1. Bevezetés

A nagy adat (big data) napjainkban az egyik vezető cím az informatikai terminológiák körében, különböző címekkel ellátva, mint: *Sikerhez és boldogsághoz vezet a big data* [**ori**] vagy *Új korszak kezdődött a tudományban és* [**oria**]. Olyannyira megnőtt a kereslet a nagy adattal foglalkozó szakemberek iránt, hogy már az informatikán kívül is megjelent, Hal Varian, a híres közgazdász –aki egyben Google vezető közgazdásza– szerint a *következő 10 éven belül az egyik legvonzóbb szakma lesz az adatokkal foglalkozó állás.* [**varian**]. De mit is jelent pontosan? Nincs explicit meghatározás a fogalomra, de Doug Laney 2001-es definíciója egy jó kindulópontnak tekinthető: az adatok nagy mennyiségben (volume), gyorsan (velocity) és különböző formátumban (variety) jelennek meg (3V's) [**3v**]. Azonban, ma már kiegészíthetjük ezt a fogalmat még 4V-vel: bizonyosság (veracity) és érték (value), adategyezés (variability) és megjelenítés (visualization). [**7v**] Az adatmennyiség amit előállítunk exponenciálisan növekszik olyan szintre, aminek tárolását, menedzselését és elemzését már nem tudjuk megoldani a saját, lokális erőforrásainkon belül az eddig megszokott adatelemzési eszközökkel, mint például Microsoft Excel, vagy különböző relációs adatbázis technológiák által. Becslések [**2020**] szerint az adatok mennyisége kétévente duplázódik, így 2020-ra az összekézben forgó adatmennyiség elérheti a 44 zetabájtnyi (vagy 44 trillió gigabájtnyi) mennyiséget.

A „big data" lehetőséget biztosít arra, hogy ezeket az adatokat ne csak tároljuk, hanem új módokon tanuljunk belőle, értéket állítsunk elő, többet megtudjunk ügyfeleinkről, a saját üzleti folyamatainkról, ami versenyelőnyhöz vezethet. E mellett az áttörő kutatások számát is megnövelheti azáltal, hogy rejtett összefüggéseket mutat meg. [**brk**]

A cloud computing, és új technológiák megszületése és az, hogy a fizikai világ egyre jobban átterelődik az online térbe, új nehézségeket állít elő mind az adatokat kiszolgáló, mind az adatokat elemző infrastruktúrák számára. Ezek a problémák komoly gondot jelentek az informatikai iparnak, mivel érintik az fizikai manifesztációt (hardver), mind az ezt vezérlő és feldolgozó réteget (szoftver és algoritmus). Ezek a problémák, [**dst**] –amelyek a tradícionális adattárház technológiákra jellemzőek– többek között származhatnak a hiba-tolerancia hiányából, a sokféle adatfajtából, a párhuzamosság hiányából, mely azt eredményezi, hogy a mai technológia fejlettség (és a központi számítási egységek fizikailag limitáltsága miatt) nem lesz megfelelő számítási teljesítmény a megnövekedett adatmennyiség menedzselésére.

2. A dolgozat célja

A technológia fejlődése és a számítási teljesítmény megnövekedése hozta létre azt az üzleti igényt [**rta**], hogy egyre gyorsabban, egyre nagyobb adatmennyiség feldolgozása történjen meg. Ilyen igény például:csalás felderítés [**fraud**], "dolgok" internete (IoT) [**iot**] vagy alkalmazás monitoring [**ganalitycs**]. Ez az adatfeldolgozási sebesség olyan szintre eljutott, hogy közel valós időben, az adat keletkezése után megtörténhet ennek feldolgozása. Ilyen gyorsaságú adatfeldolgozásra csak elosztott rendszerek segítségével vagyunk képesek, [**ucl**] amelyek felépítésükből fakadóan sok lehetőség és költség jellemez, amelyeket a későbbiekben fogok kifejteni. A dolgozatomban használt Apache Flink (mely az Apache Foundation egyik legújabb és legmodernebb terméke) platform közel 40 millió elem feldolgozására képes egy 40 magos architektúrán másodpercenként. [**flink**].

Ahhoz, hogy ezt az adatmennyiséget ki tudjuk elemezni és ajánlásokat tudjunk adni, gépi tanulásra van szükségünk. A gépi tanulás az informatikának és a matematikának egy olyan ága, amely az adatok folyamatos betáplálása során új ismereteket szolgáltat, megpróbál előrejelzéseket adni anélkül, hogy explicit módon be lenne erre programozva. [**ml**]. A gépi tanulás egy olyan fajtáját fogom alkalmazni, ahol a termékek és felhasználóktól szerzett adatokból a lehető legpontosabban próbáljuk megjósolni, hogy milyen termék szerepelhet a felhasználó preferencialistája elején. Célom, hogy elosztott módon felépítsem az algoritmust, megnézzem, hogy milyen esetekben érdemes a párhuzamosítás, hol van az a határ, ahol a felépítésből fakadó többletköltség és komplexitás ellenére is megéri párhuzamos architektúrát alkalmazni.

A választott metódus a sztochasztikus gradiens leszállás (SGD, stochastic gradient descent) [**sgd**], amely egy olyan egyszerűsítési illetve optimalizációs eljárás, ahol adott célfüggvény gradiensét folyamatosan, iteratív módon számoljuk ki. Dolgozatomban megtervezem Apache Flinkben az SGD algoritmust, összehasonlítom a teljesítményét a már implementált algoritmusokkal és megkezdem a szükséges módosítások implementálást.

3. Elosztottság

A megnövekedett fizikai modellezési és katonai problémák miatt már 1950-es évektől kezdve felmerült az a kérdés, hogy hogyan lehetne összetett számítási feladatokat minél gyorsabban, egyszerűbben elvégezni. [**cocke**] Rájöttek, hogy a problémákat olyan részproblémákra kell bontani, melyek meg-

oldása egymástól független, így párhuzamosan is megoldhatóak, így jött létre a párhuzamos programozás.
Párhuzamos programozás következő lépéseként jött létre az elosztott rendszerekre való igény, mely esetében a számolások már nem csak az adott számítási egységben különülnek el, hanem jóval több, akár különböző fizikai lokáción levő számítógép is ugyanannak a problémának egy számítását végzi. Azonban az ily módon elért számítási teljesítmény növekedés a komplexitás megnövekedésével is jár. Ezt a komplexitást különböző keretrendszerek elfedik, így biztosítva a lehetőséget elosztott alapon működő alkalmazások fejlesztésére. A komplexitás növekedése mellett a lehetőségeink is korlátozottak egy elosztott rendszer esetén.

Egy elosztott fájlrendszer esetén a fájljaink blokkokra vannak osztva a klaszter különböző csomópontjai (node) között.Ezek a különböző fájlblokkok másolatai elérhetőek különböző (logikai és/vagy fizika) lokációkon, mellyel biztosítjuk, hogy hardver meghibásodás esetén is elérhető legyen az adat. E mellett a rendszerben lévő névcsomópont (namenode) tárolja a fájlok és a hozzátartozó blokkok helyét. [**hdfs**] –TODO bővíteni

3.1. Hibatűrés

Az informatikai rendszerek összetettségének növekedésével megnő annak a valószínűsége, hogy a rendszerünk (vagy az alkalmazásunk) nem fogja a tőle elvárt, korrekt és pontos kimenetet biztosítani. Ezek a hibafaktorok lehetnek hardver (tároló lemez kiesés) vagy szoftver (rossz adat partícionálás) esetleg a köztes szoftverből (middleware) származóak. Ahhoz, hogy megbízható rendszert építsünk, lényeges, hogy a tervezett rendszer a különböző komponensek hibáját (közel) maximálisan tudja kijavítani. Az elosztott rendszerek kliens-szerver architektúra szerinte vannak felépítve, ahol távoli eljáráshívásoknál (RPC) kommunikálunk. Különböző mechanizmusokkal kell garantálnunk a hívás megérkezését, a hibák kijavítását és korrigálását. Ezek a különböző mechanizmusok lehetnek [**szemantika**]:

- Kliens nem találja a szervert, ezért a szervernek tájékoztatnia kell a klienst a kimaradásról
- Kliens és szerver is megfelelően működik, viszont hálózati hiba miatt a hívás nem jut el a szerverig, esetleg a szerver válasza nem jut el a klienshez. Ilyen esetekben időkorlát (timeout) bevezetése egy jó megoldás, ami során újra küldjük a hívást

- Hívásoknak idempotensnek kell lennie, tehát a többszöri azonos hívás futtatás sem fog hibát okozni a rendszerben

Amikor a szerver összeomlik olyanfajta szemantikákat kell alkalmaznunk, amely biztosítja a klienst a szerver állapotáról, illetve a szerver felépülés esetén tájékoztatja a klienst a jelenlegi állapotáról. Különböző hibatűrési szinteket vállalhatunk a hibatűrö rendszerünkkel: [**akka**]

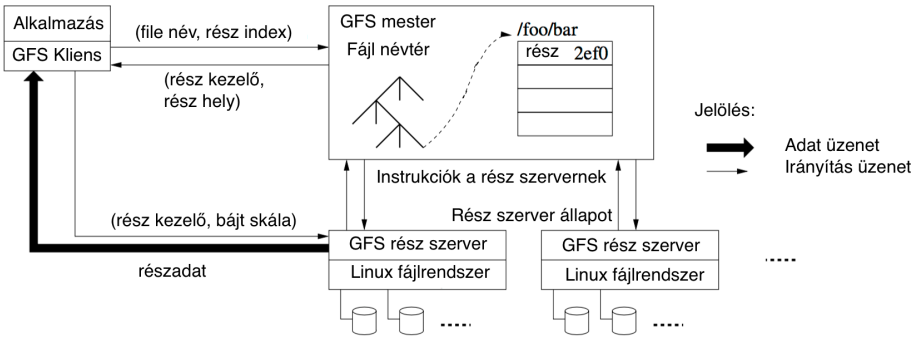
- Nincs semmilyen garancia az üzenet megérkezésére
- At-most once (legfeljebb egyszer): Minden üzenet legfeljebb egyszer kézbesítésre kerül, de ezeknek az üzeneteknek egy rész nem jut el a címzetthez (üzenetek elveszhetnek).
- At-least once (legalább egyszer): Minden üzenet legalább egyszer kézbesítésre kerül, de az üzenetek többször is eljuthatnak egy címzetthez, ami duplikált rekordokhoz vezethez (de semmiképpen nem veszik el)
- Exactly once (Pontosan egyszer): Minden üzenet pontosan egyszer kerül kézbesítésre és fogadásra, nincs elveszett és duplikált üzenet sem.
- Garantáljuk az üzenet elküldését pontosan egyszer, és ezek az üzenetek helyes sorrendben érkeznek meg.

Nem minden alkalmazás/szolgáltatás szegmensnek van szüksége arra, hogy mindig maximális pontosságú kimenetet adjon, ez az ún. részleges hiba tolerancia. –TODO bővíteni -exactly once, at most once, etc

3.2. Google fájl rendszer

Google 2003-ban megalkotta a google fájl rendszert (GFS, Google File System) [**gfs**], ami napjaink elosztott, hibatűrö rendszereinek az alapja. Tervezésénél fontos szempont volt, hogy skálázható, megbízható, elérhető és magas teljesítménnyel rendelkező legyen mindezt úgy, hogy adat-intenzív applikációk alapját fogja szolgálni. Figyelembe vették, hogy a komponens (mind adat, mind hardver oldalon) hibák inkább általánosak, mint kivételek, így az architektúra tervezésénél ez különös figyelmet kapott. E mellett relatív nagy fájlokra szabták (64 megabájt) és fájlok felülírása helyett (overwrite) hozzáfűzték (append) az adatokat a létező fájlokhoz. Az elrendezés könyvtár alapú, a fájlokat névtér és fájl név alapján lehet azonosítani. Támogatja a megszokott létrehozás, törlés, megnyitás, olvasás és írás operátorokat, de bevezet kettő újat is a legoptimálisabb működés miatt: pillanatkép (snapshot) aminek segítségével a fájlrendszer pillanatnyi állapotról lehet mentést

készíteni és párhuzamos hozzáfűzés (record append) ami lehetővé teszi, hogy egyszerre több kliens is tudjon egy fájlt írni.



1. ábra. Google fájl rendszer felépítése http://static.googleusercontent.com/media/research.google.com/en//archive/gfs-sosp2003.pdf

A fájlokat 64 megabájtos részekre (chunk) osztják, amit egy 64 kilobájtos fájl (chunk handle) kezel. Különböző szerverek (chunk server) tárolják a részeket, és minden fájlból három másolat készül a nagyobb konzisztencia érdekében. Az állapotkezelést és utasításokat mester-szolga (master-slave) módon építették fel, mester tárolja memóriában a metaadatokat (névtér, fájl helyek és fájl - > chunk leképzést). Ezen kívül biztosítva van az automatikusan, pár másodperc alatt feléledő a rendszer, kritikus, infrastruktúra hiba esetén.

3.3. CAP tétel

Az elosztott rendszerek tervezésekor és használatakor három képességet várunk el elsősorban: legyen konzisztens (consistency), elérhető (availability) és partícionálás-tűrő (partition tolerance).

- Konzisztencia: Bármelyik csomópontból kérdezem le az adatokat, mindig helyes választ kapok. Elosztott rendszereknél a konzisztencia megtartásához elengedhetetlen, hogy a csomópontok kommunikáljanak egymással.
- Elérhető: Bármelyik időpillanatban kapok választ egy kérés küldése után
- Partícionálás-tűrő: Ha –a teljes hálózati kieséstől eltekintve– egy csomópont eltűnik a hálózatról, akkor is kapok választ a rendszertől.

A CAP tétel kimondja [**cap**], hogy ebből a háromból legfeljebb kettő lehet igaz egy adott időpillanatban. Azonban a valóság nem ennyire letisztult, különböző rendszerek különböző prioritással veszik figyelembe a három alapképességet, így korlátozottan megtalálhatjuk a három alapképességet egy modern elosztott rendszerben. [**ecap**]

3.4. Párhuzamos kötegelt adatfeldolgozás

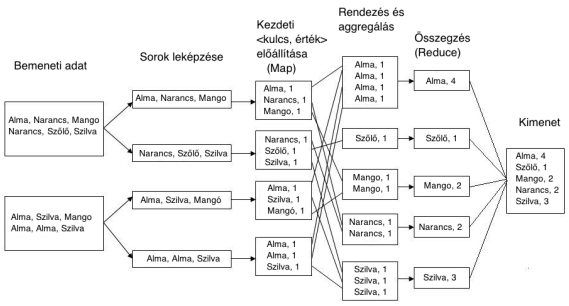
A kétezres évek elején és közepén, a megfelelő számítási kapacitás hiányában a párhuzamos adatfeldolgozás általában *kötegelt* (batch) módon történt. Felhasználói interakció nélkül, megadott időközönként történik a nagy mennyiségű adat feldolgozása. Előnye, hogy akkor futhat a program, amikor a rendszer leterheltsége alacsony, ezzel biztosítva az egyenletes kihasználtságot. Mivel a parancsok automatikusan futnak le, ezért kisebb az üres, nem számítással töltött idő. [**batch**].

3.5. MapReduce

A Google által fejlesztett MapReduce volt az első olyan szélesebb körben is elismert programozási modell, ami lehetővé tette, hogy nagymennyiségű adatot is lehessen feldolgozni párhuzamosan, elosztott módon. [**mapreduce**] Az adatokat <kulcs, érték> párokra bontjuk, ahol a kulcs egy referencia, ami hivatkozik az adatra, míg az érték maga az adat. A bemenet az adat, míg a kimenet a köztes <kulcs, érték> pár. Ezt az ún. Map függvény valósítja meg, amit a felhasználónak kell biztosítania. A köztes <kulcs, érték> párt a felhasználó által írt Reduce függvény dolgozza fel. A Reduce függvényben a köztes <kulcs, érték> párok összegzése zajlik, tehát megkeressük a vizsgált kulcshoz az összes értéket.
Az egyik klasszikus példa a szószámolás, ahol meg kell számolnunk azt, hogy a bemeneti adatunk szavai milyen mennyiséggel fordulnak elő.

Elsőként a bemeneti adatunkat (szövegfájl), sorokra bontjuk, majd a sorokat szavakra és a hozzájuk tartozó előfordulási értékekre (Map). Az így előállt <kulcs, érték> párokat aggregálom, majd összefésülöm (Reduce). A végső kimenet pedig ezeknek a összesített pároknak az összessége.

A ma használt keretrendszerek, az egyszerű ötletnek és magas absztrakciós szintnek köszönhetően lehetővé teszik, hogy a felhasználónak csak a konkrét adatfeldolgozó kóddal kelljen dolgoznia, mivel a keretrendszer elfedi az elosztott számítási komplexitást. [**hadoop**]



2. ábra. MapReduce alkalmazása szószámoló példán keresztül. (forrás:http://kickstarthadoop.blogspot.hu/2011/04/word-count-hadoop-map-reduce-example.html)

3.6. Adatfolyam alapú feldolgozás

Mit is tekinthetünk adatfolyamnak? Olyan adatfeldolgozó motort, mely arra van tervezve, hogy végtelen és rendezetlen adatsorokat dolgozzon fel. Eddig az ilyen rendszereket alapvetően alacsony pontossággal és/vagy megbízhatatlansággal vádolták, mely csak spekulálni tud az egyes adatok valódi értékéről. Elmondhatjuk, hogy jobb megértést tud biztosítani a problémán az, hogy ha a végtelen adatunkat valós időben dolgozzuk fel, mivel az esetek többségében az adatunk elavul, csak egy limitált időkorláton belül tudjuk értelmezni, illetve hasznos információkat kinyerni. Lambda architektúra a maga idejében egy rendkívül jó megoldás volt, biztosítva az alacsony válaszidő és a pontosság egyvelegét. Ahogy azonban feljődtek a technológiai megoldások, egyre jobban kiütköztek a hátrányok is. A többszörös adatfeldolgozás miatt egyszerre két infrastruktúrát kell fent tartani, ami növeli a komplexitást, hibalehetőséget, és a befektetett időt, mivel minden kódmódosítást két helyen kell egyszerre elvégezni. Léteznek félmegoldások a problémára, mint a Twitter által fejlesztett Summingbird [**summingbird**], ami egy magas szintű függvénykönyvtár, mely fordítás után optimalizál a kötegelési és a sebesség rétegre. Viszont ebben az esetben is megmarad az operatív teher, amit 2 különböző infrastrutúra fenttartása okoz. Másrészt pedig csak olyan technikai megoldásokat használhatunk, amely a két engine metszéspontjában szerepel. Ennek kiváltásra születtek meg a ma ismert modern adatfolyam alapú feldolgozó egységek (streaming). A rendezetlen (unorder), végtelen (unordered) és teljes globális skálázású rendszerek kiszolgálását csak úgy lehet megoldani, ha olyan rendszert fejlesztünk, ami ténylegesen ezekre a problémákra ad

megoldást. [**tyler**]. E mellett egyéb előnyei is vannak az adatfolyam alapú rendszereknek:

- Ki tud elégíteni olyan valós idejű üzleti igényeket, mint pl.:anomália keresés, csalásfelderítés, hírdetéselhelyezés
- Hosszú távon az erőforrás eloszlás kiegyenlítettebb lesz, mivel mindig (majdnem) akkor kerül feldolgozásra, amikor létrejön

Ha két dolgot szem előtt tartunk a rendszer tervezésnél, akkor túl tudunk lépni a Lambda architektúra (és a fejlettebb micro-batch rendszerek [**microbatch**]) lehetőségein. Az első a konzisztens tárolás, ami azt jelenti, hogy hosszú idő után is, esetleges gépi hiba esetén is megmaradjanak az adatok helyes formájukban, pontosan egyszer (at-most-once). A másik, hogy biztosítsuk, hogy a különböző időben érkező, de összetartozó, rendezetlen adatok helyes feldolgozása is megtörténhessen. Ahhoz, hogy ezt megértsük, két fogalmat kell definiálnunk:

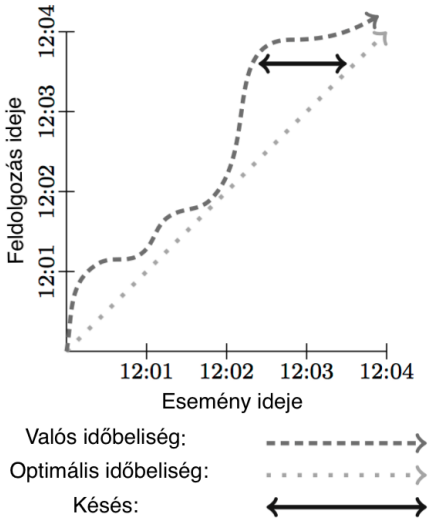
- Esemény ideje (event-time), amikor megszületett az adat
- Feldolgozás ideje (processing-time), amikor feldolgozásra került az adat

Ideális esetben ez a két időhorizont megegyezik, tehát közvetlenül akkor dolgozzuk fel az adatot, amikor az létrejött. Sajnos, azonban a bemeneti forrás késése, a feldolgozó motor hibája vagy hardver üzemszünet miatt nem lehetséges.

3.7. Lambda architektúra

A big data megjelenésével, megjelent az igény, hogy ezeket az adatokat közel valós időben tudjuk feldolgozni. Egy fejlettebb megoldásnak tekinthetjük a Lambda architektúrát, ami a kötegelt feldolgozást (batch processing), és a adatfolyam feldolgozást (stream processing) egyesíti. Megalkotása során törekedtek arra, hogy

- Robosztus, hiba toleráns legyen, mind hardver, mind szoftver oldalon
- Széles körű felhasználhatóság biztosítson, alacsony válaszidővel
- Horizontálisan skálázható legyen (több "általános célű" gép használatával lehessen növelni a teljesítményt)
- Bővíthető legyen

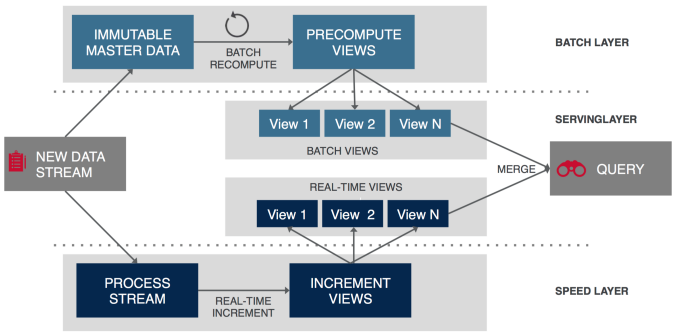


3. ábra. Adatfolyam feldolgozás a valóságban.3

Ezen feltételek mellett egy három rétegű architektúrát alkotott meg Nathan Marz [**lambda**]. A kötegelési réteg (batch layer), a sebesség réteg (speed layer) és a kiszolgáló réteg (serving layer) biztosítja az adatfeldolgozást. Az adat változatlan formában eljut mind a kötegelt, mind a sebesség rétegbe. A kötegelt réteg tartalmazza a mester adathalmazt, ami nem módosítható, csak egyszer írható formában tárolja az adatokat. Ez a réteg meghatározott időszakonként, ciklikusan lefutattja a számításait, amivel létrehozza az ún. kötegelt nézetet (batch view). Ez a réteg felel azért, hogy az adat pontosan (lehetséges újraszámítás, ahogy érkeznek az újabb adatok) és teljesen jelenjen meg a felhasználó előtt (magas válaszidővel). A kiszolgáló réteg indexeli ezeket a nézeteket, ezzel biztosítva az ad-hoc, gyors lekérdezéshetőségüket. A sebesség réteg csak friss adatokkal dolgozik, így csökken a pontosság és a teljesség, viszont gyors, inkrementális algoritmusok segítségével, alacsony válaszidővel tudja az adatokat a kimenetre küldeni. A kiszolgáló réteg a kötegelt és a sebesség nézetek összefűzéséből állítja elő az elvárt kimenetet.

3.8. Adatfeldolgozó mintázatok

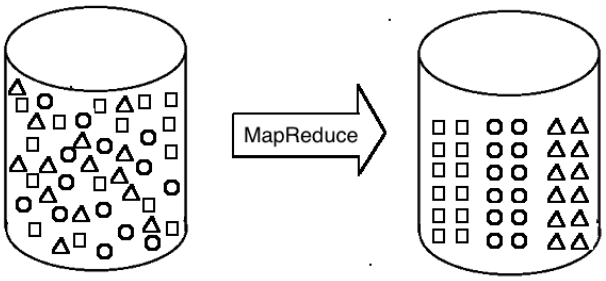
Ahhoz, hogy tudjuk mivel dolgozik egy modern, adatfolyam alapú feldolgozó egység, részleteznünk kell az eddig használtakat.



4. ábra. Lambda architektúra felépítése

3.8.1. Kötegelt feldolgozás véges adaton

Az adatfeldolgozási folyamat nagyon egyszerű, vesszük a különböző típusú adatokat meghatározott időközönként és egy adatfeldolgozó architektúra (pl.: MapReduce [**mapreduce**]) segítségével átalakítjuk struktúrált adatokká.



5. ábra. Bal oldalon található entrópikus adatokból MapReduce segítségével struktúrált, információval rendelkező adatokat generálunk. http://radar.oreilly.com/2015/08/the-world-beyond-batch-streaming-101.html

3.8.2. Kötegelt feldolgozás végtelen adaton

Végtelen adatnál még mindig használhatunk kötegelt feldolgozó rendszereket, azzal a kiegészítéssel, hogy az adatok felbontjuk véges részekre, így

azok feldolgozhatóvá válnak. Két fő módszertan van elterjed, a rögzített windowing és a session.

Rögzített windowing arra vonatkozik amely során fix méretű átmenetei időblokkokat vezetünk be, és ezeken az ablakokon futtatjuk le feldolgozó algoritmusunkat. A problémák ennél a megoldásnál nyilvánvalóak. Többidejűséggel (több sorozatban érkezik meg az adat) és többhelyűséggel (több földrajzi elhelyezkedésről érkezik meg az adat) nem tud foglalkozni, így nem tudja biztosítani az adat teljességét és pontosságát.

Sessionnél valamilyen felhasználói aktivitás alapján meghatározzuk, hogy valószínűleg mennyi ideig fog tartani az adatfolyam. Ez alapján hozok létre ablakokat, melyeken végrehajtom a műveleteimet. Főbb probléma, ha átcsúszik az adat a következő session-be, akkor csak a komplexitás (addícionális logikával) vagy a válaszidő növelésével(növelem a session idejét) tudjuk figyelembe venni.

3.8.3. Adatfolyam feldolgozás végtelen adaton

Ebben az esetben az adataink rendezetlenek és nem tudjuk explicit megmondani azt az *epszilon* időt, ami az adat létrejötte és feldolgozás között van. 4 kategóriába lehet sorolni az ennél a csoportnál alkalmazott technikákat: időfüggetlen (time-agonostic), közelítő algoritmusok, windowing feldolgozási és windowing az adat létrejötte függvényében.

Időfüggetlen feldolgozás esetében nem vesszük számításba (és nem is fontos) az, hogy mikor érkeznek meg az adatok, mivel *adatvezérelt* módon történik a logika meghatározása. Szűrők és inner-join-ok segítségével dolgozunk. Az előbbi esetben mindig csak a soron következő adatról kell eldöntenünk, hogy megfelel-e a feltételeknek (pl.: adott IP címről érkező adatok kategorizálása), míg az utóbbinál egynél több forrásból érkező adatokat úgy kapcsoljuk össze, hogy az elsőnek beérkezett adatot perzisztens módon eltároljuk.

Becslő algoritmusok mint például Top-n [**topn**] alkotják a második kategóriát az adatfolyam alapú feldolgozó technikáknál. Végtelen adatból egy nagyjából jó, véges kimenetet generálnak, ami egyes esetekben megfelelő adatot eredményez. Ezeknek az algoritmusoknak hátránya, hogy az eredmény nem teljesen jó, illetve általában feldolgozási idő alapján dolgoznak, így a rendezetlen adatoknál pontatlan kimenet lehet az eredmény.

Windowing létrehozásánál megkülönböztetünk az adat születése és feldolgozása felett működő modellt. A feldolgozás alapú ablakoknál a fő problémánk az, hogy az adataink nem sorrendben érkeznek meg a feldolgozó motorhoz, így előfordulhat, hogy egyes adatok lemaradnak, így rontva a feldolgozás eredményességét. Az adat születése alapú ablaknál viszont tekintettel vagyunk arra, hogy mikor születik az adat, így az azonos időben született adatokat együtt tudjuk feldolgozni.

4. Gépi tanulás

Amikor tanulunk, a célünk, hogy minél jobb eredményeket érjünk el a számonkérésen, vagy minél több tudást halmozzunk fel, amit a későbbiek során (valószínűsíthetően) hasznosítani tudunk. A gépi tanulásnak is ugyanez a célja, különböző modellek megalkotása után a megadott példákból (input adat) különböző kimeneteket (output adat) ad ki. Az input adatokból próbál általánosítani oly módon, hogy az felhasználható legyen számára ismeretlen problémák során. Ebből következően minél több bemeneti adatunk és tapasztalatunk (adat) van, annál jobb okosabb és pontosabban fog előrejelezni a használni kívánt algoritmus. Gépi tanulást használunk például:

- Web keresés
- Spam szűrés
- Ajánló rendszerek
- Online hírdetések

esetén is. Egy 2011-es Mckinsey riport **[mckinsey]** szerint a gépi tanulás (illetve a prediktív analitika) lesz a következő évek innovációinak alapja. IBM Watson-ja **[watson]**, már képes a beadott tünetek alapján, megjósolni, hogy mi lehet a páciens betegsége (egyelőre még csak fejlesztőknek, API-n keresztül).

Két fő csoportja van a gépi tanulási algoritmusoknak: a felügyelt (supervised) és nem felügyelt (unsupervised) tanítás.

4.1. Felügyelt tanítás

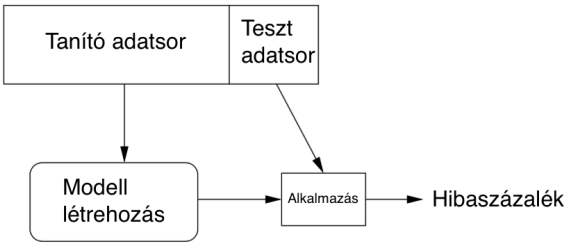
Általánosan fogalmazva, az adat amit betáplálunk a gépi tanulás modellünkbe, tréning példáknak (training set) nevezzük. A tréning példák x, y párokat tartalmaznak, ahol x az érték vektor (feature vector). Minden x

érték: kategórikus (diszkrét értékeksorozatból származik, pl.: {kék, piros, sárga}) vagy numerikus (az érték egész vagy valós szám). y a címke (label), ami kategorizáló érték x-re nézve.A célünk az, hogy felfedezzük azt az

$$y = f(x)$$

függvényt, ahol a legjobban előre tudjuk jelezni az y értéket a meghatározott x-re nézve.

Fontos, hogy szétválasszuk az adatainkat tréning és teszt adatokra. Ez biztosítja azt, hogy ne fordulhasson elő az a probléma, hogy a modellünk túlságosan fontos súllyal vesz egyes objektumokat az adatsoron (amik nem jellemzőek a lehetséges valós adatokra), ami azt eredményezi, hogy a valós problémákon már nem fog eredményesen működni. A problémát túltanulásnak vagy magolásnak (overfitting) nevezik. [**overfit**]



6. ábra. Felügyelt tanulás általános modellje **??**

4.2. Nem felügyelt tanítás

Nem felügyelt tanítás esetén adottak: $(x_1, x_2, ..., x_n)$ adataink, és nincs célfüggvényünk, vagy elvárt kimenetünk. Alapvetően nem struktúrált *zajból* próbálunk mintázatot keresni, olyan modellt létrehozni, ami jól reprezentálja adatok valószínűségi eloszlálást. Annak ellenére, hogy nincs információnk arról, hogy az egyes adatok milyen kapcsolatban vannak egymással, (x_t) valószínűségi eloszlását meg tudjuk jósolni $(x_1, x_2, ..., x_{t-1})$ alapján, ahol $P(x_t|x_1, x_2, ..., x_{t-1})$. Egyszerűbb esetekben, ahol az bemenet sorrend irreleváns, lehet modellt építeni az adatra, ahol $(x_1, x_2, ...)$ az adatsorunk, és ezek függetlenül de egyöntetűen származnak a $P(x)^2$-ból. [**unsupervised**]

4.3. Ajánlórendszerek

Az ajánlórendszerek olyan, főként webes rendszerek, ahol a különböző forrásból származó adatok alapján ajánlunk olyan lehetőségeket a felhasználóknak, ami nagy valószínűséggel megfelel a preferenciájának. Például:

- Youtube, ahol az nézettségi történet alapján kapja az ajánlásokat a felhasználó
- Amazon, ahol az a cél, hogy a felhasználónak olyan termékeket ajánljunk, amit a hozzá hasonló felhasználók (k-legközelebbi analízis segítségével) is preferálnak. [**knearest**]

Két fő fajtáját különböztetjük meg az ajánló rendszereknek. A Content-based (tartalom alapú) ahol az item tulajdonságait vizsgáljuk, illetve a collaborative filtering (együttműködésen alapuló szűrés), ahol a hasonló érdeklődésű felhasználóknak nyújtott ajánlásokat vesszük alapul.

Az ajánlórendszerek alapja az ún. rating matrix, ami tartalmazza a felhasználókat és hozzájuk kapcsolt elemek adatait. Minden felhasználó, elem párhoz egy értéket rendelünk, ami jellemzi, hogy mennyire preferálja az adott elemet. Ez az érték általában egy rendezett skáláról (pl.: 5 elemű skála, 1-től 5-ig számozva, ahol az 1 a legkevésbé, az 5 pedig a legjobban kedvelt elemet jelöli) kerül ki. Alapfeltevés, hogy a mátrixunk ritka (nincs teljesen kitöltve), mivel nem értékel minden felhasználó minden elemet. A hiányzó, nem ismert értékekről semmilyen explicit információval nem rendelkezünk.

	HP1	HP2	HP3	TW	SW1	SW2	SW3
A	4			5	1		
B	5	5	4				
C				2	4	5	
D		3					3

7. ábra. Rating matrix, ahol az A, B, C, D felhasználók a Star Wars, Harry Potter és Twilight filmeket értékelték **??**

Célünk az, hogy hiányzó értékeket a mátrixban minél jobban megjósoljuk. Természetesen nem fontos, hogy az összes elemét kitöltsük, törekednünk kell arra, hogy az ajánlás a preferált filmek/cikkek körében legyen, mivel ezek eladása/elolvasása racionalizálható gazdasági szempontból. Jelen esetben, kíváncsiak lehetünk, hogy A felhasználónak ajánlhatjuk-e Harry Potter 2. részét. Láthatjuk, hogy HP 1. részét kedvelte, és tudjuk, hogy a két film kapcsolatban van egymással (rendezők, színészek, történet, stb.) ezért gyaníthatjuk, hogy a második részt is kedvelni fogja.

4.3.1. Hosszú farok

Amikor bemegyünk egy könyvesboltba, többféle módon is választhatunk egyes könyvek közül. Vannak kiemelt könyvek, amikre nagy az érdeklődés,

ezért a könyvesbolt jobban reklámozza ezeket. Személyes ajánlásra nincs lehetőség, a választék korlátozott, erőforrás hiányában az összes könyvnek csak egy szűk szeletét mutatja meg egy könyvesbolt. Ezzel szemben egy online bolt bármit ajánlhat, ami létezhet, nem csak a populárisabbakat, hanem a kevésbé keresetteket is. Ezt a megkülönböztetést nevezzük *hosszú farok*nak [**longtail**], és ez az, ami létrehozta az ajánlórendszerek igényét. Muszáj ajánlanunk a felhasználónak termékeket –mivel nincs olyan nyilvánvaló módon prezentálva, mint a fizikai boltoknál– ahhoz, hogy nagyobb eséllyel vásároljon belőlük.

4.3.2. Rating mátrix feltöltése

Utilty mátrix nélkül nem lehetséges a felhasználóknak ajánlattétel. Ahhoz, hogy feltudjuk tölteni, két általános megközelítés létezik.

1. Megkérdezzük a felhasználót a véleményéről (pl.: IMDb, ahol a felhasználók a filmeket értékelhetik egy 1-10-es skálán:) [**imdb**]: Ebben az esetben a sikerességünk limitált, mivel a felhasználónak nem származik rövid távon gazdasági előnye abból, hogy értékel (természetesen, ha minden film után reális értékelést adna, akkor hosszú távon jobb ajánlásokat kapna). Másrészt az emberi irracionalitás miatt előfordulhat, hogy részrehajlóan (akár pozitív, akár negatív irányban) értékel, ami eltorzíthatja a rating mátrixot. [**introspection**]

2. Megvizsgáljuk a viselkedését: ebben az esetben azt vizsgáljuk, hogy a felhasználó megtekintette/megvette/stb. az adott terméket. Ha igen, explicit 1-el töltjük fel a mátrixot, ha nem akkor 0-val. Tehát, ha megveszünk (vagy csak megnézünk) egy könyvet az Amazonon, akkor 1-el fogja értékelni a rating mátrixban az algoritmus. Így kiküszöbölhetjük azt, hogy a felhasználó, nem szeretne vagy nem tud helyesen értékelni. Hátránya, hogy nehéz súlyozni a különböző véleményeket, mivel csak 0, 1 intervallum skálán dolgozunk.

4.3.3. ALS

Mikor ajánlásokat adunk a felhasználónak, a célünk az, hogy a rating mátrixban a hiányzó elemeket megkeressük. Feltételezhetjük, hogy a felhasználó elemei között kapcsolat van (mivel a felhasználó preferenciája vélhetően *rögzített* egy rövid időszakon belül), ezért különböző optimalizációs eljárásokat alkalmazhatunk. Az egyik legismertebb ilyen eljárás a mátrix faktorizáció egy fajtája az ALS (ALS, Alternating Least Squares), ahol az M rating mátrixunkat faktorokra, részmátrixokra bontjuk.

Legyen U és I a felhasználók és elemeik sorozata. $U = \{1, ..., U\}$ és $I = \{1, ..., I\}$. Implicit feltöltött rating legyen M, ahol a kitöltött elemek r_{ui}, az üresek (amire szeretnénk ajánlást adni) pedig $\hat{r}_{ui}$. M-et felbonthatjuk két részmátrixra, P és Q, és inicializáljuk elemeiket megfelelően kis k-val. Válasszuk M egy ismert elemét, legyen X. Ha ez a választott X eltér a megfelelő P és Q elem szorzatától, akkor változtassuk meg az eltérés irányában. Ha X értéke megegyezik a vizsgált P és Q faktor szorzatával, akkor válasszunk másik X-et.

$$\begin{bmatrix} 1 & 1 \\ 1 & 1 \\ 1 & 1 \\ 1 & 1 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \end{bmatrix} = \begin{bmatrix} 5 & 4 & ? & ? & 5 \\ ? & 4 & ? & ? & ? \\ 5 & ? & ? & ? & 4 \\ 5 & 4 & 5 & ? & 5 \\ 5 & ? & 4 & ? & 5 \end{bmatrix}$$

Ha egyszerre változtatjuk a két részmátrix elemeit, akkor az NP-nehéz probléma, ezért mindig csak az P részmátrixon iterálunk végig amíg P értékei fixálvan vannak. Ha megfelelően jó eredményt kapunk, akkor váltunk a Q-ra, ahol megismételjük ezt a műveletet, amíg el nem érjük a konvergencia állapotát. Alapvető probléma még emellett, hogy ha a részmátrixon a változtatás túl kicsi, akkor rengeteg erőforrást elpazarlunk, ha túl nagy, akkor lehet, hogy átugorjuk az optimális megoldást.

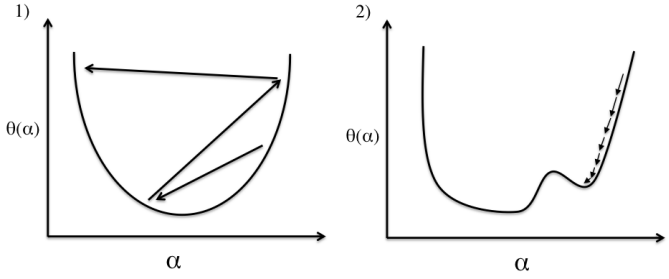
4.3.4. Gradiens leszállás

Ahhoz, hogy nagy adatmennyiségben a lehető legkevesebb költséggel (idő, számítási) tudjunk dolgozni, különböző optimalizációs eljárásokat alkalmazunk. A dolgozatomban vizsgált gradiens leszállás egy olyan metódus, aminek a feladata, hogy egy általában konvex célfüggvény lokális (vagy optimális esetben globális) minimumát megtalálja. Legyen $J(\theta)$ a célfüggvény, aminek a minimumát keressük, α a tanulási ráta és $j = 0, 1, ..., n$ pedig a függvény paraméterei:

$$\theta_j := \theta_j - \alpha \sum_{i=1}^{n} \frac{\delta}{\delta \theta_j} J(\theta)$$

[**andrewml**] Veszek egy kezdeti véletlenszerű értéket, ami a függvény paraméter(einek) értéke lesz. Az algoritmus megvizsgálja a paraméterek gradiensét (parciális deriváltját), ami megadja, hogy milyen irányban kell csökkentenem a paramétert ahhoz, hogy a célfüggvényem is csökkenjen. Ezután kiválasztok egy lépésközt (tanulási ráta), ami megadja, hogy mekkora mértékben

változtatom a paramatéreket. Addig ismétlem a folyamatot, míg konvergenciára nem jutok vagy elérem az előre definiált ϵ küszöböt. Fontos, hogy két ismétlés (update) között az összes paramétert változtatom.
Az α tanulási ráta megadja, hogy mekkora lépésközökkel operál az algoritmusom, tehát mekkorát csökkentek (vagy növelek) az adott változóm értékén. Ha túl nagyra veszem, akkor lehet, hogy váltakozva fogok divergálni és konvergálni, míg ha túl kicsire veszem, lehet, hogy csak a lokális minimumot találom meg.

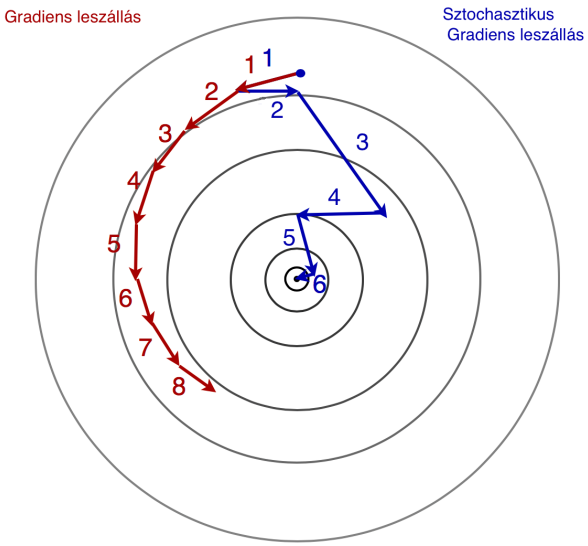


8. ábra. Konvergencia eltévesztése túl nagy és túl kicsi tanulási rátánál. **??**

4.3.5. Stochasztikus Gradiens leszállás

Gradiens leszállás egy jól müködő, egyszerű algoritmus, mellyel líneárisan tudok konvergenciára jutni. Azonban alapvető probléma vele, hogy nagy adatmennyiségnél („big data") költséges a megvalósítása, mivel minden egyes paramétert ki kell számolnunk ahhoz, hogy a következő iterációra jussunk. Robbins és Monroe 1951-ben [**stoch1951**] írták a tanulmányukat, melyben az úgynevezett sztochasztikus optimalizációs eljárást vizsgálták. Ebben arra jutottak, hogy nem szükséges minden lépésben az összes paramétert kiszámolni, elég véletlenszerűen venni mintákat az adatsorból. Ezt nevezzük sztochasztikus gradiens leszállásnak (SGD), ami a gradiens algoritmus egy bővített, egyszerűsített változata.
Miért jó ez? Ha minden lépésünk csak 1 deriválttól függ (tehát 2 iteráció között csak 1 paraméter változásával számolunk), az jelentősen megtudja gyorsítani (illetve a számítási költségeket csökkenteni) a folyamatot, mivel a gradienshez képest $1/n$-es komplexitásról beszélünk, ha n jelöli a tanuló adatsorunk számosságát. [**asgd**]. E mellett arra is alkalmas a véletlenszerű mintavétel, hogy a zajos és redundáns adatokat kiszűrje. [**gatsby**]

A 9 ábrán láthatjuk, hogyan alakulhat a konvergencia elérés a két vizsgált algoritmusnál.



9. ábra. Gradiens és Sztochasztikus Gradiens konvergencia elérés minta **??**

Az SGD általános modellje, $J(\theta)$ költségfüggvény minimalizálása esetén:

$$\theta_j := \theta_j - \alpha \frac{\delta}{\delta \theta_j} J(\theta)$$

Természetesen a csökkent komplexitással hátrányok is járnak. A véletlenszerű mintavétel miatt, a konvergálás véletlenszerű lesz, a variancia pedig nagy ami nem líneáris, de gyors konvergenciához vezet. Ezzel szemben a gradiens leszállással, ahol líneáris, de lassú konvergenciával dolgozunk. A pontossága sem lesz mérhető a gradiens leszálláshoz, mivel ha el is éri a lokális (vagy globális) minimumot, oszcilláló mozgást fog végezni körülötte (hacsak nem csökkentjük le az α tanulási rátát közel 0-ra. [**bottou**]) Az, hogy mikor éri meg SGD-t alkalmazni mindig az adott üzleti alkalmazástól függ, mérlegelnünk kell, hogy a pontosságot feláldozzuk-e a sebesség oltárán.

5. Megvalósítás

5.1. Java, Scala

Dolgozatomban Java és Scala nyelvek alatt fogom implementálni a sztochasztikus gradiens leszállás algoritmust, mivel a Flink is Java alapokon írodott. Java nagyon erős támogatással bír az Apache alapítvány részéről, a 370 támogatott projektből 222 Java alapú. [**asf**] E mellett a rendelkezik olyan tulajdonságokkal, ami a nagyobb, elosztott rendszerek fejlesztésének kezdeténél

–'90-es évek közepe és vége– nagyon kevés nyelv rendelkezett:

- Objektum orientált alapok, mely a nagy komplexitású projektek felépítését megkönnyíti
- A beépített függvénykönyvtár erős hálózati (TCP/IP, UDP, HTTP, stb) támogatással rendelkezik
- Platformfüggetlenség
- Kivétel és hibakezelés, try/catch
- Hálózati hatás: a Java nyelv elterjedtsége nagyon magas, az egyik legnépszerűbb nyelv a világon **[tiobe]**, ami azt jelenti, hogy nagyon kiterjedt és fejlett külső keretrendszer jellemzi.

Scala –Scalable Language kifejezésből származtatva **[odersky]**– az elmúlt pár évben lett közismert a nagy adatfeldolgozó, elosztott projektek kapcsán. A vegyes objektum-orientált és funkcionális megközelítés lehetővé teszi, hogy a programozók könnyen átálljanak a nyelvre, mégis kihasználjanak újfajta paradigmákat (first-class functions, immutable data structures, immutability over mutation). Java Virtuális Gép (JVM) alapokon írodott, ezért lehetővé teszi, hogy a megszokott Java könyvtárakkal is dolgozzunk, és emiatt közel azonos sebességre is képes. E mellett típus-biztos (type-safe), tehát a fordítási időben ellenőrzi a változók típusát és értékét, így elkerülve a futási időben keletkező hibákat [**toptal**].

5.2. Flink

Az Apache alapítvány által támogatott Flink egy olyan valós idejű adatfolyam feldolgolozó architektúra, ami biztosítja, hogy valós időben, végtelen adaton is kellő pontossággal tudjunk műveleteket végezni, anélkül, hogy bonyolult, csak a feladatra tervezett architektúrát kellene létrehoznunk. Ezt az egyszerűséget és pontosságot következő tulajdonságaival éri el **[flink]**

- Egyszerűsített adatfeldolgozó csővezeték (pipeline), amely az adat megkapásától az adat feldolgozásáig tart
- Az adatot olyan módon modellezi és dolgozza fel, ahogy az létrejött, valós-időbeli események alapján
- Nem sorrendben érkezett adatokból származó hibák kiküszöbölése azzal, hogy esemény alapú ablakok létrehozását támogatja

- Biztosítja, hogy a lehető leghamarabb de szükségszerűen legkésőbb történjen az adatfeldolgozás, szem előtt tartva a még meg nem érkezett adatokat

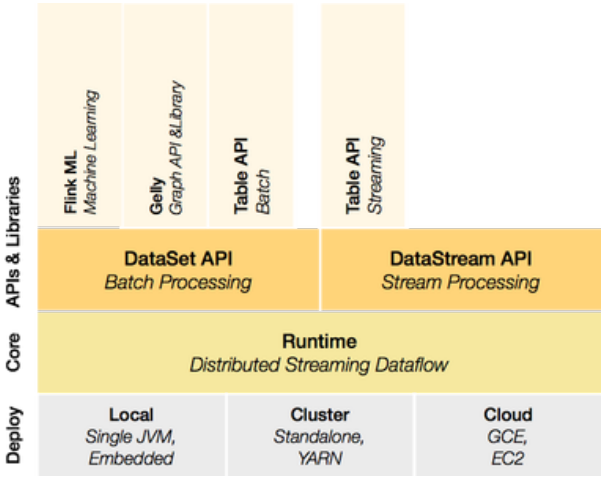
Az eddig elterjedt megoldások többségében csak kötegelt (Apacha Spark) **??** vagy csak adatfolyam feldolgozásra (Apache Storm**??**, IBM Infosphere**??**) használhatóak. Flink alapvetően váltja meg ezt a paradigmát, egyszerre képes kötegelt és adatfolyam feldolgozásra is, úgy, hogy a programozási modell és futtató környezet is megegyezik a két esetnél. A megismert esemény és feldolgozási idő mellett egy új fogalmat is bevezet: fogyasztási idő? (ingestion time)**??** ami egy olyan hibrid megoldás, It assigns wall clock timestamps to records as soon as they arrive in the system (at the source) and continues processing with event time semantics based on the attached timestamps.

A kötegelt feldolgozást egy, az adatfolyam specifikus részének tekinti, így tudja megvalósítani a közös architektúrát. Négy fő részről beszélhetünk, a bevezetés (deployment), központ (core), API és könyvtárak. A központi részben találhatjuk meg az elosztott adatfolyam motort, ami végrehajtja a az adatfolyam programokat. Futtatási időben, egy irányított körmentes gráf (DAG) állapotartó operátorokhoz csatlakozik az adatfolyamokon. Az API DataSet és DataStream részekre van osztva melyek a kötegelt és folytonos adatfeldolgozásért felelnek. Különböző domain specifikus nyelvekkel (DSL) oldották meg felsőbb rétegű függvénykönyvtárak alkalmazását: FlinkML (gépi tanulás), Gelly (gráf feldolgozás) és Table (SQL jellegű lekérdezések) könyvtárak állnak a felhasználók rendelkezésére. Futásidőben három típusú folyamatot kezelünk:

- Kliens: Program kódot átalakítja adatfolyam gráffá, és elküldi a Job Managernek
- Job Manager: Felel az elosztottságért és nyilvántartja a program állapotát és előrehaladását minden operátornak és adatnak, e mellett új operátorok időzítéséért, ellenőrző pontok felállításáért és a rendszer felállításáért is felel.
- Task Manager: Adatfeldolgozás a fő feladata

6. Fejlesztési lehetőségek

Konvergencia minél gyorsabb elérése (accelerated sgd).



10. ábra. Apache Flink felépítése **??**