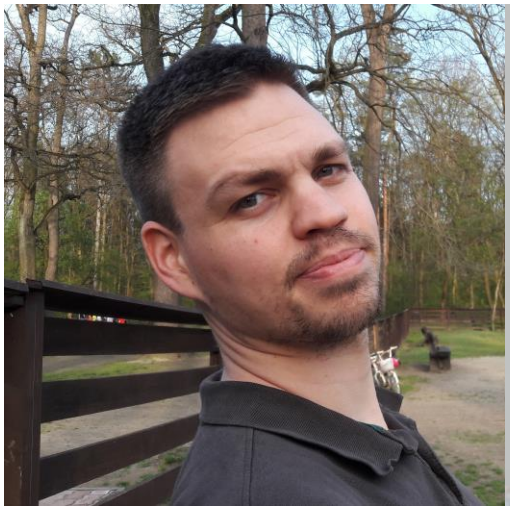



Dr. Hegedűs Péter, Dr. Ferenc Rudolf

Nagyméretű adatbázisok

Jelen tananyag a Szegedi Tudományegyetemen készült az Európai Unió támogatásával.

Projekt azonosító: EFOP-3.4.3-16-2016-00014

Szegedi Tudományegyetem
Cím: 6720 Szeged, Dugonics tér 13.
www.u-szeged.hu
www.szechenyi2020.hu

Az Apache Spark működésének részletei

Összefoglalás

Az olvasó ebből az olvasóleckéből az Apache Spark általános adat elemző és kiértékelő feladatokat végrehajtó motorról kap részletes információt. Áttekintjük, milyen alapvető koncepciókat és adat struktúrákat használ a Spark, valamint sorra vesszük az összes főbb komponensét. Részletesen foglalkozunk az RDD adatszerkezettel, a DataFrame koncepcióval, valamint a Spark által támogatott 80 operátor közül a fontosabbakkal, amelyek segítségével adat feldolgozó feladatok implementálhatók. Az olvasó arról is képet kap, milyen nyelveken és milyen API segítségével készíthet Spark programokat.
A lecke fejezetei:

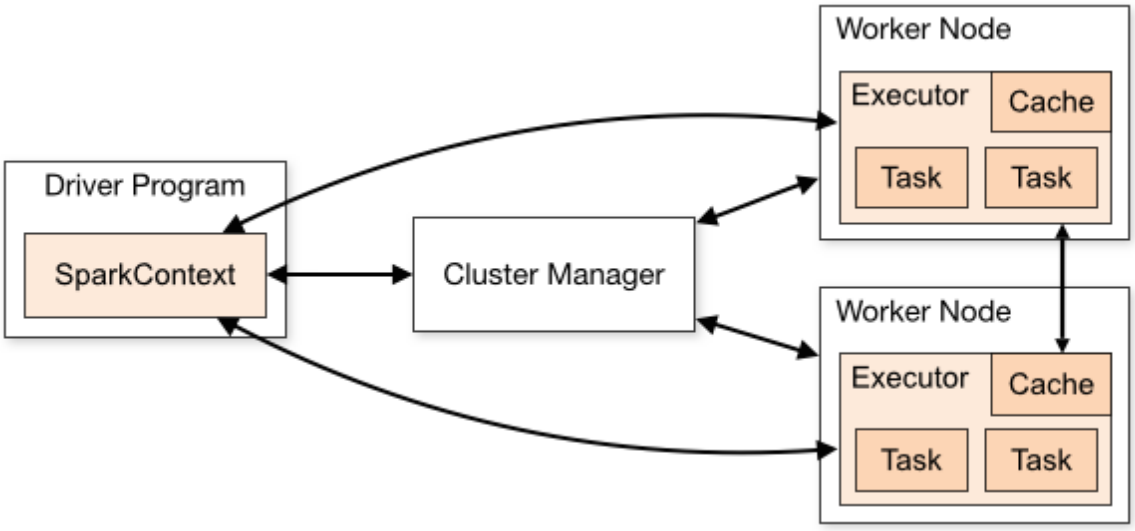
- 1. fejezet: **A Spark egyik központi absztrakciójának, az RDD-nek a bemutatása (olvasó)**
- 2. fejezet: **Az RDD-ken végezhető gyakoribb transzformációk és akciók ismertetése (olvasó)**
- 3. fejezet: **Egyéb kapcsolódó fogalmak, mint az RDD perzisztálás vagy Dataset/DataFrame (olvasó)**

Téma típusa: **elméleti**
Olvasási idő: **45 perc**

📖 1. fejezet
Resilient distributed dataset (RDD) fogalma

Az Apache Spark [1] egy villámgyors klaszter számítási keretrendszer, amit nagyon gyors adatfeldolgozásra terveztek. A Hadoop MapReduce modellen alapul (koncepcionálisan, nem kód szintjén), de olyan módon általánosítja és terjeszti ki azt, ami lehetővé teszi a hatékony felhasználását interaktív lekérdezések készítéséhez vagy stream feldolgozáshoz is. Az alapvető technika, ami ezt a fajta sebesség növekedést lehetővé teszi, a memóriában tárolt klaszter számítási modell.

Magas szinten minden Spark alkalmazás egy *meghajtó programból (driver program)* áll, ami a felhasználó által definiált `main` függvényt futtatja, valamint több különböző *párhuzamos műveletet* hajt végre a klaszteren (lásd lenti ábra). A Spark által nyújtott legfontosabb absztrakció az ún. *resilient distributed dataset* (RDD), a klaszter csomópontjain elosztva tárolt elemek gyűjteménye, melyeken párhuzamosan végezhetők műveleteket. RDD-k létrehozhatók fájlok betöltésével a Hadoop fájlrendszerről vagy bármilyen Hadoop kompatibilis adattárból, vagy a driver programban már létező kollekció transzformálásával. A felhasználók kérhetik az RDD-k kimentését a memóriába, ezáltal az elérhetővé válik más párhuzamosan futó művelet számára is. Az RDD-k automatikusan helyreállítódnak a csomópontok meghibásodása esetén.



> https://spark.apache.org/docs/latest/img/cluster-overview.png

A Spark másik fontos absztrakciója az ún. *megosztott változók (shared variables)*, amelyek párhuzamos műveletek során használhatók. Alapértelmezésként amikor a Spark egy függvényt futtat több feladatként csomópontokon szétosztva, a függvény által használt minden változót átmásol minden feladathoz. Néha előfordul, hogy egy változót meg kell osztani különböző feladatok között, vagy egy feladat és a driver program között. A Spark két fajta megosztott változó típust támogat:

- *Broadcast variables* - ez a fajta megosztás minden csomópont memóriájában eltárol egy értéket (cache)
- *Accumulators* - olyan változók, amelyek értékéhez csak "hozzáadni" lehet, például számlálók vagy összegek

Az RDD-k két fajta műveletet támogatnak:

- *Transzformáció (transformation)* - egy létező RDD-ből egy másik RDD létrehozása
- *Akció (action)* - egy művelet végrehajtása az RDD-n, és a kiszámított érték visszaadása a driver program számára

A `map` művelet például egy transzformáció, amely az RDD minden egyes elemére meghív egy függvényt, és a függvény által visszaadott értékekből egy újabb RDD-t hoz létre. Másrészről a `reduce` művelet egy akció, ami egy RDD összes elemén valamilyen összegző műveletet végez el egy függvény segítségével, és a végeredményt visszaküldi a driver program számára (noha létezik egy párhuzamos `reduceByKey` művelet is, ami vissza is ad egy RDD-t).

Spark-ban minden transzformáció *laza (lazy)*, ami azt jelenti, hogy nem számítják ki azonnal az eredményüket. E helyett csak megjegyzik az alkalmazandó műveleteket egy adathalmazon (pl. egy fájl). A transzformáció ténylegesen csak akkor fog megtörténni, ha egy akció vissza szeretné küldeni a kiszámított eredményt a driver program számára. Ez a tervezési döntés hatékonyabbá teszi a Spark feladatok végrehajtását. Például ha egy `map` művelet után egy `reduce` akció is történik, akkor elég csak a `reduce` eredményét visszaküldeni a driver programnak a teljes transzformált RDD helyett.

Alapértelmezetten minden transzformált RDD újra számítódhat egy akció futtatásakor, azonban lehetőség van az RDD-k mentésére is a `persist` (vagy `cache`) műveletek segítségével, amely esetén a Spark az RDD elemeit a klaszter memóriájában tárolja a későbbi sokkal gyorsabb feldolgozáshoz. Továbbá lehetőség van az RDD-k merevlemezre mentésére is, valamint azok replikálására klaszteren belüli több csomópontra.

A következő kis kódrészlet az RDD-k használatának alapjait mutatja be a Java Spark API-n keresztül:

```
JavaRDD<String> lines = sc.textFile("data.txt");
JavaRDD<Integer> lineLengths = lines.map(s -> s.length());
int totalLength = lineLengths.reduce((a, b) -> a + b);
```

Az első sor egy kiindulási RDD-t hoz létre fájl beolvasás segítségével. Ez egyelőre még nem hozza létre az RDD-t memóriában, sőt semmi nem történik vele, a `lines` egyszerűen csak egy referencia lesz a fájlra. A második sor definiálja a `lineLengths`-et, mint a kiinduló RDD-n elvégzett `map` művelet eredménye. A `lineLengths` szintén nem számítódik ki azonnal a Spark lazy kiértékelési stratégiája miatt. Végezetül lefuttatjuk a `reduce` műveletet, ami egy akció. Ezen a ponton a Spark szétbontja a számítást kisebb feladatokra, amelyeket a klaszter különböző gépein futtat, amelyek mind a `map` mind a `reduce` műveletek rájuk eső részeit lefuttatják, és csak az eredményt küldik vissza a driver programnak.

Amennyiben a későbbiek során újra szeretnénk használni a `lineLengths` RDD-t, a következő sort is hozzáadhatjuk a programhoz:

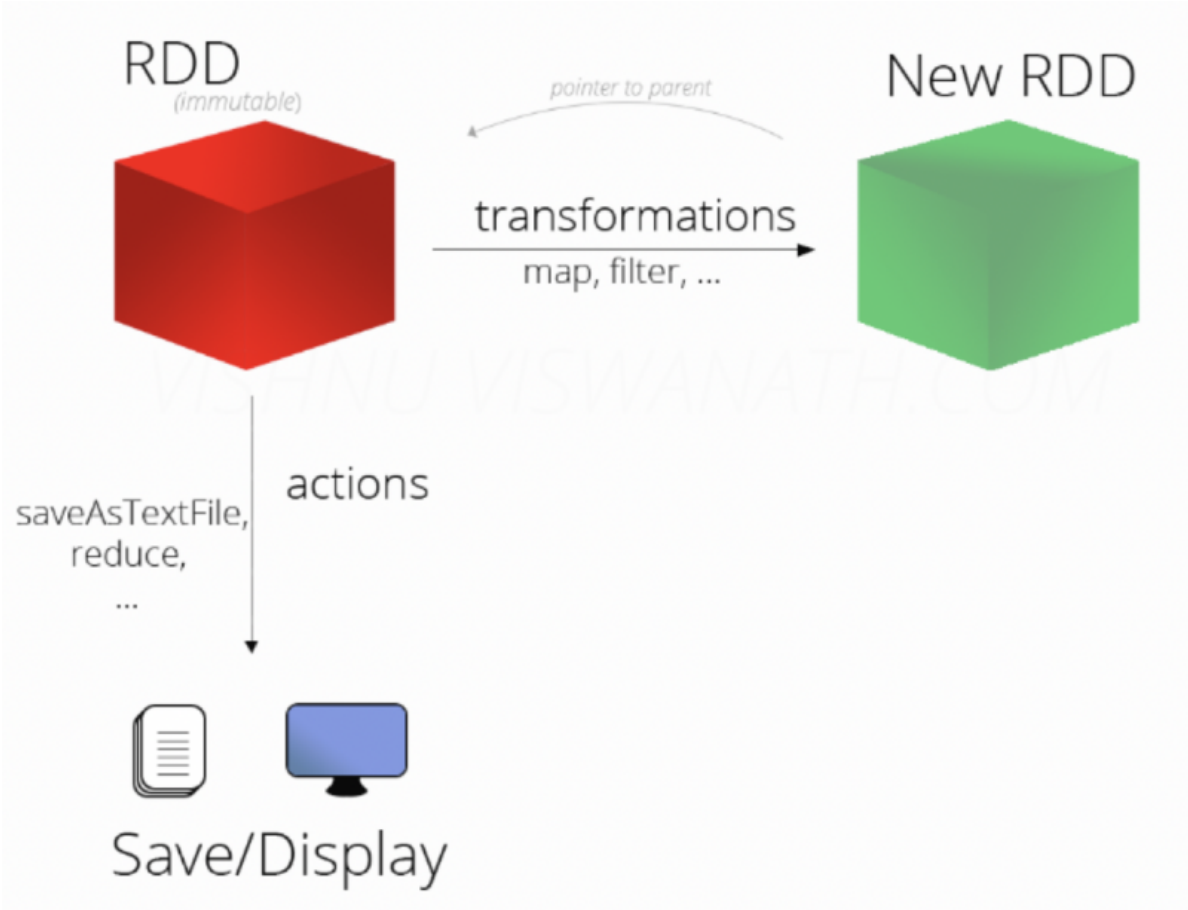
```
lineLengths.persist(StorageLevel.MEMORY_ONLY());
```

Ezt a `reduce` művelet elé beszúrva a `lineLengths` eltárolódik a memóriában miután első alkalommal kiértékelődött.

📖 2. fejezet

RDD transzformációk és akciók

Az RDD-ken végezhető két művelet típus a transzformáció és akció. Ezen műveletek egy szemléltetése látható a lenti ábrán. A kiinduló RDD-k csak olvasható (immutable) adathalmazok, amelyből transzformációval újabb RDD-k állíthatók elő, vagy akciókkal az RDD elemein valamilyen számítási művelet végezhető el.



> https://imgs.developpaper.com/imgs/852248641-5db6e5a679a65_articlex.png

Transzformációk

Az alábbi táblázatban összefoglaljuk a Spark által támogatott leggyakrabban használt transzformációs műveleteket. Bővebb részletekért lásd a hivatalos dokumentációt [2].

Transzformáció	Jelentés
map(*func*)	A forrás RDD minden egyes elemére végrehajtja a *func* függvényt, és annak az eredményeiből összeállítja a transzformált RDD-t.
filter(*func*)	Az előálló RDD a forrás RDD csak azon elemeit tartalmazza, amelyekre a *func* függvény igaz értéket ad vissza.
flatMap(*func*)	A `map`-hez hasonló művelet, de minden bemenő elemhez 0 vagy több kimenet is rendelhető, nem csak egy (azaz a *func* egy sorozatot ad vissza egyetlen elem helyett).
mapPartitions(*func*)	A `map`-hez hasonló művelet, de ez az RDD minden egyes adat partícióján (blokk) külön fut, tehát a *func* típusának Iterator => Iterator kell lennie, ha T típusú elemek RDD-jén futtatjuk.
mapPartitionsWithIndex(*func*)	Hasonló a `mapPartitions`-höz, de egy egész értéket is meg kell adni a *func* függvénynek, ami a partíció index-ét jelöli, tehát a *func* típusának (Int, Iterator) => Iterator kell lennie, ha T típusú elemek RDD-jén futtatjuk.
sample(*withReplacement*, *fraction*, *seed*)	Az adatok egy *fraction* részét mintavételezi, helyettesítéssel vagy anélkül, a megadott véletlenszám generátor seed értékkel.
union(*otherDataset*)	Egy új RDD-t ad vissza, amely a forrás RDD és a paraméterben megadott másik RDD elemeinek unióját tartalmazza.
intersection(*otherDataset*)	Egy új RDD-t ad vissza, amely a forrás RDD és a paraméterben megadott másik RDD elemeinek közös metszetét tartalmazza.
distinct([*numPartitions*]))	Egy RDD-t ad vissza, amely a forrás RDD különböző elemeit tartalmazza.

Transzformáció	Jelentés
groupByKey([*numPartitions*])	Ha egy (K, V) típusú párokat tartalmazó RDD-re hívjuk meg a transzformációt, akkor az egy (K, Iterable) párokból álló RDD-t ad vissza. **Megjegyzés:** ha összegző művelet (szumma, átlag) elvégzése céljából végezzük a csoportosítást minden kulcs mentén, célszerűbb a `reduceByKey` vagy `aggregateByKey` transzformációkat használni, mert azok sokkal hatékonyabbak ebben az esetben. **Megjegyzés:** A párhuzamosítás szintje alapesetben a szülő RDD partícióinak számától függ. Ha ettől eltérő számú feladatot szeretnénk, azt a `numPartitions` paraméter megadásával tehetjük meg.
reduceByKey(*func*, [*numPartitions*])	Ha egy (K, V) típusú párokat tartalmazó RDD-re hívjuk meg a transzformációt, akkor eredményül egy olyan szintén (K, V) párokból álló RDD-t kapunk, ahol minden kulcshoz úgy állnak elő az értékek, hogy az eredeti RDD azonos kulcsú értékeire meghívódik a *func* reduce függvény, aminek a típusa (V, V) => V kell, hogy legyen. Ahogyan a `groupByKey` esetében is, a reduce feladatok száma egy opcionális paraméterrel állítható.
aggregateByKey(*zeroValue*)(*seqOp*, *combOp*, [*numPartitions*])	Egy (K, V) párokat tartalmazó RDD-n meghívva a transzformáció egy (K, U) párokból álló RDD-t ad vissza, ahol az egyes kulcsokhoz tartozó értékek a megadott combine függvény és semleges "nulla" értéket felhasználva aggregálódnak. Ez a transzformáció tehát lehetővé teszi, hogy az aggregált érték típusa különbözzön a bemenő értékek típusától, elkerülve a felesleges helyfoglalást. Ahogyan a `groupByKey` esetében is, a reduce feladatok száma egy opcionális paraméterrel állítható.

Transzformáció	Jelentés
sortByKey([*ascending*], [*numPartitions*])	Egy (K, V) párokat tartalmazó RDD-n meghívva, ahol a K implementálja az Ordered interfészt (azaz sorba rendezhető), egy szintén (K, V) párokból álló RDD-t ad vissza, amelyikben az elemek a K kulcs érték alapján növekvő vagy csökkenő sorrendbe vannak rendezve a megadott `ascending` paraméter értékének függvényében.
join(*otherDataset*, [*numPartitions*])	Egy (K, V) párokat tartalmazó RDD-n meghívva, ahol a paraméterben kapott másik RDD (K, W) típusú párokat tartalmaz, egy (K, (V, W)) párokból álló RDD-t ad vissza, ahol minden azonos kulcsú elem pár szerepel a két adathalmazból minden kulcsra. A külső join műveleteket (outer join) a `leftOuterJoin`, `rightOuterJoin`, és `fullOuterJoin` valósítják meg.
cogroup(*otherDataset*, [*numPartitions*])	Egy (K, V) párokat tartalmazó RDD-n meghívva, ahol a paraméterben kapott másik RDD (K, W) típusú párokat tartalmaz, egy (K, (Iterable, Iterable)) hármast ad vissza. Ezt a műveletet még `groupWith`-nek is nevezik.
cartesian(*otherDataset*)	Ha egy T és U típusú elemeket tartalmazó adathalmazon hívják meg, ez a transzformáció visszaadja az elemek összes lehetséges (T, U) párját egy RDD-ben.
pipe(*command*, *[envVars]*)	Ez a transzformáció az RDD minden partícióját egy shell utasításba irányítja, pl. egy Perl vagy bash szkriptbe. Az RDD elemei a szkript standard bemenetére íródnak, és a szkript eredménye a standard kimenetre íródik, amiből egy string elemekből álló RDD készül. **Megjegyzés**: vesd össze a Hadoop streaming-gel!
coalesce(*numPartitions*)	Lecsökkenti az RDD partícióinak számát a paraméterben megadott értékre. Lehetővé teszi a műveletek futtatását egy nagyobb adathalmaz filterezése után.

Transzformáció	Jelentés
repartition(*numPartitions*)	Véletlenszerűen újra keveri az adatokat az RDD-ben, hogy több vagy kevesebb partíciót hozzunk létre és kiegyensúlyozzuk azokat. Ez az újra keverés mindig a hálózaton keresztül történik.
repartitionAndSortWithinPartitions(*partitioner*)	Újra partícionálja az adatokat a megadott partícionáló alapján, és minden partíción belül sorba rendezi az elemeket kulcs szerint. Ez hatékonyabb, mint mehívni a `repartition` műveletet, majd minden partíciót rendezni, mert ez az újrakeverés művelet során már el tudja végezni a rendezést.

Akciók

Az alábbi táblázatban a Spark által támogatott leggyakrabban használt akciókat foglaljuk össze. Bővebb részletekért lásd a hivatalos dokumentációt [2].

Akció	Jelentés
reduce(*func*)	A paraméterben kapott func függvény (aminek két bemenő argumentuma van, és egy kimenő) segítségével aggregálja az elemeket. A függvénynek kommutatívnak és asszociatívnak kell lennie, hogy helyesen kiszámítható legyen párhuzamosan.
collect()	Az RDD összes elemét visszaadja a driver program számára egy tömbben. Ez általában filterezés vagy egyéb olyan művelet után hasznos, ami kellően kis számú elemet ad vissza.
count()	Az adathalmazban lévő elemek számát adja vissza.
first()	Az adathalmaz első elemét adja vissza(a `take(1)` -hez hasonlóan).
take(*n*)	Az adathalmaz első `n` elemét adja vissza egy tömbben.
takeSample(*withReplacement*, *num*, [*seed*])	Egy `num` elemszámú véletlen mintát ad vissza az adathalmazból, helyettesítéssel vagy anélkül, opcionálisan egy véletlen szám generátor seed-et használva.
takeOrdered(*n*, *[ordering]*)	Az RDD első `n` elemét adja vissza az elemek természetes rendezését vagy egy saját összehasonlítót felhasználva.
saveAsTextFile(*path*)	Az adathalmaz elemeit kiírja egy vagy több szövegfájlba a megadott könyvtárba a lokális fájlrendszerre, HDFS-re, vagy bármely más Hadoop által támogatott fájlrendszerre. A Spark minden elemre meghívja annak a toString metódusát, hogy szöveggé konvertálja a sor kiíráskor.
saveAsSequenceFile(*path*) (Java and Scala)	Az adathalmaz elemeit egy Hadoop SequenceFile-ba írja ki egy megadott útvonalra a lokális fájlrendszerre, HDFS-re, vagy bármely más Hadoop által támogatott fájlrendszerre. Ez a művelet a kulcs-érték párokat tartalmazó RDD-kre alkalmazható, amik implementálják a Hadoop Writeable interfészét. Scala nyelven azokra a típusokra is használható, amelyek implicit módon konvertálhatók Writeable-lé (a Spark tartalmaz néhány alap konverziót, pl. Int, Double, String, stb. típusokra).
saveAsObjectFile(*path*) (Java and Scala)	Az elemeket egyszerű formában, Java szerializációt felhasználva írja ki, amely később a `SparkContext.objectFile()` hívással betölthető.
countByKey()	Csak a (K, V) típusú párokat tartalmazó RDD-k esetében támogatott. Egy (K, Int) Hashmap-ben visszaadja az egyes kulcsok számát.

Akció	Jelentés
foreach(*func*)	Az adathalmaz minden elemére lefuttatja a *func* függvényt. Ezt többnyire valamilyen mellékhatás érdekében hívják, mint pl. egy akkumulátor frissítése vagy külső adattár elérése. **Megjegyzés**: az akkumulátorokon kívül más változók módosítása nem várt viselkedéshez vezethet [3]!

📖 3. fejezet

RDD-hez kapcsolódó további tudnivalók

RDD perzisztencia

A Spark keretrendszer egyik legfőbb képessége az adathalmazok műveleteken átívelő *kimentése/perzisztálása* (*gyorsítótárazása*). Ha valaki perzisztál egy RDD-t, minden node az általa tárolt partíciókat lementi a memóriába és felhasználja azt, ha valaki az adathalmaz azon partíciójához szeretne hozzáférni. Ez az RDD-n végzett későbbi műveleteket nagy mértékben meggyorsítja (gyakran több mint 10x). Ez a fajta gyorsítótárazás kulcsfontosságú az iteratív és interaktív algoritmusok megvalósításában.

Egy RDD kimentését a `persist()` vagy `cache()` hívással kezdeményezhetjük, ami az első kiszámítás után a csomópontok memóriájába menti az RDD-t. A Spark gyorsító tára hibatűrő, ha bármelyik partíció elveszne, az automatikusan újraszámolódik az eredeti transzformáció alapján, ami az RDD-t előállította.

Ezen felül minden RDD különböző *tárolási szinttel (storage level)* rendelkezik. Ez lehetővé teszi például az adathalmaz merevlemezen történő tárolását, vagy a memóriában tartását, de szerializált Java objektumként (helytakarékosságból), esetleg más node-okra történő replikálását. Ezek a tárolási szintek a `persist()` metódus számára egy `StorageLevel` objektum átadásával szabályozhatók. A `cache()` hívás egy rövidítés az alapértelmezett tárolási szint használatához, ami a `StorageLevel.MEMORY_ONLY` (deszerializált objektumok memóriában tartása). Az alábbi tárolási szintek támogatottak:

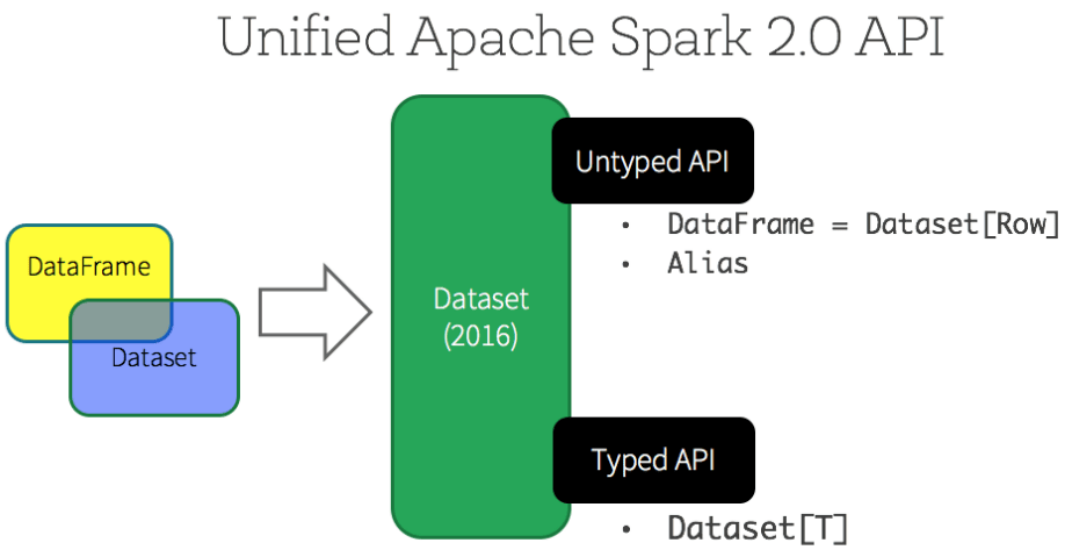
- MEMORY_ONLY - deszerializált Java objektumok tárolása memóriában (ha egy partíció nem fér el, nem lesz tárolva)
- MEMORY_AND_DISK - deszerializált Java objektumok tárolása memóriában (ha egy partíció nem fér el, merevlemezen tárolódik)
- MEMORY_ONLY_SER (Java and Scala) - szerializált Java objektumok tárolása memóriában (helytakarékosabb, de CPU intenzív az olvasásuk)
- MEMORY_AND_DISK_SER (Java and Scala) - mint a MEMORY_ONLY_SER, de a partíciók, amik nem férnek el a memóriában, merevlemezen tárolódnak
- DISK_ONLY - RDD tárolása merevlemezen
- MEMORY_ONLY_2, MEMORY_AND_DISK_2, etc. - a fentiekhez hasonló, de minden partíció replikálandó két vagy több csomópontra.

Dataset-ek és DataFrame-ek

A Spark 1.6-ás változatától kezdve egy új adat absztrakció és API került bevezetésre az RDD-k fölé. Ez a Dataset API, ami az RDD-khez hasonlóan adatok egy nem módosítható, elosztott kollekciója. A Dataset-ek hatékonyan támogatják az optimalizált SQl lekérdezéseket. JVM objektumokból vagy transzformációk segítségével állíthatók elő. Az RDD és Dataset-ek között könnyű az átjárás, néhány API hívással konvertálhatók. A Dataset erősen típusos, így csak a Java és Scala nyelvek számára elérhető ez az API.

A DataFrame nem más, mint egy Dataset, amiben az adatok itt névvel ellátott oszlopokba vannak rendezve, és az adathalmaz több sorból áll, pont úgy, mint egy relációs adatbázis esetén [4, 5]. Nagyon hasonlít a Python/R DataFrame-hez, csak sokkal gazdagabb optimalizációval ellátva. DataFrame-eket sok fajta forrásból létre tudunk hozni, pl. strukturált fájlok, Hive táblák, külső adatbázisok, RDD-k. A DataFrame API már nem csak Java-ban és Scala-ban, de Python-ban és R-ben is elérhető.

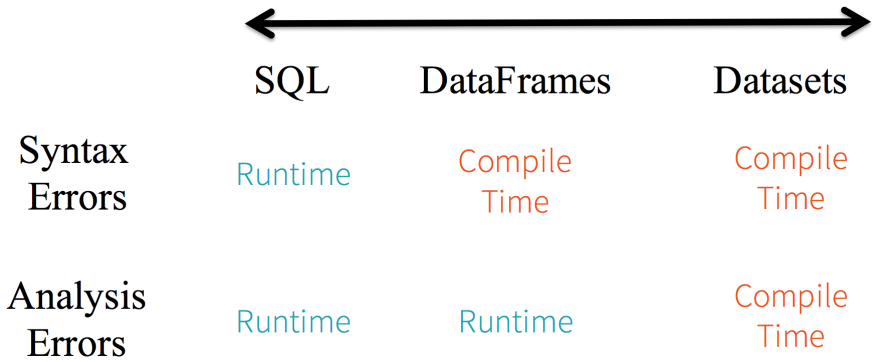
A 2.0-ás Spark verziótól kezdve a típusos és nem típusos Dataset API összevonásra kerültek, a DataFrame nem más, mint egy alias a Dataset típusra, azaz generikus (típus nélküli JVM) objektumok egy adathalmaza (lásd lenti ábra).



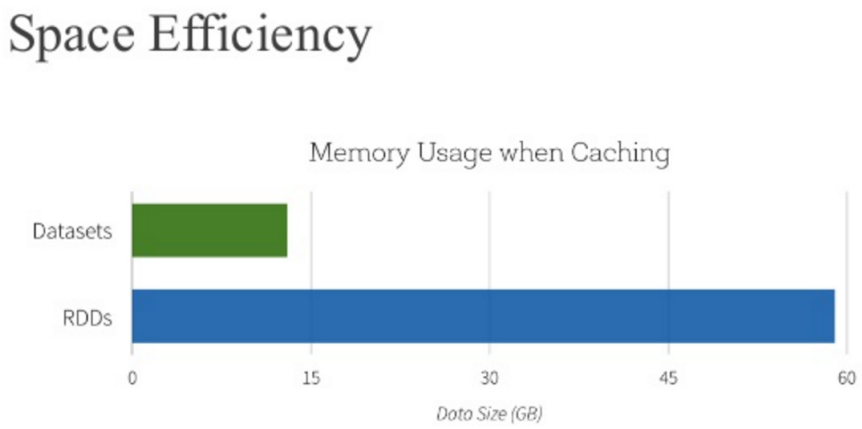
> https://databricks.com/wp-content/uploads/2016/06/Unified-Apache-Spark-2.0-API-1.png

A Dataset API főbb előnyei:

- Statikus típusosság és futás idejű típus ellenőrzés (lásd ábra)



- Magas szintű absztrakció, adatok egyedi nézete strukturált és félig strukturált adatként
- Strukturált adatok API használatának megkönnyítése
- Teljesítmény és optimalizálás (lásd ábra)



Ellenőrző kérdések

1. Mit takar az Apache Spark RDD fogalma? Mik a főbb jellemzői!
2. Milyen műveletek végezhetők az RDD-ken? Ezek hogyan módosítják az RDD-ket?
3. Magas szinten hogyan néz ki egy Spark program? Milyen elemei vannak?
4. Mire szolgálnak Spark-ban a megosztott változók (shared variables)? Milyen típusai vannak?
5. Milyen nyelveken írhatunk Apache Spark programot?
6. Mit jelent az, hogy a Spark lazy kiértékelési stratégiát használ az RDD-k transzformációja során?
7. Mit jelent egy RDD esetén a partíciók száma?
8. Mi a hasonlóság és különbség a `map` és a `flatMap` transzformációk között?
9. Milyen esetben célszerű a `reduceByKey`-t vagy az `aggregateByKey`-t használni a `groupByKey` helyett?
10. Melyik akció adja vissza az RDD első `n` elemét egy tömbben?
11. Milyen tárolási szinteket (storage level) ismersz az Apache Spark-on belül? Mik az egyes tárolási szintek főbb jellemzői?
12. Mi az a Dataset, milyen nyelven támogatott a Dataset API?
13. Milyen viszonyban áll egymással a Dataset és a DataFrame? Hogyan kapcsolódnak ezek az RDD-khez?

Referenciák

[1] https://spark.apache.org/

[2] https://spark.apache.org/docs/latest/api/java/index.html?org/apache/spark/api/java/JavaRDD.html

[3] https://spark.apache.org/docs/latest/rdd-programming-guide.html#understanding-closures-a-nameclosureslinka

[4] https://databricks.com/blog/2016/07/14/a-tale-of-three-apache-spark-apis-rdds-dataframes-and-datasets.html

[5] https://spark.apache.org/docs/latest/sql-programming-guide.html