kezelésére.
- Nem biztosit véletlenszerű adat elérést közel valós idősbben.
- Nem teszik lehetővé akár több millió apró fájl kezelését. E mögött az egyik ok, hogy egy–egy számítógép csak véges számú fájlt nyithat meg egyszerre.

De a Google Mail és Analytic termékük pont sok kis fájl használatát igényelte. Ezért a kutatás egy olyan adattároló terelődött, amely felhasználja a már meglevő infrastruktúrát, képes kis entitások tárolására és ezt a háttérbe nagy fájlokká rakja össze (így belefér a GFS–be). A gyors véletlenszerű merevlemez elérés érdekében valamilyen indexelési stratégiát vettünk be. Ugyanakkor használja fel a MapReduce–t is mint elosztott adatfeldolgozó rendszer.

Ehhez először is a Google lemondott az adattárház relációs jellegéről, majd öt kulcs fontosságú műveletre gyúrt: *írás* (create), *olvasás* (read), *frissítés* (update), *törlés*(delete) és *felderítés*(scan). Az eredményt 2006–ban a BigTable cikkben[5] tárgyalták: létrehoztak elosztott tároló rendszert strukturált adathalmazra. A rendszer fő tulajdonságai: az elosztottság, az adat lehet szórványos (ritka), perszisztens és több dimenziós rendezett szótár. Egy évvel később innen kiindulva alakult meg a HBase (Hadoop Database).

2.2.1. Adat modell

Az alap építőelem az oszlop[15]. Egy vagy több oszlop egy sort alkot. A sort egyértelműen megcímez egy egyedi azonosító. A sorok egy halmaza táblát alkot. Minden oszlopnak lehet több verziója, egy–egy ilyenre, mint cella vonatkozzunk. Két cellát egymás között egy automatikus vagy felhasználó által szolgáltatott idő bélyeg különböztet meg.

A sorok mindig lexikografikus módon vannak rendezve a kulcsuk szerint. Érdemes megemlíteni, hogy e miatt a számok nem numerikus sorrendbe lesznek rendezve. Például a 10 a 2 előtt jön. Amennyiben számokat akarunk rendezni, azokat egészítsük ki nullákkal, azaz 2 helyet 02–öt rendelünk hozzá a sor kulcsához. Két kulcs összehasonlítása bájt szinten történik, az első különböző bájtnál az összehasonlítás leáll. Így a sor kulcsa a relációs adatbázisokban már megismert fő indexként viselkednek.

Az oszlopok továbbá oszlop családba csoportosíthatók. Szerepük, hogy rengeteg HBase által szolgáltatott funkció oszlopcsalád szinten konfigurálható. Ilyen például az alkalmazott sűrítési metódus. De például az oszlopcsalád oszlopai mindig ugyanazon fájlba tárolódnak, így az oszlopcsaládok elvégzett lekérdezés elvégzéséhez biztos, hogy legfeljebb csak egy fájlt kell beolvasnunk egy adott gépen a merevlemezről.

A HBase által használt fájlt HFile–nak nevezzük, és perszisztens, illetve változhatatlan. A fájl tartalma a kulcs szerint rendezve van. A fájl egy blokk szekvencia, amelynek az indexe alkotja az utolsó blokkot. Így egy sor kiolvasásához legfeljebb két művelet szükséges. Egy, amelyik beolvassa az index blokkot és egy, amely az index szerint a következő sort. Az index blokk viszont bekerül a memóriába ekkor és a következő lekérdezésekkor már csak a sort kell beolvasnunk.

Az oszlop családokra igaz, hogy:

- 🙂 A család belsejében az oszlop számra nincs korlát.
- 😐 Struktúrája *család:minősítő* alakú.
- 😐 Definiálni kell a tábla létrehozásakor.
- 😐 Ritkán változnak.
- 🙁 Neve nyomtatható kell, hogy legyen.
- 🙁 Csupán pár tíz használható fel belőle, ha hatékonyan működő rendszert óhajtunk.

A cella értéke bájt sorozat formájában tárolódik. Annak értelmezése a felhasználó alkalmazás feladata. Egy HBase adat elérési útvonalát megadhatjuk, mint:

$$(\text{Tábla}, \text{Sor Kulcs}, \text{Oszlop család}, \text{Oszlop}, \textit{Idő bélyeg}) \mapsto \text{Érték}$$

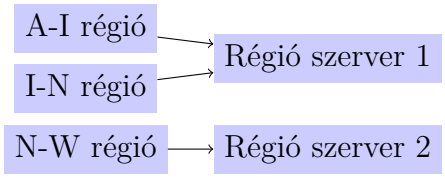
Ami adat struktúra szinten:

```
Map<Row, List < Map <Column, List<Value, Timestamp> > >
```

2.2.2. Adat partíció

A partíció központi eleme a régió. A régió nem más, mint egy folytonos sor halmaz, amely egy helyen van tárolva. Kezdetben egyetlen régió létezik. Amint egy régió átlépi az előre definiált határokat (írás, olvasás) a régió közepénél fogva automatikusan kettéválik, két új régióra. Két régió összes is olvadhat, ha azoknak mérete és kihasználtságuk kicsi lesz.

A régiók *régió szervereken* helyezkednek. Egy régió szerveren akár több régió is helyet kaphat, viszont egy régió csak egy régió szerveren fordul elő. Például attól függően, hogy milyen betűvel kezdődik a sor kulcsa a 2.7 ábrán láthatunk egy lehetséges leosztást.



2.7. ábra: Régiók és régió szerverek

Egy szerver általában 10–1000 régiót is elbír 1–2GB–os fájlokkal. Amennyiben egy szerver leesik, egy másik szervernek csap épp be kell olvasnia a replikált régiót, és máris szolgálhatja ki a kérést. Ezzel gyors hiba javító és automatikusan adat elosztó rendszert nyertünk. A HFile–t a HBase alatt levő elosztott fájlrendszerben, általában HDFS-ben tároljuk. Ez a Google File System–ből ihletve biztosit egy skálázható, perszisztens és replikázható tárolást.

A fájlba írt módisítások automatikusan replikázódnak, egy előre definiált számú fizikai szerveren keresztül. Amikor a kliens egy íráskéréssel érkezik, először is beírjuk a kérést egy napló fájlba (Write Ahead Log – *WAL*), majd beírjuk a memória tárolóba. Ha a memóriatároló elég nagy lesz, azt kiírhatjuk a háttér tárolóra. Ezután a kiírt bejegyzések törölhetők a napló fájlból.

A WAL fájl maga is a HDFS–ben tárolódik, ezáltal automatikusan többszörözve van. Ha netán kiesik egy szerver a WAL fájl alapján vissza tudjuk állítani a memória tárolót és minden folytatódik tovább. A HDFS fájl folyam optimalizált, ennek megfelelően a törlés költséges művelet lenne, így a benne tárolt fájlok változtathatatlanok.

Törlés parancsnál csupán egy törlésre megjelölés történik. Az olvasás ezzel a jelöléssel maszkozva van, így ha az be van állítva, akkor úgy látjuk, mintha az ott se lenne. A fizikai törlés egy periodikus tömörítési műveletnél hajtódik végre, amikor is a rendszer beolvassa a fájlt, és egy teljes új példányt létrehozva írja vissza. Hogy olvasáskor biztos konzisztens nézetett lássunk e merevlemez HFile–ja és a memóriatároló tartalmának uniója lesz megvizsgálva.

A HBase komponensei:

Kliens API Egy főleg JAVA csomag, de egyre több nyelvre is elkészül.

Mester szerver Feladata a terhelés elosztása: ennek érdekében menedzseli a régiók mozgatását és ugyanakkor tárolja a rendszer meta adat információit (séma leírás, változás és így tovább). Fontos, hogy nem tárol konkrét adatott és nem vesz részt az adat lekérdezés műveletében.

ZooKeeper Egy fájl rendszer jellegű rendszer, amellyel tulajdon viszonyt lehet egyeztetni, szolgáltatásokat tud nyilvántartani és különböző műveleteket felülügyelni. A Google Chubby rendszer ihlete[16].

Régió szerverek Változó csomópontok a rendszerben. Élettartamuk szesszió alapú. A ZooKeeper rendszerben periodikusam szív ütemszerűen jelezniük kell elérhetőségük. Ezt egy rövid időtartamú bejegyzéssel végzik el. A mester szerver folyamatosan figyeli e bejegyzéseket. Ez alapján térképezi fel a hálóztatott és innen tudja, ha egy gép kiesett. Ha ez megtörtént, szól egy régió szervernek, hogy vegye át a feladatott. A régió szerverek direkt kommunikálnak a kliensekkel és szolgálják ki az adatot.

A sorokon végzett művelet egységnyi idő alatt történik meg (atomikus). Az adat replikációs feladata delegálva van a HBase által használt elosztott fájl rendszernek, mint például a HDFS. Mivel egy sor egy időben csak egy fizikai szerveren helyezkedik el a rendszer erősen konzisztens.

3. fejezet

Összegzés

A szerző reménye, hogy e jegyzet oldalainak sikerült az olvasó részére egy alapos betekintést szolgáltatni az oszlopcsaládokba. Habár ezen irat magában kevés lesz ahhoz, hogy a Cassandra vagy HBase rendszert az olvasó használni tudja, jó képet szolgáltat ezek struktúrájáról.

A nagy adat világa itt van. Ezt tényként lehet kezelni. A szoftver mérnök feladata meghatározni, hogy hogyan tároljuk el e napról–napra növekvő adatmennyiséget. Választhatunk akár relációs, NoSQL, vagy esetleg NewSQL megoldást is feladatunkra. A fontos, hogy a feladathoz mérten tegyük meg az adattároló választást és minden körülmény paraméterét figyelembe vegyük ilyenkor.

A NoSQL alapú adatbázisok akkor válnak igazán erősé, ha nagy mennyiségű, esetleg relatív strukturálatlan, avagy gyakran változó struktúrájú adat érkezik a rendszerbe. Ilyenkor egy skálázható rendszert, biztosit, amely könnyedén ki tud használni egy szerverpark tároló és számítási képességét, ha az számunkra elérhető. A kulcsozó itt a többség, s nem az erősség. Sok, akkár gyengébb gépre épitűnk, s nem pár csúcs teljesítményűre.

Az ipart követve úgy a Hadoop (HBase), mint a Cassandra még hosszú éveken át el fog kísérni. A Cassandra előnye a könnyebb rendszer felállítás, menedzselés és SQL világra inkább emlékeztető CQL nyelvezet. Ennek megfelelően gyorsan teret hódit a kisebb vállalkozásoknál. A Hadoop (HBase) előnye, hogy az adatfeldolgozást delegálhatjuk az adatot tároló gépekre, a MapReduce rendszer segítségével. Az elmúlt években úgy az Amazon, mint a Microsoft is Hadoop-ot használ az Amazon Elastic Compute Cloude, illetve Microsoft Azure felhőszolgáltatásaikba.

Végezetül két tanáccsal szolgálnék, a bátor felhasználónak, aki belevág e rendszerek használatába; két dolgot tartson szem előtt: új gondolkodásmód és fiatalság. Készüljünk fel, hogy a dolgok nem úgy működnek, mint ahogy azt megszoktuk az SQL világban. E rendszerek fiatalsága miatt meg e rendszerrekre jellemző a gyakori változás; pozitív értelembe mikor új funkcióval bővülnek és negatívan, amikor egy újabb verzió megtöri a kompatibilitást az előbbivel.

Irodalomjegyzék

[1] E. F. Codd, "A Relational Model of Data for Large Shared Data Banks," *CACM*, vol. 13, no. 6, pp. 377–387, Jun. 1970.

[2] D. J. Abadi, P. A. Boncz, and S. Harizopoulos, "Column oriented database systems.," *PVLDB*, vol. 2, no. 2, pp. 1664–1665, 2009. [Online]. Available: http://dblp.uni-trier.de/db/journals/pvldb/pvldb2.html#AbadiBH09.

[3] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The Google File System," *SIGOPS Oper. Syst. Rev.*, vol. 37, no. 5, pp. 29–43, 2003, ISSN: 0163-5980. DOI: http://doi.acm.org/10.1145/1165389.945450.

[4] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, Jan. 2008, ISSN: 0001-0782. DOI: 10.1145/1327452.1327492. [Online]. Available: http://doi.acm.org/10.1145/1327452.1327492.

[5] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, "Bigtable: a distributed storage system for structured data," in *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI'06)*, 2006. [Online]. Available: http://labs.google.com/papers/bigtable.html.

[6] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: amazon's highly available key-value store," *SIGOPS Oper. Syst. Rev.*, vol. 41, no. 6, pp. 205–220, 2007, ISSN: 0163-5980. DOI: http://doi.acm.org/10.1145/1323293.1294281.

[7] E. A. Brewer, "Towards robust distributed systems.," in *Symposium on Principles of Distributed Computing (PODC)*, 2000. [Online]. Available: http://www.cs.berkeley.edu/~brewer/cs262b-2004/PODC-keynote.pdf.

[8] S. Gilbert and N. Lynch, "Brewer's conjecture and the feasibility of consistent available partition-tolerant web services," in *In ACM SIGACT News*, 2002, p. 2002. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.20.1495&rep=rep1&type=pdf.

[9] J. Browne. (Jan. 2009). Brewer's CAP Theorem, [Online]. Available: http://www.julianbrowne.com/article/viewer/brewers-cap-theorem.

[10] S. Edlich, A. Friedland, J. Hampe, and B. Brauer, *NoSQL: Einstieg in die Welt nichtrelationaler Web 2.0 Datenbanken*. Hanser Fachbuchverlag, Oct. 2010, ISBN: 9783446423558. [Online]. Available: http://amazon.de/o/ASIN/3446423559/.

[11] S. Edlich. (Jan. 2011). NOSQL Databases, [Online]. Available: http://nosql-database.org/.

[12] D. C. G. Prof. Dr. Marc H. Scholl, "Lecture: advanced database technologies," DBIS Group - Universität Konstanz.

[13] A. Lakshman and P. Malik, "Cassandra: a decentralized structured storage system," *SIGOPS Oper. Syst. Rev.*, vol. 44, no. 2, pp. 35–40, Apr. 2010, ISSN: 0163-5980. DOI: 10.1145/1773912.1773922. [Online]. Available: http://doi.acm.org/10.1145/1773912.1773922.

[14] *Apache cassandra 1.1 documentation*.

[15] L. George, *HBase: The Definitive Guide*, 1st ed. O'Reilly Media, 2011, ISBN: 1449396100. [Online]. Available: http://www.amazon.de/HBase-Definitive-Guide-Lars-George/dp/1449396100/ref=sr_1_1?ie=UTF8&qid=1317281653&sr=8-1.

[16] M. Burrows, "The chubby lock service for loosely-coupled distributed systems.," in *OSDI*, USENIX Association, 2006, pp. 335–350.