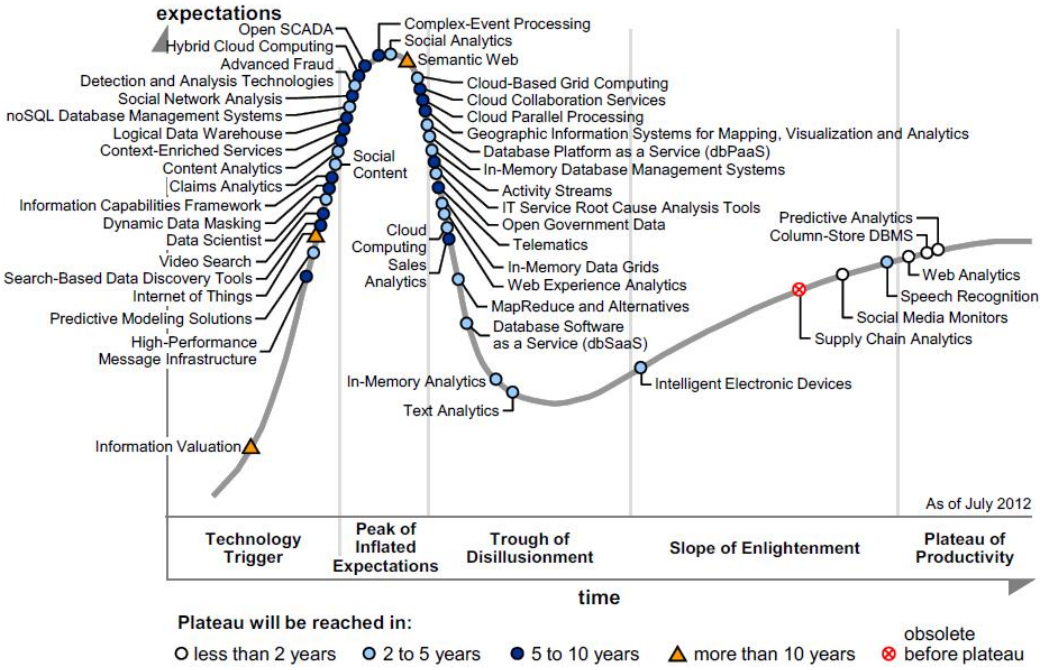
2. függelék: NoSQL adatbázis-kezelők[38]

Napjaink hatalmas adathalmazainak hatékony feldolgozása komoly kihívást jelent a mérnökök és az elemzők számára. Az IBM szerint napi 2,5 exabájt ($2{,}5 \cdot 10^{18}$ bájt) adat keletkezik – a növekedés olyan sebességű, hogy az elmúlt két évben jött létre a ma tárolt adatok 90%-a [1].
A relációs adatbázis-kezelő rendszerek évtizedeken át kielégítették a piac döntő részének műszaki és üzleti igényeit, azonban a 21. század elejétől olyan új igények fogalmazódtak meg (pl. web 2.0, számítási felhő, szenzorhálózatok, mobil eszközök által), melyek kiszolgálása túllépi a relációs rendszerek határait.
A világméretű internetes rendszerek rövid válaszidőket, magas rendelkezésre állást, nagy mennyiségű adat feldolgozását és horizontális skálázhatóságot kívántak meg. Ugyanakkor nem mindenhol követelmény a szigorúan vett adatbázis-konzisztencia, és számos esetben még valamennyi adatvesztés is tolerálható. Ezen igények kielégítésére jöttek – és jönnek – létre olyan új adatbázis-kezelő rendszerek, melyekre számos esetben NoSQL rendszerekként hivatkoznak. Az információs technológia ugyancsak támogatta a fejlődést: a számítógépes hálózatok adatátviteli sebessége nagyságrendekkel növekedett, fejlődtek az elosztott technológiák, jelentősen csökkentek a háttértárak és a memóriamodulok fajlagos költségei.



4. ábra: a „big data” technológiák és trendek életciklusgörbéje [2]

A Gartner *életciklusgörbéken* (hype cycle), ábrázolja az egyes trendek, technológiák érettségét és elterjedtségét. A nagy adathalmazok tárolását, feldolgozását és elemzését jelentő „big data” kifejezés 2011-ben került fel a görbére. 2012-ben a big data olyannyira aktuálissá és népszerűvé vált, hogy külön életciklusgörbét készítettek számára,

[38] Barabás-Szárnyas-Gajdos: NoSQL adatbázis-kezelők, BME-TMIT belső tanulmány, 2012. alapján.

amelyen a NoSQL adatbázis-kezelő rendszereket a felfokozott várakozások csúcsának közelébe helyezték [2].
A big data kifejezésnek nincs egységesen elfogadott definíciója, általában az alábbi három tulajdonsággal jellemzik [3]:

- *Mennyiség* (volume): az eddig megszokottnál nagyságrendekkel megnövekedett adatmennyiség.
- *Sebesség* (velocity): az adatok nagy sebességgel generálódnak, a rendszertől minél gyorsabb feldolgozást és visszacsatolást várnak el.
- *Változatosság* (variety): az adatok többféle forrásrendszerből érkeznek, lazán struktúráltak és gyakran kérdéses minőségűek.

A NoSQL rendszerek alapjául szolgáló technikák többnyire nem új keletűek. A ’70-es években készítették az első ún. *kulcs-érték tárolót* (key-value store) és a hálós adatbázis-kezelőket.
A napjainkban (2012.) legelterjedtebb NoSQL rendszerek fejlesztése 2006 és 2009 között kezdődött. A modern NoSQL rendszerek elméletében a Google játszott úttörő szerepet: 2004-ben bemutatták a MapReduce keretrendszert [4], majd 2006-ban a BigTable adatbázis-kezelőt [5] és az elosztott zárolást megvalósító, a BigTable-ben is alkalmazott Chubby protokollt [6]. Ezután sorra kerültek bejelentésre a nagy internetes vállalatok – így a Yahoo!, az Amazon, a Facebook és a LinkedIn – saját rendszerei.

Elnevezés

A NoSQL kifejezés a 2009-ben megtartott nyílt forráskódú, elosztott adatbázisokkal foglalkozó no:sql(east) konferenciát követően terjedt el [7].
Fontos megjegyezni, hogy az SQL (relációs) és a NoSQL rendszerek közötti határ nem éles, ugyanis számos NoSQL adatbázis-kezelő a relációs adatmodellen alapul, ill. ACID tranzakciókat kínál. A NoSQL közösség a nevet not only SQL-ként oldja fel, arra utalva, hogy ezek a rendszerek nem a relációs adatbázisok ellenpólusai, hanem azok mellett, azokat kiegészítve működnek. A NoSQL adatbázis-kezelőket gyakran hívják *poszt-relációs* (post-relational) vagy *nem-relációs* (non-relational) rendszereknek [8]. A NoSQL rendszereket jellemzően nem önmagukban, hanem többféle, különböző adatmodellen alapuló tárolási technológiákkal együtt használják, ezt *többnyelvű perzisztenciának* (polyglot persistence) nevezik [9].

A http://nosql-database.org/ szerint a NoSQL mozgalom eredeti célja modern, jól skálázódó adatbázis-kezelő rendszerek készítése [10]. A legtöbb NoSQL rendszer az alábbi szempontok szerint készült:

- nem-relációs adatmodell,
- elosztott működés,
- nyílt forráskód,
- horizontális skálázhatóság.

A definíció nem szigorú: nem szükséges, hogy egy rendszerre mindegyik tulajdonság igaz legyen. A NoSQL rendszerek további gyakori tulajdonságai:

- sémamentesség vagy gyenge séma kényszerek,
- replikáció támogatása,

- *egyszerű alkalmazásprogramozási interfész* (application programming interface, API),
- *fokozatos konzisztencia* (eventual consistency).

A konzisztenciával és skálázhatósággal kapcsolatos feltételeknek a továbbiakban külön alfejezeteket szentelünk, és külön foglalkozunk a replikációval is.
A nyílt forráskódúság feltétele vitatott, hiszen ez alapján nem tartozhatna például a NoSQL adatbázis-kezelők közé a már említett úttörő, a Google BigTable. Bár a legtöbb rendszer egyszerű API-val rendelkezik, komplex lekérdezéseket jellemzően egyszerűbb megfogalmazni SQL nyelven, mint a NoSQL rendszerek sajátos keretei között. Ezért néha olyan rendszereknél is megjelenik az SQL vagy ahhoz hasonló lekérdezések támogatása, ahol ezt az adatmodell nem indokolja.

Skálázhatóság

A bevezetőben említett adatmennyiségek mellett az igények kiszolgálása már nem lehetséges (vagy nem praktikus) egyetlen szervergéppel, ezért az ilyen rendszerek tervezésénél már fő szempont, hogy azok több számítógépen elosztottan működjenek. Tekintettel arra, hogy esetenként több tízezer számítógépről van szó, kritikus kérdés, hogy hogyan viselkedik a rendszer újabb és újabb erőforrások (számítógépek) hozzáadása következtében.
A skálázhatóságnak nincs általánosan elfogadott definíciója [11]. Informálisan egy rendszert akkor nevezünk *skálázható*nak (scalable), ha az erőforrásait növelve, a rendszer teljesítménye a hozzáadott erőforrásokkal *arányosan* javul. A teljesítmény jelentheti a rendszer által egyszerre feldolgozható lekérdezések számát, a feldolgozott adathalmazok vagy adatelemek méretét, a késleltetést stb. [12].
A plusz erőforrások hozzáadásának másik oka a rendszer hibatűrésének, ill. rendelkezésre állásának növelése. Fontos követelmény, hogy ez ne jelentsen túl nagy többletterhelést a rendszer számára. A skálázhatóság két fő típusát különböztetjük meg.

- ***Vertikális skálázhatóság* (vertical scalability, scale up):** a rendszer kiválasztott elemét bővítjük új erőforrással, leggyakrabban erősebb/több processzorral vagy több memóriával. A módszer előnye, hogy egyszerű megvalósítani, és a megfelelő, a rendszer szűk keresztmetszeteit okozó erőforrások bővítése esetén biztosan teljesítménynövekedéssel jár. Ezt az elvet egy- és többszerveres elosztott rendszerekben is lehet alkalmazni.
- ***Horizontális skálázhatóság* (horizontal scalability, scale out):** a rendszert bővítjük új számítógéppel. Előnye, hogy sok, olcsó gépből nagy teljesítmény érhető el. Hátránya, hogy elosztottsága miatt többféle meghibásodás is felléphet, valamint bonyolult szoftvert igényel. Ezt támogatja a MySQL Scale-Out, az Oracle Real Application Cluster (RAC) és a legtöbb NoSQL rendszer is.

A skálázhatósággal szorosan összefüggő szempont, hogy a rendszer komponensei milyen közös adattárat használnak. Ez alapján három fő elosztott architektúrát különböztetünk meg [13], melyek egyre lazábban csatolt rendszereket definiálnak:

- ***Megosztott memória* (shared memory):** a komponensek közös memórián dolgoznak. Ilyen architektúrát egy számítógépen belül használnak, pl. többprocesszoros rendszerekben.

- ***Megosztott lemez* (shared disk):** a komponensek saját memóriával rendelkeznek és közös lemez(ek)re mentenek. Ilyenek a közös háttértáron dolgozó adatbázis-kezelők, pl. az Oracle RAC.
- ***Megosztás nélküli* (shared nothing):** a komponensek nem rendelkeznek sem közös memóriával, sem közös lemezzel.

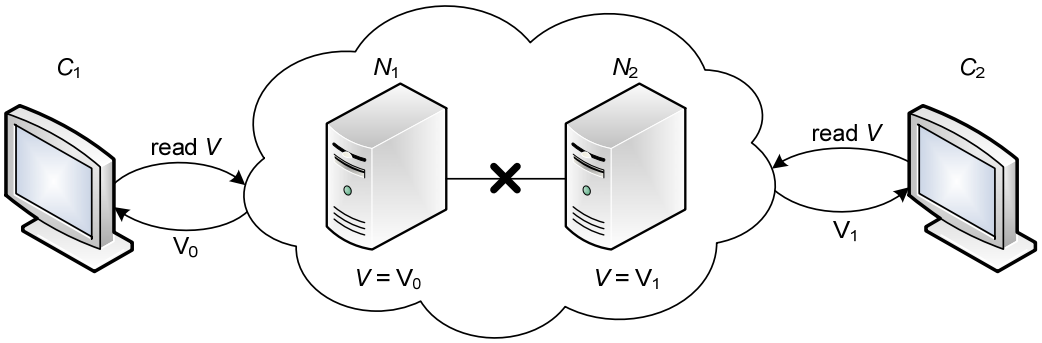
Természetesen egy rendszer annál jobban skálázható, minél kevesebb megosztott komponenssel rendelkezik. Ezért a NoSQL rendszerek jellemzően megosztás nélküli architektúrájúak.

A CAP-tétel

A NoSQL rendszerek tervezését befolyásoló egyik legfontosabb eredmény az ún. CAP-tétel, amely az elosztott rendszerek által nyújtott garanciákra ad formális korlátot.
Eric Brewer, a Berkeley Egyetem professzora 1999-es publikációjában definiálta (informálisan) a CAP tulajdonságokat és a CAP alapelvet [14], amelyet 2000-ben a PODC (Principles of Distributed Computing) szimpóziumon ismertetett CAP-sejtés néven [15]. A sejtés szerint egy webszolgáltatás a CAP rövidítésben szereplő *konzisztencia* (consistency), *rendelkezésre állás* (availability), *partíció tolerancia* (partition tolerance) tulajdonságok közül egyidejűleg legfeljebb kettőt garantálhat teljesen.
A sejtést Seth Gilbert és Nancy Lynch, az MIT (Massachusetts Institute of Technology) kutatói formalizálták és bizonyították be 2002-ben [16]. Ebben a formájában a CAP-tétel nem csak webszolgáltatásokra, hanem minden elosztott rendszerre érvényes. Az alábbiakban definiáljuk a három tulajdonságot, majd ismertetjük a tétel bizonyításának vázlatát.

Konzisztencia

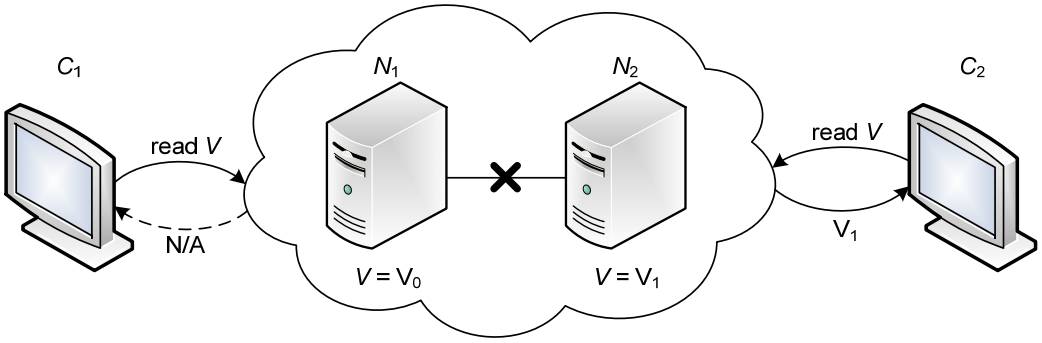
Egy elosztott rendszer akkor konzisztens, ha bármely időpillanatban egy adategység értékét bármely csomóponttól lekérdezve ugyanazt az értéket kapjuk.
Fontos, hogy az elosztott rendszerek konzisztenciája nem egyezik meg az tranzakciókezelésnél használt ACID (atomicity, consistency, isolation, durability) tulajdonságokban definiált konzisztenciával [16] [17]. Az ACID konzisztencia definíciója azt garantálja, hogy az egyes *tranzakciók* az adatbázist konzisztens állapotból konzisztens állapotba viszik át, ahol az adatbázison mindig csak a sikeresen lefutott tranzakciók eredménye látszik. Itt a konzisztencia a *rendszer* tulajdonsága. Gilbert és Lynch az említett bizonyításukban az *atomi* (atomic) kifejezést használták (szintén nem tévesztendő össze az ACID tranzakciókra vonatkozó azonos nevű tulajdonságával). Definíció szerint atomi konzisztencia esetén az elosztott rendszernek külső szemlélő számára úgy kell tűnnie, mintha a műveleteit egy csomóponton, sorban egymás után hajtotta volna végre, azok átlapolódása nélkül. Ehhez léteznie kell a műveletek egy olyan teljes sorrendezésének, ahol a külső szemlélő számára úgy tűnik, hogy minden művelet „egy pillanat alatt” végrehajtódott.



5. ábra: A *V* adategység nem konzisztens, a C_1 és a C_2 kliensek eltérő értéket látnak

Rendelkezésre állás

Egy elosztott rendszer rendelkezésre áll, ha minden működő csomóponthoz érkező kérésre válaszol, tehát a csomópontokon futtatott algoritmusoknak véges idő alatt be kell fejeződniük. A formális definíció nem korlátozza a futási időt, de később látni fogjuk, hogy a gyakorlati alkalmazások során a késleltetésnek kiemelt szerepe van.
A 6. ábrán egy olyan rendszer látható, amely garantálja a konzisztenciát. A hálózat két részre szakadt, ezért az N_1 csomópont nem tudja kiszolgálni a C_1 kliens kérését.



6. ábra: A C_1 kliens számára a *V* adategység nem elérhető

Partíció tolerancia

A hálózat partíciója annak alapján modellezhető, hogy a hálózat egyik csomópontjából a másikba küldött üzenetekből tetszőleges számú elveszhet. Egy nem összefüggő hálózatban a hálózat egyik komponenséből a másikba küldött minden üzenet elveszik. Minden üzenetvesztési szekvencia modellezhető a hálózat ideiglenes partíciójával, majd újraegyesülésével. A partíció oka lehet a hálózat hibája, valamint a csomópontok hardveres vagy szoftveres hibái. A partíció tolerancia a másik két tulajdonság tükrében értelmezhető:

- A konzisztencia követelményhez azt kell teljesíteni, hogy a rendszer műveletei akkor is atomiak legyenek, ha az azokat megvalósító algoritmusok üzeneteiből tetszőleges számú elveszhet.
- A rendelkezésre álláshoz azt kell biztosítani, hogy a csomópontok *minden*, a kliensektől érkező *kérésre* érvényes választ adjanak, annak ellenére, hogy tetszőlegesen sok üzenet elveszhet.

Egy rendszer tehát akkor partíció toleráns, ha a kérésre hálózati partíció esetén is helyes[39] választ ad (kivéve a teljes hálózat kiesésének esetét). Ha egy rendszer nem partíció toleráns, a hálózat partíciója esetén semmilyen garanciát nem nyújt a konzisztenciára és a rendelkezésre állásra.
A partíció tolerancia elengedhetetlen a nagyméretű elosztott rendszerben, hiszen a hálózat partíciójának valószínűsége a gépek számának növekedésével egyre magasabb lesz.

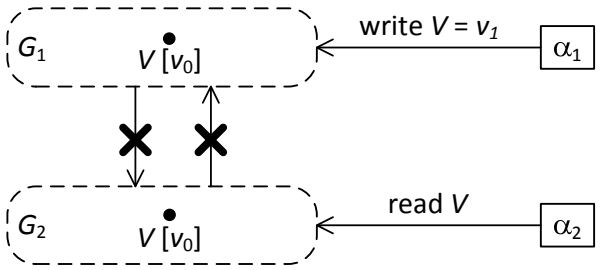
A CAP-tétel formálisan

A CAP-tétel röviden úgy fogalmazható meg, hogy elosztott rendszerben a hálózat partíciója esetén a rendszer műveletei nem lesznek atomiak és/vagy az adategységei elérhetetlenek lesznek.
Formálisan: egy aszinkron vagy részben szinkron[40] hálózaton működő elosztott rendszerben nem biztosítható, hogy a rendszer *mindig* (üzenetek elvesztése esetén is) garantálja az alábbi tulajdonságokat:

- rendelkezésre állás,
- atomi konzisztencia.

Bizonyítás

A bizonyítást aszinkron hálózatra mutatjuk meg, Gilbert és Lynch cikke tartalmazza a részben szinkron hálózatra vonatkozó bizonyítást is [16]. Az aszinkron hálózati modellben nincs óra és az egyes csomópontok döntéseit kizárólag a beérkezett üzenetek és a lokális számításaik befolyásolják [18].
Indirekt módon bizonyítunk. Tegyük fel, hogy létezik olyan A algoritmus, amely teljesíti a CAP tulajdonságokat. Feltételezhetjük, hogy a hálózat legalább két csomópontból áll, így felosztható két diszjunkt, nem üres halmazra. Legyen a felosztás (G_1, G_2). Tegyük fel, hogy hálózati partíció miatt G_1 és G_2 között minden üzenet elveszik, valamint G_1-hez és G_2-höz nem fordul más kliens. Megmutatjuk, hogy ha G_1-ben írunk, akkor egy későbbi, G_2-beli olvasás művelet nem térhet vissza a korábbi írás művelet értékével.



7. ábra: G_1 és G_2 részekből álló hálózati partíció

Mindkét halmazban tároljuk a V adategység értékét, ami kezdetben v_0.

[39] Érvényes (helyes) válasz alatt azt értjük, hogy a rendszer végrehajtja a kért írást vagy olvasást (a konzisztencia, tehát a művelet atomicitásának garantálása viszont nem kötelező). A „nem elérhető" jellegű hibaüzenetek nem számítanak érvényes válasznak.
[40] A *részben szinkron* hálózati modellben a csomópontok azonos ütemben növekvő órákkal rendelkeznek. Az órák azonban nem teljesen szinkronizáltak.

Legyen α_1 az A algoritmus olyan lefutásának első lépése, amiben egy írás G_1-ben V értékét a v_0-tól különböző v_1 értékre állítja. A rendelkezésre állás tulajdonság miatt tudjuk, hogy az írás sikeresen befejeződik.
Hasonlóan, legyen α_2 egy olyan lefutás első lépése, amiben G_2-ben V értékét olvassuk. A rendelkezésre állás miatt az olvasásnak vissza kell térnie egy értékkel. Amennyiben α_2 előtt más műveletet nem futattunk, ez az érték biztosan v_0 lesz.
Legyen α az algoritmus egy olyan lefutása, ahol α_1 után lefut α_2. A G_1 és G_2 közötti kommunikáció hiánya miatt a G_2-beli csomópontok számára α megkülönböztethetetlen α_2-től, míg α_1 nem fordul G_2-beli csomópontokhoz.
A fentiek miatt α lefutása során az α_2-beli olvasás művelet v_0-t fog visszaadni. α_2 olvasása azonban csak akkor kezdődhet meg, ha α_1 írása befejeződött, így a konzisztencia tulajdonság nem teljesül, hiszen V értéke G_1-ben már v_1. Ezért nem létezhet olyan algoritmus, ami a CAP tulajdonságok közül mindhármat teljesíti.

Triviális esetek

A CAP-tétel alapján az alábbi esetek adódnak egy elosztott rendszer tulajdonságaira.

- Konzisztens és rendelkezésre álló (CA): megpróbáljuk megszüntetni a partíció lehetőségét, például úgy, hogy a rendszert egy gépen futtatjuk. Ezzel a hálózati partíció lehetőségét kizártuk, de részleges hibák előfordulhatnak. Természetesen a hálózati hibákra való érzékenység miatt ezek a rendszerek csak korlátozottan skálázódhatnak, ezért általában megbízható LAN hálózatokon üzemeltetik őket. Ezt az architektúrát alkalmazza a napjainkban használt RDBMS-ek többsége.
- Konzisztens és partíció toleráns (CP): hálózati partíció fellépése esetén azok adategységek elérhetetlenné válnak, amelyekre nem a rendszer nem tudja biztosítani a műveletek atomi végrehajtását. Ennek ellenőrzése és az egyes csomópontok újra beléptetése meglehetősen összetett lehet. Elméleti szempontból CP az a triviális rendszer, amely nem dolgozza fel a beérkező üzeneteket, azaz a rendszer mindig elérhetetlen.
- Rendelkezésre álló és partíció toleráns (AP): a konzisztencia gyengítésével egyidejűleg elérhető partíció tolerancia és rendelkezésre állás is. Bizonyos alkalmazások (pl. keresők, közösségi oldalak) nem feltétlenül igényelnek erős konzisztenciát, más alkalmazásoknál a konzisztencia megvalósítható az alkalmazási rétegben is [17].

Fontos, hogy a fenti triviális esetekkel nem fedhető le az összes (elosztott) adatbáziskezelő.

Mindhárom CAP tulajdonság garantálása

Brewer prezentációjában úgy fogalmazott, hogy egy rendszer a CAP tulajdonságokból legfeljebb kettőt garantálhat [15]. Ez a korlátozás azonban csak egy adott időpillanatra érvényes, nem a rendszer működésének egészére.
A CAP-tétel nem zárja ki ugyanis azt, hogy egy rendszer jól működő hálózat esetén konzisztens és rendelkezésre álló legyen, hálózati partíció esetén pedig valamelyik (esetleg mindkettő) tulajdonságra gyengébb garanciát nyújtson. Ezért a rendszereket célszerű úgy vizsgálni, hogy a hálózat állapotától függően milyen tulajdonságokat garantálnak.

A gyakorlati rendszerek vizsgálatánál célszerű a késleltetés csökkentésére irányuló kompromisszumokat is figyelembe venni [20].

A CAP-tétel kritikája

A CAP-tétel jelentős eredmény az elosztott rendszerek elmélete terén, ugyanakkor több szempontot figyelmen kívül hagy.
Ilyenek például olyan kritikus rendszertervezési kérdések, mint a rendszer teljesítménye (áteresztőképessége) és késletetése, valamint az *egyszeres hibapont*ok (SPOF, Single Point of Failure) jelenléte. A tétel szintén nem érvényes alkalmazáshibák, az adatbázis-kezelő összeomlását okozó tranzakciók esetén (amelyek a hálózat más csomópontján futtatva is összeomlást fognak okozni).

Késleltetés

A valós alkalmazások során kiemelt szerepe van a késleltetésnek is. Kísérletek kimutatták, hogy az Amazonnál a lapbetöltési idő minden 100 ms-os növekedése 1%-kal csökkentette az eladásokat, míg a Google-nél a keresési találatok megjelenési idejének fél másodperces növekedése 20%-os bevételcsökkenést okozott [19].
A késleltetés csökkentésének érdekében sok NoSQL rendszer (pl. az Amazon Dynamo [21]) összefüggő hálózaton futtatva sem garantál atomi konzisztenciát, cserébe a rendszer hálózati partíció kialakulása esetén is elérhető marad. Ezek a rendszerek tehát csak az *AP* tulajdonságokat garantálják. A megközelítés BASE (Basically Available, Soft-state, Eventually consistent) néven ismert [23]. A BASE kifejezést szintén Brewer alkotta meg [22], azonban ő is elismeri, hogy a kifejezés – az ACID-hoz hasonlóan – nem precíz, formális definíció, hanem egy jól hangzó rövidítés a gyengébb konzisztencia követelményekre[41] [17].
Léteznek olyan rendszerek is (pl. a GenieDB [24]), amelyek normál működés során erős konzisztenciát nyújtanak, partíció esetén viszont gyengébb konzisztencia kritériumok mellett folytatják a működést, így a hálózat állapotától függően *CA* és az *AP* tulajdonságpárokat teljesítik.
Az elosztottan működő, ACID tranzakciókat garantáló rendszerek, többek között a VoltDB RDBMS és az OrientDB NoSQL rendszer, normál működés során konzisztensek és rendelkezésre állók, hálózati partíció esetén pedig a rendelkezésre állást csökkentik a konzisztencia megtartása érdekében. Ezek a rendszerek a *CA* és a *CP* tulajdonságpárokat teljesítik.
Eltérő megközelítést alkalmaz a Yahoo! több kontinensen elosztva működő PNUTS (Platform for Nimble Universal Table Storage) rendszere [25]. A PNUTS normál működés során az atominál gyengébb konzisztenciát[42] nyújt az alacsony késleltetés érdekében, hálózati partíció esetén viszont ugyanazt a konzisztencia garanciát nyújtja, és szükség esetén a rendelkezésre állást csökkenti. A PNUTS tehát partíció esetén sem az *A*, sem a *C* tulajdonságot nem teljesíti, annak érdekében, hogy a szerverek közötti fizikai távolság ellenére alacsonyan tarthassa a késleltetést.

[41] Brewer a *sav-bázis* (acid–base) reakcióban szereplő kifejezések alapján választotta a BASE rövidítést, mert konzisztencia szempontból a BASE a megszokott, elvárt ACID ellentétének tekinthető.
[42] A PNUTS egy speciális, a sorosíthatónál gyengébb, a fokozatosnál erősebb garanciákat nyújtó konzisztenciamodellt használ (a konzisztenciamodellek definícióit ld. később).

Konzisztenciamodellek

A konzisztenciamodell az adattár és az ahhoz hozzáférő folyamatok között létrejött megállapodás, amely szerint ha a folyamatok betartanak bizonyos szabályokat, az adattár helyesen fog működni [26]. Másképp, a konzisztenciamodell meghatározza a frissítések láthatóságára és látszólagos sorrendjére vonatkozó szabályokat [31]. A konzisztenciamodelleket két fő csoportra oszthatjuk.

- A *kliensközpontú* (client-centric) modell a kliensek és a rendszer közötti üzenetváltások alapján ad garanciát a konzisztenciára, ezért a fejlesztők számára praktikusabb definíciót nyújt.
- Az *adatközpontú* (data-centric) modell leírja, hogy az adategységek frissítései milyen korlátozásokkal terjednek az egyes szerverek között (ezért gyakran szerveroldali konzisztenciaként hivatkoznak rá [27]).

Kliensközpontú konzisztenciamodellek

A konzisztenciát illetően többféle kompromisszum is megengedhető, az alábbiakban a leggyakrabban használtakat ismertetjük. Fontos, hogy az alábbi konzisztenciamodelleket *egy adategységre* értelmezzük. A több adategységre értelmezett konzisztencia tranzakciókkal valósítható meg. A tranzakciók elosztott rendszerekben költségesek, ezért sok NoSQL rendszer csak korlátozottan vagy egyáltalán nem támogatja őket.

Gyenge konzisztencia (weak consistency)

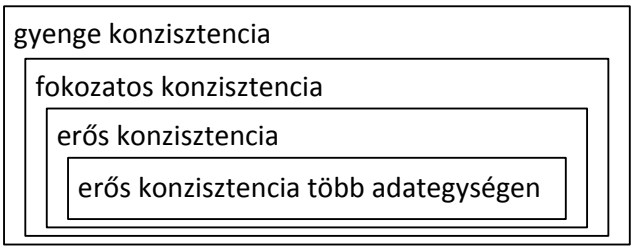
A rendszer nem garantálja, hogy az írást követő olvasások a legutoljára beírt adatot érik el. Az írás és azon pillanat között eltelt időt, amíg nem garantálható, hogy minden megfigyelő a frissített adatot látja, *inkonzisztenciaablaknak* (inconsistency window) nevezzük.

Fokozatos konzisztencia (eventual consistency)

A gyenge konzisztencia egyik típusa. A rendszer garantálja, hogy ha nincsenek további frissítések, előbb-utóbb minden olvasás a legutóbbi írás értékét éri el. Ha nem lépnek fel hibák, az inkonzisztenciaablak mérete meghatározható bizonyos tényezők alapján (pl. kommunikációs késleltetés, a rendszer terheltsége, a replikák száma stb.). A legismertebb fokozatos konzisztenciára építő rendszer a DNS (Domain Name System), ahol a gyorsítótárak előre meghatározott időközönként frissülnek.

Erős konzisztencia (strong consistency)

Minden olvasás művelet az adategységen legutóbb befejezett írás művelet eredményével tér vissza, függetlenül attól, hogy az adategységet melyik csomóponton éri el. Ez a konzisztenciamodell könnyen érthető a rendszer felhasználói számára, és jelentősen egyszerűsíti a kliensalkalmazások fejlesztőinek munkáját.
Erős konzisztencia megvalósításához egy adott adategység összes műveletét egy csomóponton kell végrehajtani vagy megfelelő elosztott protokollt kell használni.
Több adategységen végzett műveletek esetén erős konzisztenciát tranzakciókezeléssel biztosíthatunk, ami azonban elosztott környezetben csak költségesen valósítható meg.



8. ábra: Kliensközpontú konzisztenciamodellek.
A szűkebb halmazok modelljei erősebb garanciákat nyújtanak.

A fokozatos konzisztencia többféle módon gyengíthető, a gyakori konzisztencia-modellek az alábbiak [26] [27].

Monoton olvasási konzisztencia **(monotonic read consistency, MR)**

Ha egy folyamat beolvas egy *x* adatelemet, akkor az *x* adatelem minden további olvasásának ugyanazt vagy frissebb értéket kell szolgáltatnia.

Példa: legyen a rendszer egy elosztott emailadatbázis, amelyben a felhasználó postaládáját elosztott módon, többszörözve tárolják. Ha felhasználó az egyik városban elolvassa a leveleit, majd átmegy egy másik városba, a monoton olvasás biztosítja, hogy a korábban olvasott levelek a másik városból csatlakozva is a fiókjában lesznek.

Monoton írási konzisztencia **(monotonic write consistency, MW)**

Adott folyamat által végrehajtott, *x* adatelemet módosító műveletnek be kell fejeződnie, mielőtt ugyanez a folyamat újabb írási műveletet hajtana végre az *x* adatelemen. Azaz a rendszer garantálja, hogy egy folyamat az írásai sorosan végzi.

Példa: egy verziókezelés alatt álló programkód esetén az írások során csak a programkód egy része változik. Ilyenkor elengedhetetlen, hogy egy fejlesztő módosításai a megfelelő sorrendben kerüljenek be a rendszerbe. Az MW-t nem biztosító elosztott rendszerek komoly kihívások elé állítják az alkalmazásfejlesztőket [27].

„Olvasd az írásod" konzisztencia **(read your writes consistency, RYW)**

Adott folyamat által az *x* adatelemen végrehajtott írási művelet eredményének mindig láthatónak kell lennie a folyamat által *x* adatelemen végrehajtott későbbi olvasási művelet számára.

Példa: Sok elosztott webes szolgáltatás (pl. a Gmail) RYW konzisztenciát nyújt. RYW konzisztencia hiányában a felhasználó számára hibásnak tűnhet a rendszer működése. Előfordulhat például, hogy a felhasználó megváltoztatja a jelszavát, de a változás nem terjed rögtön végig a rendszeren (pl. a jelszavakat egy erre a célra dedikált szerveren tárolják és időre van szükség ahhoz, hogy a többi szerverhez ez eljusson), így egy ideig nem tud belépni az új jelszavával.

„Írás után olvasás" konzisztencia **(writes follow reads, WFR)**

Adott folyamat által az *x* adatelemen végrehajtott írási művelet, amely az *x* adatelemnek a folyamat által korábban történt olvasását követi, feltétlenül ugyanazt az értéket vagy annál frissebbet állít be, mint amit korábban olvasott.

Példa: legyen az elosztott rendszer egy bibiliográfiai adatbázis, amelybe a felhasználók tudományos cikkek adatait tölthetik fel. Egy felhasználó olyan bejegyzést talál,

ahol a cikk valamelyik adata (pl. az oldalszám) hibás és ezt a mezőt javítja. A WFR konzisztencia követelménye szerint az írás csak akkor jelenik meg a szerveren, ha az alapjául szolgáló bejegyzés már megjelent rajta. Így garantálható, hogy az írás minden szerveren az olvasott (és javított) bejegyzéshez tartozó értékeket állítja be [26] [28].

Az MR, MW, RYW és WFR modelleknek többféle kombinációja is elképzelhető, amelyekkel eltérő teljesítményű és konzisztenciájú rendszerek definiálhatók.

Adatközpontú konzisztenciamodellek

Az adatközpontú konzisztenciamodellek [26] alapján. Az erősebb garanciákat nyújtó modellektől haladunk a gyengébbek felé.

Szigorú konzisztencia (strict consistency)

Adott adatelemen végrehajtott bármely olvasási művelet az ugyanezen az adatelemen végrehajtott legutolsó írási művelet értékével tér vissza. Ez a legerősebb adatközpontú konzisztenciamodell.
A definícióban szereplő „legutolsó” kitétel megvalósításához abszolút globális idő meghatározására lenne szükség, ezért elosztott rendszerben a szigorú konzisztencia megvalósítása gyakorlatilag lehetetlen. A többi adatközpontú konzisztenciamodell úgy enyhítjük a feltételeket, hogy definiáljuk az ütköző műveletek esetén még elfogadott viselkedést.

Lineáris konzisztencia (linearizability)

A szigorú konzisztenciánál gyengébb modell. A modellben minden műveletet egy globális, de véges pontosságú óra által kiosztott időbélyeggel látunk el. $t_{OP}(x)$ az x adatelemen végrehajtott OP művelethez rendelt időbélyeg, ahol OP lehet írás (R) vagy olvasás (W). Az adattár akkor lineáris konzisztenciájú, ha az alábbi feltételeket teljesíti:

1. Bármely futás eredménye ugyanaz, mintha az adattáron dolgozó összes folyamat minden írási és olvasási műveletét meghatározott sorrendben hajtanánk végre, megőrizve bármely adott folyamat saját műveleteinek sorrendjét.
2. Ha bármely $OP1(x)$, $OP2(y)$ műveletpárra $t_{OP1}(x) < t_{OP2}(x)$, akkor a műveletsorban $OP1(x)$-nek meg kell előznie $OP2(y)$-t.

Soros konzisztencia (sequential consistency)

Egy adattár sorosan konzisztens, ha a lineáris konzisztencia 1. feltételét teljesíti. A lineáris és a soros konzisztencia közötti különbség, hogy utóbbi nem tesz semmilyen megkötést a fizikai időre vonatkozóan. Soros konzisztenciánál egyidejűleg futó folyamatok esetén a konkurens írási és olvasási műveletek bármilyen sorrendje elfogadható, de mindegyik folyamatnak ugyanazt a végrehajtási sorrendet kell észlelnie.

Megjegyzés: a definíciókból következik, hogy ha egy rendszerben minden tranzakcióban csak egyetlen írás vagy egyetlen olvasás művelet szerepel, a tranzakciókezelés *sorosíthatóság* (serializability) fogalma megegyezik a soros konzisztenciával [32].

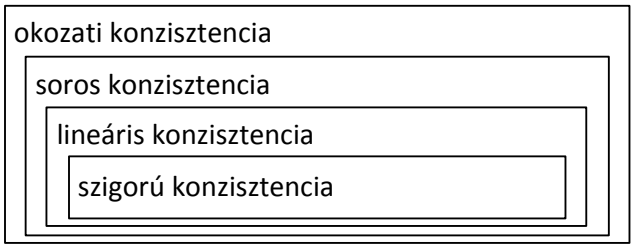
***Okozati konzisztencia* (causal consistency)**

Az okozati konzisztenciához be kell vezetnünk egy további definíciót. Két esemény *oksági viszonyban* (causally related, causality) van, ha:

- két írás ugyanazon az adategységen dolgozik,
- egy folyamatban egy olvasás művelet után írás következik (akár különböző adategységekre),
- egy olvasás egy írás eredményét adja vissza (az írás történhet bármely folyamatban) vagy
- két művelet a fentiek bármely kombinációjában (tranzitívan) függ egymástól.

Okozati konzisztencia: a potenciálisan oksági viszonyban lévő eseményeket a rendszer minden csomópontja ugyanabban a sorrendben látja. Azaz, ha egy *B* esemény egy korábbi *A* eseményből következik, vagy eredményét a korábbi *A* esemény befolyásolja, mindenki először az *A* eseményt látja és csak utána *B*-t.

Belátható, hogy ha egy rendszerben a (kliensközpontú) MR, MW, RYW és WFR konzisztenciamodellek feltételei teljesülnek, az okozati konzisztencia is fennáll [29]. Egyes szerzők ezért az okozati konzisztenciát kliensoldali modellként definiálják [27] [30].



9. ábra: Adatközpontú konzisztenciamodellek.
A szűkebb halmazok modelljei erősebb garanciákat nyújtanak.

Elosztott tárolás

A *replikáció* (replication) azonos adategység többszörözését jelenti – ugyanazt az adategységet különböző szervereken, replikálva tárolják. A *sharding* különböző adategységek különböző szervereken tárolását jelenti. A két technológia tehát működhet külön-külön és együtt is.

Replikáció

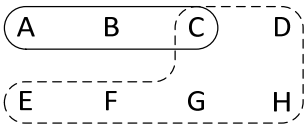
Replikáció esetén fontos feladat a konzisztencia biztosítása. Például egy 3 szerveren replikált adategység esetén elég 2 csomóponton írni az adategységet. Így biztosítható, hogy a legutolsó írás művelet eredménye lesz látható a szerverek többségén. Az ún. *testületalapú protokollok* (quorum-based protocols) az írandó és az olvasandó adategységekre definiálnak küszöbértékeket, melyekkel képesek erős konzisztenciát nyújtani. Az adategységet *N* példányban tároljuk, a küszöbértékek:

- *R*: az olvasáshoz szükséges szerverek száma
- *W*: az íráshoz szükséges szerverek száma

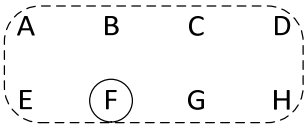
A küszöbértékekre a következő feltételeket definiáljuk [25] [26]:

1. $W > N / 2$
2. $W + R > N$

Az 1. feltétel miatt bármely két *írási testületnek* (write quorum) van közös szervere, ez lehetővé teszi az egymás utáni írások sorrendjének megfigyelését.
A 2. feltétel miatt az *olvasási* (read quorum) és az írási testületek mindig átlapolódnak, így erős konzisztencia biztosítható.
(Megjegyzés: az elosztott zárolásnál ismertetett „*k* az *n*-ből protokoll" is a fenti feltételeket kielégítő küszöbértékeket definiál a globális adategységek zárolására.)



10. ábra: Írási (szaggatott vonal) és olvasási testületek ($N = 8$, $R = 3$, $W = 6$)



11. ábra: Írási (szaggatott vonal) és olvasási testületek ($N = 8$, $R = 1$, $W = 8$)

A 10. és a 11. ábrán a testületi taglétszámok helyes megválasztására láthatók példák. A 11. ábra taglétszámválasztása „olvasáshoz egy, íráshoz mind" (Read One, Write All, ROWA) néven ismert.
(Megjegyzés: A séma hasonlít az elosztott zárolás *write locks all* sémájához, ami a *k* az *n*-ből protokoll speciális esete $k = n$-re)

Fokozatos konzisztencia

A 2. feltétel sérülése, azaz $W + R \leq N$ esetén előfordulhat, hogy az olvasási és az írási halmazok diszjunktak. Ezekben a rendszerekben – mivel erős konzisztencia nem garantálható – az olvasási küszöbérték (R) tipikusan 1 és a frissítések valamilyen *lusta* (lazy) algoritmus szerint terjednek a rendszerben. Az inkonszitenciaablak ebben az esetben az az idő, amíg a frissítés az adategység minden replikáját el nem éri. Amennyiben garantálható, hogy egy kliens egy *munkameneten* (session) belül mindig ugyanahhoz a szerverhez fordul, az MR és az RYW konszitenciamodellek feltételei viszonylag könnyen garantálhatók, de a *terheléselosztás* (load balancing) és a *hibatűrés* (fault tolerance) biztosítása nehezebbé válik [27].

A NoSQL rendszerek típusai

Az alábbiakban a NoSQL adatbázis-kezelők típusait ismertetjük, mely besorolás a NoSQL közösség által meghatározottnak felel meg – mindezt azért lényeges kiemelni, mert számos olyan rendszer létezik, mely nem sorolható be egyértelműen egyik csoportba sem [10] [35].

Kulcs-érték tárolók

A *kulcs-érték tárolók* (key-value stores) olyan egyszerű adatbázis-kezelők, melyek kulcsokat és a hozzájuk rendelt értékeket tárolják. Ennek megfelelően a kulcs-érték tárolók attribútum–attribútumérték párokat tartalmaznak. Az egyszerű adatmodell számos alkalmazási területen hasznos, azonban a lekérdezések korlátozottak; jellemzően csak kulcs szerint valósulhatnak meg. Bár az első kulcs-érték tárolót már a ’70-es években megalkották, ezek a rendszerek is csak a többi NoSQL rendszer megjelenésének idején indultak fejlődésnek, lévén a megváltozott igényeket már nem lehetett hatékonyan kulcs-érték tárolóként használt relációs adatbázis-kezelőkkel kielégíteni. A kulcs-érték tárolás ötletét az oszlopcsaládok és a gráfadatbázisok (ld. alább) is alkalmazzák.

Példák kulcs-érték tárolókra: Berkeley DB, Memcached, Project Voldemort, Redis, Riak.

Oszlopcsaládok

Az oszlopcsaládok tárgyalásánál fontos megkülönböztetnünk az oszlopalapú tárolást az oszlopcsaládoktól.

Oszlopalapú adattárolás relációs adatbázis-kezelőkben

A napjainkban elterjedt *relációs* adatbázis-kezelők többsége a rekordokat fizikailag sorokba szervezve tárolják (ld. 3. fejezet). A *soralapú* (row-based, row-oriented) tárolás előnye, hogy egy sor beszúrásához, módosításához vagy lekérdezéséhez jellemzően csak néhány blokkművelet szükséges. A módszer hátránya, hogy nehezen tömöríthető struktúrákat képez, valamint kevés attribútumot érintő lekérdezések esetén sok felesleges adatot is be kell olvasni a háttértárról.
A soralapú szervezéssel ellentétes elgondolás, az *oszlopalapú* (column-based, column-oriented) szervezés ötlete a ’80-as években merült fel. Az oszlopalapú tárolás a rekordok attribútumok szerinti csoportosítását jelenti, így az egyes attribútumok különböző értékei találhatók meg egy fizikai szervezési egységben. Ezzel a szervezéssel a kevés oszlopot és sok sort érintő elemzések hatékonyabban elvégezhetők, ezért előszeretettel alkalmazzák analitikus adatbázisokban, adattárházakban. Az oszlopalapú rendszerek magas szelektivitású (kevés sort érintő) lekérdezések esetén kerülnek hátrányba, mert az oszlopok végigolvasása sok felesleges blokkművelettel jár. Az oszlopalapú tárolók az egyes csoportokban azonos típusú adatokat tárolnak, ezért általában hatékonyan tömöríthetők.
Az alábbiakban a személy(ID, keresztnév, életkor, lakhely) sémára illeszkedő relációt szemléltetjük először sor-, majd oszlopalapú tárolással.

ID	keresztnév	életkor	lakhely
1	Klemens	42	Stuttgart
2	Rajesh	29	Delhi
3	Francesco	30	Rome
4	Colin	51	Dublin

Soralapú tárolás:

1. blokk	1	Klemens	42	Stuttgart
2. blokk	2	Rajesh	29	Delhi
3. blokk	3	Francesco	30	Rome
4. blokk	4	Colin	51	Dublin

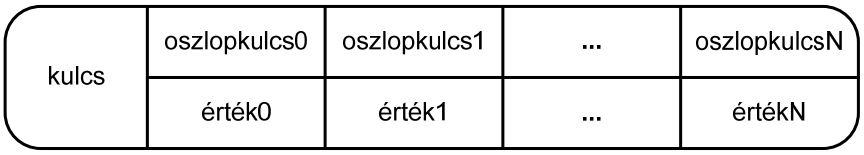
Oszlopalapú tárolás:

1. blokk	1	2	3	4
2. blokk	Klemens	Rajesh	Francesco	Colin
3. blokk	42	29	30	51
4. blokk	Stuttgart	Delhi	Rome	Dublin

Példák oszlopalapú relációs adatbázis-kezelőkre: Sybase IQ, Vertica.

A NoSQL oszlopcsaládok

Az elmúlt években megjelentek olyan NoSQL rendszerek, melyek az oszlopalapú alapelvet az adatmodellben alkalmazzák és kiegészítik más – jellemzően kulcs-érték – módszerekkel: ezek az *oszlopcsaládok* (column families). Az oszlopalapú adatbázisokhoz képest lényeges különbség, hogy míg azok relációk fizikai tárolását valósítják meg oszlopalapon, addig az oszlopcsaládok egy hierarchikus – nem relációs – adatmodell szerint, *logikailag* oszlopalapon szervezik az adatokat.
Az oszlopcsaládok sorai (rekordjai) kulcs-érték párokból állnak, melyek az oszlop (attribútum) nevét és értékét tartalmazzák. Számos implementáció tartalmaz az egyes kulcs-érték párokhoz tartozó időbélyegeket is (ld. MVCC). Egy oszlopcsalád egy kulcs-érték párból áll, ahol a kulcs egy tetszőleges elsődleges kulcs, az érték pedig az előbbiekben leírt oszlopok egy halmaza.



12. ábra: Oszlopcsalád általános felépítése

A 13. ábra a népszerű közösségi mikroblog szolgáltatás, a Twitter *bejegyzés*einek (postok) tárolásának egy lehetséges módját mutatja be oszlopcsaládokkal. Az oszlopcsaládok – rekordok – kulcsa a bejegyzés egyedi azonosítója, az egyes oszlopok pedig rendre a felhasználónevet (mely a Twitteren egyben egyedi azonosító is), magát a bejegyzést, illetve annak dátumát tartalmazzák.

12100	user_ID	text	datetime
	bmestudent	just decomposed a schema to 3NF #db #exam	2011-01-03 07:30:11

12187	user_ID	text	datetime
	bmestudent	just decomposed a schema to BCNF #db #exam	2011-01-03 07:41:36

13. ábra: Twitter bejegyzések tárolása oszlopcsaládd al

Az oszlopcsaládokat egy további kulcs-érték szinttel bővítve *szuperoszlopcsaládokat* (super column family) kapunk. Az így létrejövő szinteket *szuperoszlopoknak* (super column) nevezzük. A 14. ábra szemlélteti a szuperoszlopcsaládok általános felépítését.

kulcs	szuperoszlop kulcs0				...	szuperoszlop kulcsN			
	oszlop kulcs0	oszlop kulcs1	...	oszlop kulcsN		oszlop kulcs0	oszlop kulcs1	...	oszlop kulcsN
	érték0	érték1	...	értékN		érték0	érték1	...	értékN

14. ábra: Szuperoszlopcsalád általános felépítése

Az előbbi példához kapcsolódóan a 15. ábrán az egy felhasználó által leggyakrabban hivatkozott weboldalakat összegyűjtő szuperoszlopcsaládot láthatunk. Ebben az esetben egy szuperoszlopcsalád kulcsa a felhasználó neve, a szuperoszlopoké pedig a hivatkozott weboldalak URL-je. A szuperoszlopokat alkotó oszlopok kulcsai a szuperoszlopban található URL által kijelölt oldalnak azon aloldalai, melyekre a felhasználó leggyakrabban hivatkozott. Az oszlopok érték mezői az adott aloldalra való legutóbbi hivatkozás idejét mutatják.

bmestudent	tmit.bme.hu		en.wikipedia.org	
	/History	/InfoBScMedia	/wiki/ Relational_schema	/wiki/ Tuple_calculus
	2011-01-06 11:21:14	2011-01-06 11:25:01	2010-12-14 03:22:31	2010-12-14 04:00:56

15. ábra: Hivatkozott oldalak tárolása szuperoszlopcsaládd al

Példák (szuper)oszlopcsaládokra: Amazon SimpleDB, BigTable, HBase, Cassandra, HyperTable.

Dokumentumtárolók

Napjainkban a szemistrukturált adatok (ld. 1. függelék) egyre nagyobb jelentőséggel bírnak, így például tartalomkezelő, kereső-, ajánlórendszerekben. A szemistrukturált adatok különleges tulajdonságai tették szükségessé az ún. *dokumentumtároló* (document store) adatbázis-kezelők megalkotását. A dokumentumtárolókban

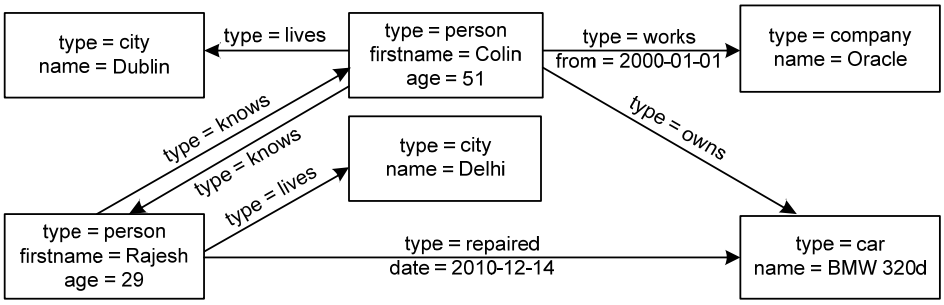
szemistrukturált adatok tárolhatók, melyeket jellemzően JSON vagy XML nyelveken írnak le [35] [36]. A dokumentumtárolók tehát nem szövegfájlokat kezelnek, hanem adatok olyan halmazát, melyek valamilyen laza módon strukturáltak.
Az alábbiakban egy JSON (JavaScript Object Notation) dokumentumra láthatunk példát, mely személyek jellemzőit és járművek egy listáját tartalmazza. Ha egy relációban szeretnénk tárolni a példa dokumentum tartalmát, igencsak bajban lennénk, ugyanis számos, ritkán használt attribútumot kéne felvennünk a relációs sémába. Az adathalmazt dokumentumként kezelve azonban nem okoz problémát, hogy Klemensről és Colinról más-más információkkal rendelkezünk (pl. age, company).
Vegyük észre, hogy a dokumentumok is kulcs-érték párokat tartalmaznak – jelen esetben egyes entitások attribútum típusait és attribútum értékeit (pl. city–Stuttgart).

```
{"document": [
  {
    "firstname": "Klemens",
    "city": "Stuttgart",
    "age": "42"
  },
  {
    "firstname": "Rajesh",
    "city": "Delhi",
    "age": "29"
  },
  {
    "firstname": "Colin",
    "company": "Oracle"
  },
  {
    "cars": ["BMW 320d", "Jaguar XF"]
  }
]}
```

Példák dokumentumtárolókra: CouchDB, MongoDB, Terrastore.

Gráfadatbázisok

Komplex, sok összefüggést tartalmazó adathalmazokat gyakran célszerű gráffal reprezentálni. Sajnos a gráfok relációs adatbázis-kezelőkben való tárolása esetén a gráfokon végzett műveletek rendkívül költségesek lehetnek: egy gráf bejárásához például több természetes illesztésre lehet szükség, ami köztudottan költséges művelet.
A *gráfadatbázisok* (graph database) gráfok hatékony tárolását és ezáltal gráfműveletek gyors végrehajtását teszik lehetővé [35].
A gráfadatbázisok jellemzően *tulajdonsággráfokat* (property graph) tárolnak, melyek csomópontjaihoz és éleihez tulajdonságok köthetők – rendszerint kulcs-érték párok formájában. Ezen tulajdonságok között jellemzően megtalálható az adott csomópontok és élek típusa is.
A 11. ábrán látható tulajdonsággráfról például a következő állítást olvashatjuk le: az 51 éves Colin Dublinban lakik.



16. ábra: Tulajdonsággráf

A gazdag adatmodell miatt a gráfadatbázisok jellemzően kevésbé skálázódnak a többi NoSQL rendszernél, a legtöbb gráfadatbázis csak replikációt támogat.

Példák gráfadatbázisokra: Neo4j, AllegroGraph, HypergraphDB, InfiniteGraph, FlockDB.

További NoSQL típusok

A részletesebben tárgyalt négy nagy csoport (core NoSQL rendszerek) mellett találhatók kisebb, kevésbé elterjedt típusok (soft NoSQL):

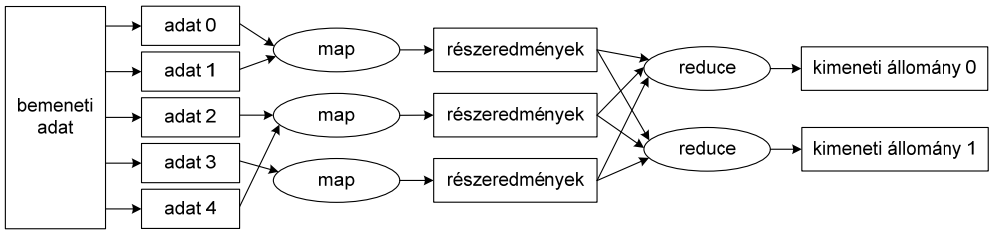
- objektum adatbázisok, pl. DB4o, Versant, ZODB
- XML adatbázisok (bár külön csoportként tartják számon, tulajdonképpen dokumentumtárolók), pl. EMC xDB, eXist.
- grid adatbázisok, pl. GigaSpaces, Hazelcast.

MapReduce

A MapReduce a NoSQL korszak jellegzetes, elosztott adatfeldolgozó algoritmusa, melyet a Google kifejezetten nagy adathalmazok párhuzamos feldolgozására fejlesztett ki [4], [34]. Valójában nem tartozik szorosan az adatbáziskezeléshez, azonban sikeressége/hatékonysága miatt több NoSQL rendszerben, ill. hozzájuk kapcsolódva is megtalálható.
A MapReduce paradigma a funkcionális nyelvek (pl. Erlang, Haskell, LISP) `map()` és `reduce()` (egyes nyelvekben `fold()`) eljárásaiban gyökerezik. A map a bemeneti lista minden elemére végrehajt egy meghatározott műveletsort, és visszatér a módosított listával, amit a reduce eljárás aggregál egy végeredménnyé.
A MapReduce keretrendszerek a map és a reduce eljárásokat párhuzamosan hajtják végre. Az eljárásokat a fejlesztőnek kell implementálnia annak megfelelően, hogy milyen feladatokat kíván egyidejűleg végrehajtani. A MapReduce keretrendszerekben az adatok feldolgozása a következő fázisokra tagolódik (ld. 17. ábra):

1. Bemeneti adatok felosztása az egyes map processzek részére.
2. Párhuzamos adatfeldolgozás a map processzekben.
3. Részeredmények tárolása és kulcs szerinti rendezése – az azonos köztes kulcscsal rendelkező részeredmények kerülnek ugyanahhoz a reduce processzhez.
4. Részeredmények párhuzamos feldolgozása a reduce processzekben.

5. Kimeneti adat(ok) írása reduce folyamatonként.



17. ábra: A MapReduce fázisai

A map metódus implementációja kötelező, míg a reduce-é jellemzően opcionális. Lehetséges továbbá a bemeneti adatok felosztásának, a köztes eredmények csoportosításának, illetve a végeredmények összesítésének egyedi implementációja is, azonban alapesetben ezeket a feladatokat a keretrendszerek végzik. Lényeges, hogy a bemeneti, a köztes és a kimeneti adatoknak kulcs-érték formátumban kell lenniük. Megjegyzendő továbbá, hogy egy bemeneti kulcs-érték párból nem feltétlenül egy köztes értékpár lesz a map és/vagy a reduce folyamatok során (pl. egy szöveges értéket karakterenként is feldolgozhatunk).

A reduce függvény implementálásakor eleget kell tennünk az alábbi szempontoknak:

- A függvény bemeneti típusa egyezzen meg a map függvény kimeneti típusával. Így ha a map függvény kimenetén csak egyetlen kulcs-érték pár jelenik meg, a reduce függvényt nem kell futtatni.
- A művelet legyen idempotens, azaz ha függvény kimenetét egy (egyetlen elemet tartalmazó) tömbbe helyezzük és újra meghívjuk rá a reduce függvényt, az eredmény maradjon változatlan. A reduce függvényt ugyanis a feldolgozás során többször is lefuthat, például a feldolgozás végén a master szerver egy reduce művelettel összesíti a különböző szerverektől kapott adatokat [34].

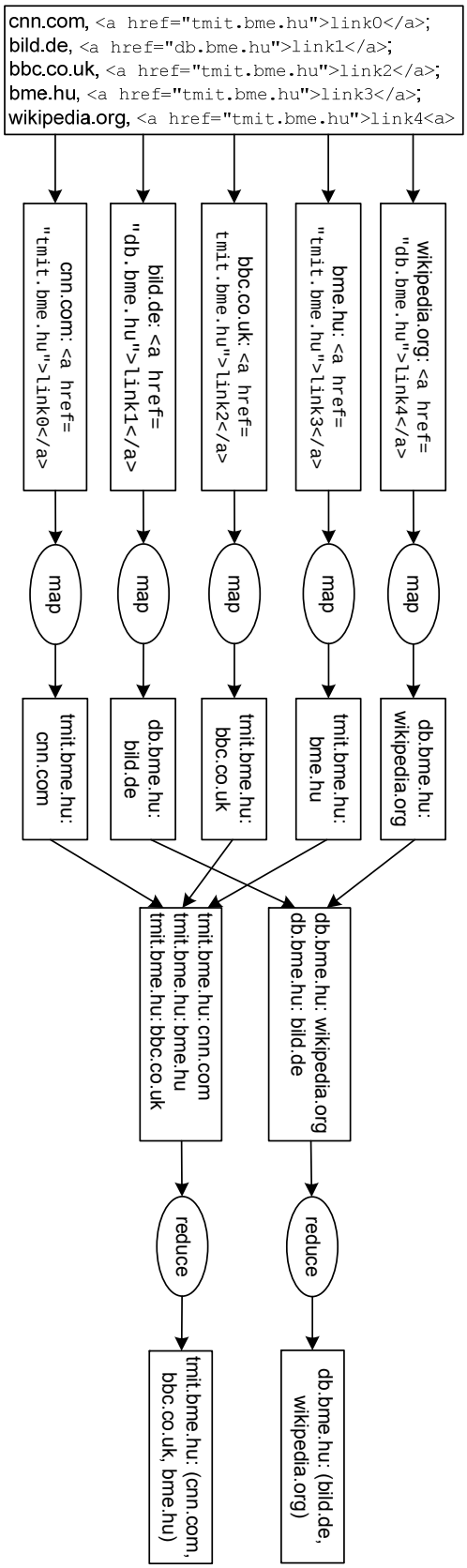
Az alábbiakban a MapReduce alkalmazására láthatunk egy példát, mely az egyes URL-ekre hivatkozó külső URL-eket gyűjti össze.
A map függvény URL-eket és az alattuk található weboldalak forráskódját kapja bemenetként, eredményül pedig <cél, forrás> párokat ad, ahol cél egy olyan weboldal URL-je, amelyre a feldolgozott – forrás – oldal hivatkozik.
A reduce függvény az egyes cél URL-ekhez tartozó forrás URL-eket fűzi össze egy-egy listába.
Az ismertetett példa map és reduce függvényeinek pszeudokódja a következő:

```
map(key, value) {
    // key = URL
    // value = page content
    For each url, linking to target
        Generate(output target, source);
}
reduce(key, values) {
    // key = target URL
    // values = all URLs that point to the target URL
    For each values
        Generate(key, values[i]);
}
```

A függvények működése a 18. ábrán látható.

Összegzés

Láthattuk, hogy a NoSQL adatbázis-kezelők a korábbiakban megismert rendszerekhez képest újszerű igényeket elégítenek ki évtizedes és friss ötletek kombinálásával. Az ilyen rendszerek száma az utóbbi években folyamatosan növekedett – ez a tendencia a jövőben is igaznak bizonyulhat.
A NoSQL rendszerek térnyerése mellett a relációs adatbázis-kezelők szerepe továbbra is jelentős marad, lévén a klasszikus – és tömeges – igényekre még mindig ezek tekinthetők a leggazdaságosabb megoldásnak.



18. ábra: A MapReduce működése

Irodalomjegyzék a 2. függelékhez

[1] IBM, *What is big data? – Bringing big data to the enterprise,* http://www-01.ibm.com/software/data/bigdata/

[2] Louis Columbus, *Roundup of Big Data Forecasts and Market Estimates,* Forbes, 2012, http://www.forbes.com/sites/louiscolumbus/2012/08/16/roundup-of-big-data-forecasts-and-market-estimates-2012/

[3] Edd Dumbill, *Volume, Velocity, Variety: What You Need to Know About Big Data,* Forbes, 2012, http://www.forbes.com/sites/oreillymedia/2012/01/19/volume-velocity-variety-what-you-need-to-know-about-big-data/

[4] Jeffrey Dean, Sanjay Ghemawat, *MapReduce: Simplied Data Processing on Large Clusters,* Google Inc., OSDI, 2004, http://research.google.com/archive/mapreduce.html

[5] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, Robert E. Gruber, *Bigtable: A Distributed Storage System for Structured Data,* Google Inc., OSDI, 2006, http://research.google.com/archive/bigtable.html

[6] Mike Burrows, *The Chubby lock service for loosely-coupled distributed systems,* Google Inc., OSDI, 2006, http://research.google.com/archive/chubby.html

[7] *no:sql(east),* https://nosqleast.com/2009/

[8] Couchbase, *NoSQL Database Technology – Post-relational data management for interactive software systems,* 2011, http://www.couchbase.com/sites/default/files/uploads/all/whitepapers/NoSQL-Whitepaper.pdf

[9] Pramod J. Sadalage, Martin Fowler, *NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence,* Addison-Wesley Professional, 2012.

[10] *NoSQL Databases,* http://nosql-database.org/

[11] Mark D. Hill, *What is scalability?,* http://dl.acm.org/citation.cfm?id=121975

[12] Werner Vogels, *Availability & Consistency or how the CAP Theorem ruins it all,* QCon, 2007, http://www.infoq.com/presentations/availability-consistency

[13] Michael Stonebraker, *The Case for Shared Nothing,* HPTS, 1985, http://db.cs.berkeley.edu/papers/hpts85-nothing.pdf

[14] Armando Fox, Eric Brewer, *Harvest, Yield, and Scalable Tolerant Systems,* Proceedings of the The Seventh Workshop on Hot Topics in Operating Systems, 1999, http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=798396&tag=1

[15] Eric Brewer, *Towards Robust Distributed Systems,* PODC Keynote, 2000. július 19, http://www.eecs.berkeley.edu/~brewer/cs262b-2004/PODC-keynote.pdf

[16] Seth Gilbert, Nancy Lynch, *Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services,* http://portal.acm.org/citation.cfm?id=564601

[17] Julian Browne, *Brewer's CAP Theorem,* 2009, http://www.julianbrowne.com/article/viewer/brewers-cap-theorem

[18] Nancy Ann Lynch, *Osztott Algoritmusok,* Kiskapu, 2002.

[19] Ron Kohavi, Roger Longbotham, *Online Experiments: Lessons Learned,* IEEE Computer, 2007. szeptember, http://ai.stanford.edu/users/ronnyk/IEEEComputer2007OnlineExperiments.pdf

[20] Daniel Abadi, *Problems with CAP, and Yahoo's little known NoSQL system,* 2010. április 23, http://dbmsmusings.blogspot.com/2010/04/problems-with-cap-and-yahoos-little.html

[21] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, Werner Vogels, *Dynamo: Amazon's Highly Available Key-value Store,* SIGOPS Oper. Syst. Rev., 2007. december, http://dl.acm.org/citation.cfm?id=1294281

[22] Armando Fox, Steven D. Gribble, Yatin Chawathe, Eric A. Brewer, Paul Gauthier, *Cluster-Based Scalable Network Services,* SOSP, 1997, http://dl.acm.org/citation.cfm?id=266662

[23] Dan Pritchett, *BASE: An Acid Alternative,* ACM Queue, http://queue.acm.org/detail.cfm?id=1394128

[24] GenieDB, *Beating the CAP Theorem,* http://www.geniedb.com/wp-content/uploads/2011/04/beating-the-cap-theorem-revised.pdf

[25] Brian F. Cooper, Raghu Ramakrishnan, Utkarsh Srivastava, Adam Silberstein, Philip Bohannon, Hans-Arno Jacobsen, Nick Puz, Daniel Weaver, Ramana Yerneni, *PNUTS: Yahoo!'s Hosted Data Serving Platform,* 2008, http://research.yahoo.com/pub/2304

[26] Andrew S. Tanenbaum, Maarten Van Steen, *Elosztott rendszerek,* Panem, 2004.

[27] Werner Vogels, *Eventually Consistent,* ACM Queue, 2008. október, http://dl.acm.org/citation.cfm?id=1466448&bnc=1
[28] Douglas B. Terry, Alan J. Demers, Karin Petersen, Mike J. Spreitzer, Marvin M. Theimer, Brent B. Welch, *Session Guarantees for Weakly Consistent Replicated Data,* 1994, http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.71.2269
[29] Jerzy Brzezinski, Cezary Sobaniec, Dariusz Wawrzyniak, *From Session Causality to Causal Consistency,* 2004, http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.106.3608&rep=rep1&type=pdf
[30] Roger Wattenhofer, *Distributed Systems course,* http://www.disco.ethz.ch/lectures/hs12/distsys/
[31] Todd Lipcon, *Design Patterns for Distributed Non-Relational Databases,* cloudera, 2009, http://cloudera-todd.s3.amazonaws.com/nosql.pdf
[32] M. Raynal, G. Thia-kime, M. Ahamad, *From Serializable to Causal Transactions for Collaborative Applications,* 1996, http://reference.kfupm.edu.sa/content/f/r/from_serializable_to_causal_transactions_126622.pdf
[33] Tom White, *Hadoop: The Definitive Guide, Second Edition,* O'Reilly Media, 2010.
[34] MongoDB, MapReduce, http://www.mongodb.org/display/DOCS/MapReduce
[35] Edlich, Friedland, Hampe, Brauer, *NoSQL,* Hanser, Berlin, 2010.
[36] J. Chris Anderson, Jan Lehnardt, Noah Slater, *x* O'Reilly Media, 2010.

A weblapok elérési ideje: 2012. augusztus