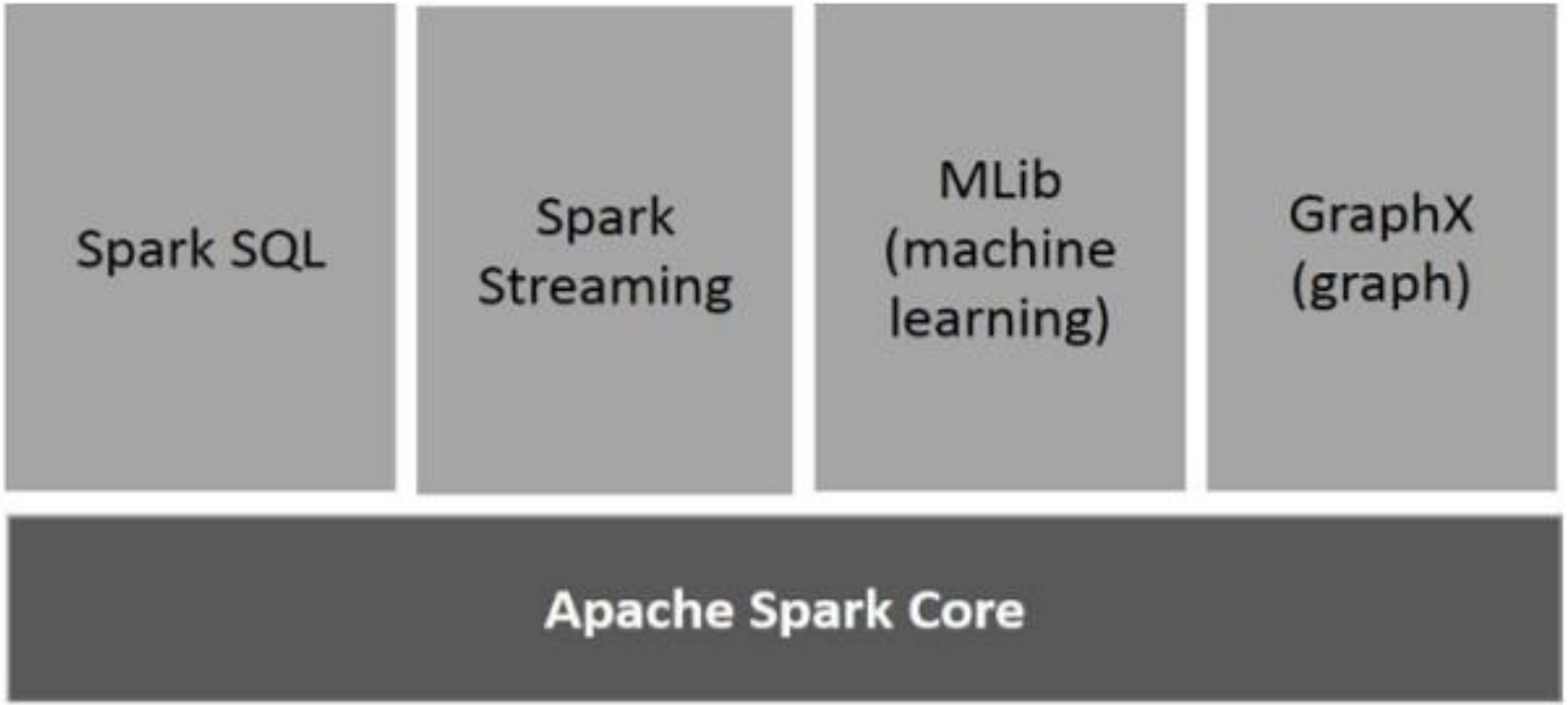
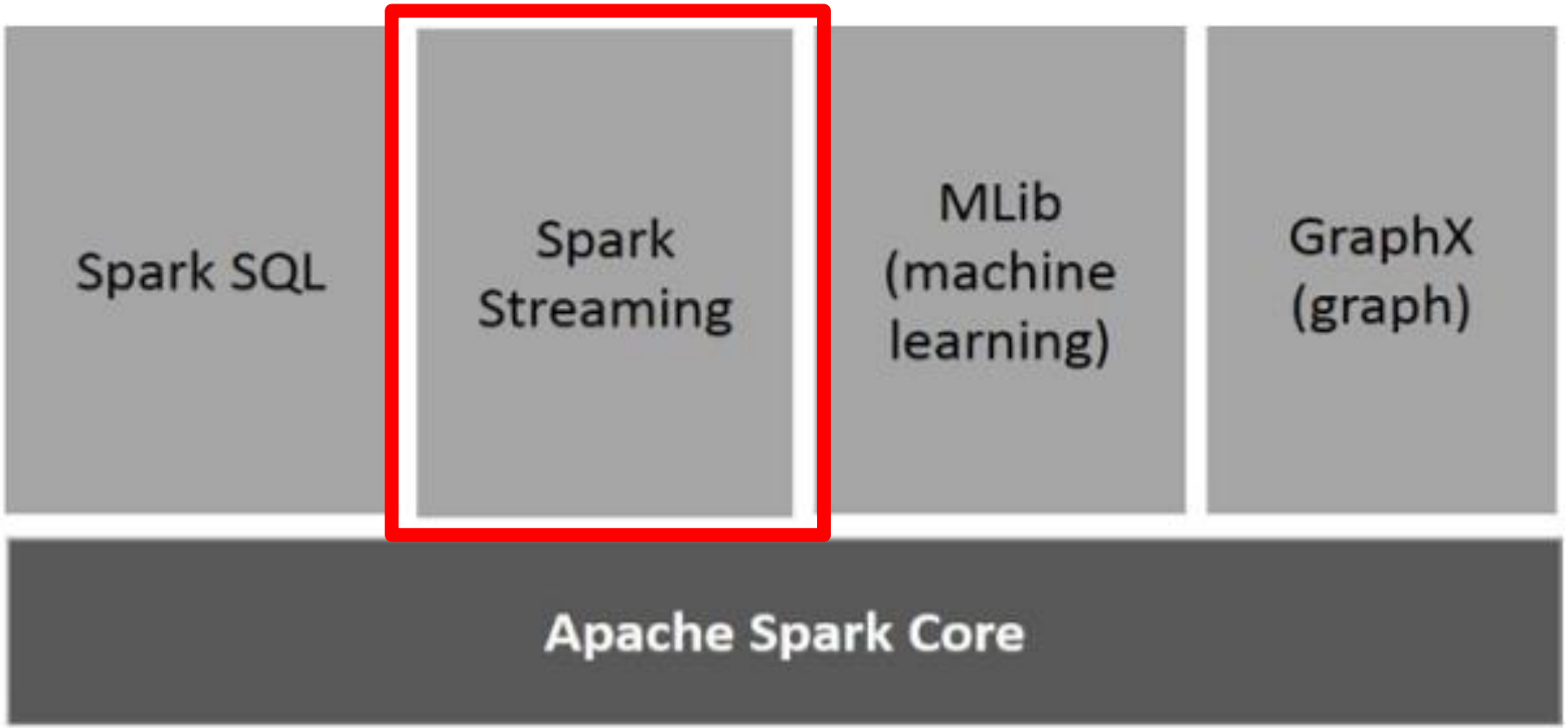
Distributed Computing, Spark Streaming, GraphX, MLib

Gombos Gergő

Spark eszközök



Spark eszközök



Spark Streaming

statikus adat

sok adat másodpercenként

Spark Streaming

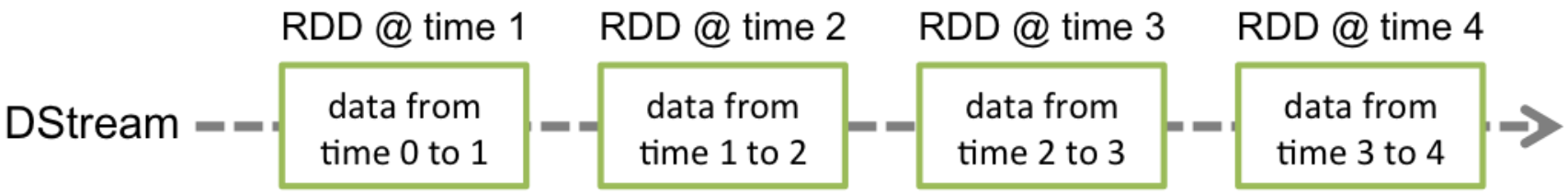
Alkalmazás	Szenzor	Web	Mobil
Behatolás detektálás	Hiba detektálás	Oldal elemzés	Hálózati mérések elemzése
Csalás detektálás	Mérési adat feldolgozás	Ajánlások	Hely alapú reklámok
Log elemzés	….	Érzelem elemzés	….

Spark Streaming

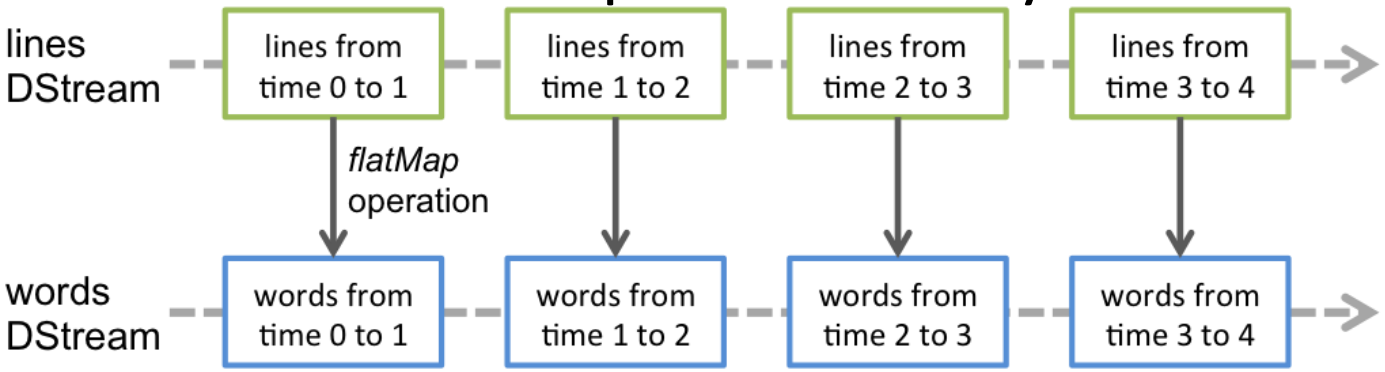
- Bementi stream darabolása
- Batch folyamat futtatása a darabokon
  - lehet: graphX, sparkML
- Kimenet tárolása

Discretize Stream (DStreams)

- DStream: RDD-k folytonos sorozata



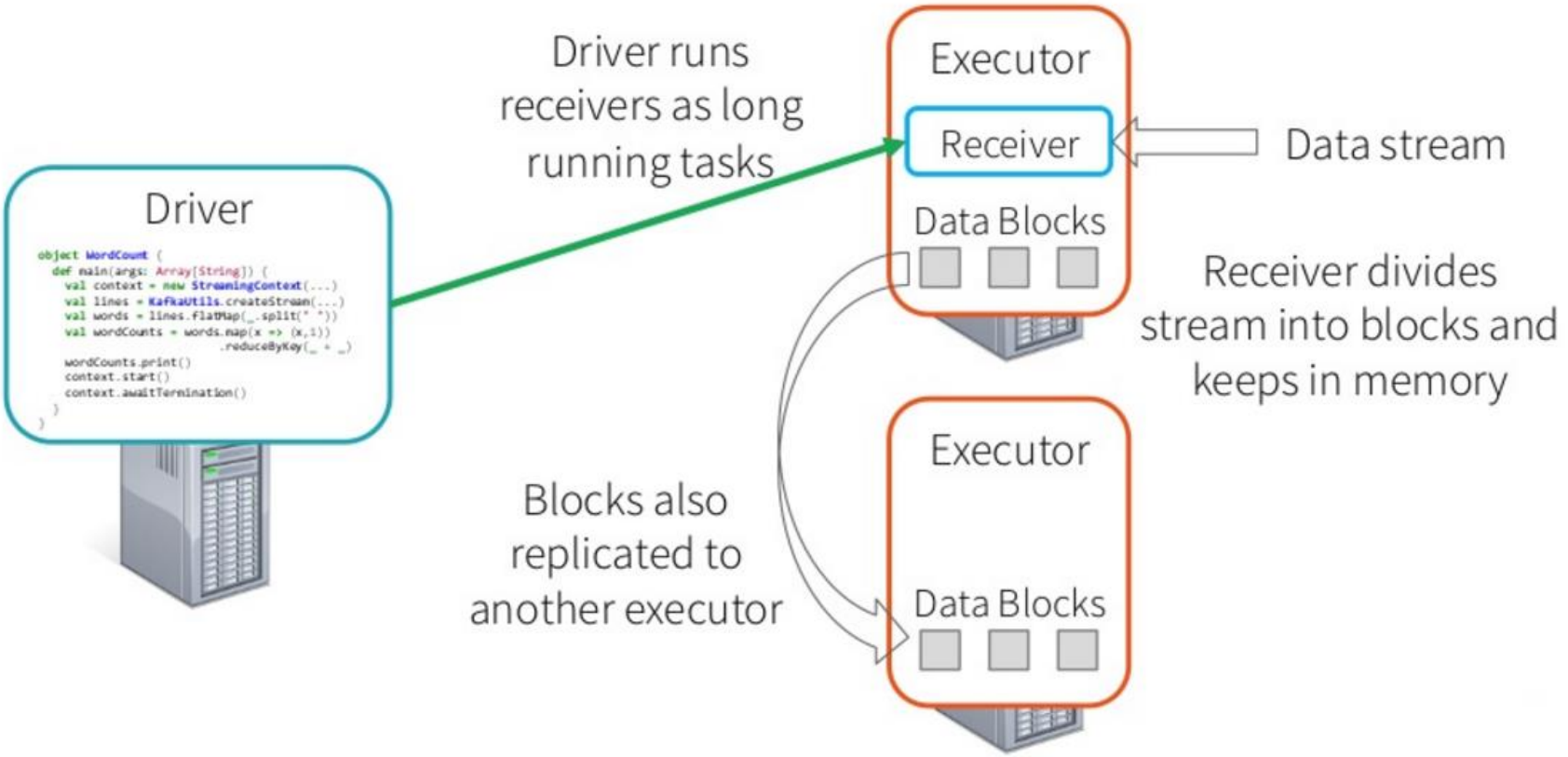
- Minden DStream-en operátor érvényes az RDD-n



Input Dstreams

- Két típusú bemenet:
  - alap források: File system, Socket connection,…
  - fejlettebb források: Kafka, Flume, Kinesis, Twitter,…

Receiver



http://www.slideshare.net/Typesafe_Inc/four-things-to-know-about-reliable-spark-streaming-with-typesafe-and-databricks

Fontos!

- Minden receiver egy magot fog használni
- Ne indítsuk „local”-lal a streaminget!
- megfelelő paraméterezés:
  - Több magot foglaljunk le a streaming-nek, mint amennyi receiver fog futni
  - Ha kevesebb akkor nem lesz mag ami feldolgozza az adatot
- Lokális futtatás pl.: „local[2]”
  - egy mag a receivernek, egy mag a processingnek

Dstream függvények

map	**flatMap**
filter	**repartition**
union	**count**
reduce	**countByValue**
reduceByKey	**join**
cogroup	**transform**

UpdateStateByKey

- Állapot változók használata
- lépések:
  1. definiálni egy állapotváltozót
  2. definiálni egy update függvényt

```
def updateFunction(newValues: Seq[Int], runningCount: Option[Int]): Option[Int] = {
   val newCount = ...
   Some(newCount)
}

val runningCounts = pairs.updateStateByKey[Int](updateFunction _)
```

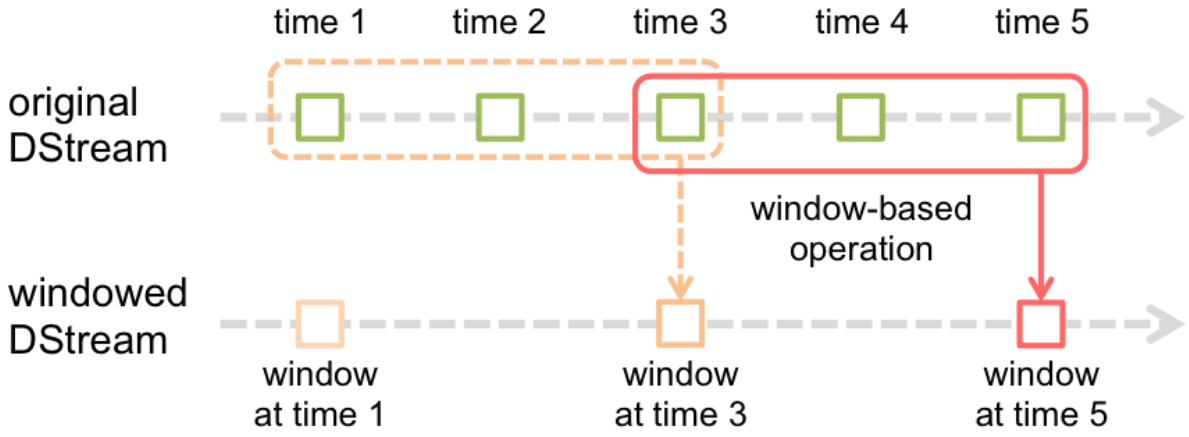
Transform

- RDD -> RDD függvény
- Itt lehet használni minden RDD függvényt
- pl:
  - Nem a streamben kiszámolt értékek hozzá vétele az adatokhoz

```
val cleanedDStream = wordCounts.transform(rdd => {
        rdd.join(spamInfoRDD).filter(...)
}
```

Window operátorok

- Lehetőség csúszó ablakos számításokra.
- Paraméterek:
  - ablak hossz (pl.: 3)
  - csúszási intervallum (pl: 2)



Window operátorok

- példa: Az utolsó 30 másodpercben akarjuk a szavak számát megszámolni minden 10 másodpercben.

```
val windowedWordCounts = pairs.reduceByKeyAndWindow(
            (a:Int,b:Int) => (a + b), Seconds(30), Seconds(10)
)
```

Window operátorok

countByWindow(*windowLength*,*slideInterval*)
reduceByWindow(*func*, *windowLength*,*slideInterval*)
reduceByKeyAndWindow(*func*,*windowLength*, *slideInterval*, [*numTasks*])
reduceByKeyAndWindow(*func*, *invFunc*,*windowLength*, *slideInterval*, [*numTasks*])
countByValueAndWindow(*windowLength*,*slideInterval*, [*numTasks*])

- A reduceByKeyAndWindow hatékonyabb használata az invFunc paraméterrel rendelkező, mert az a korábbi reduceByKey eredményeket is felhasználja. DE: a reduce függvénynek invertálhatónak kell lennie és a checkpoint lehetőség be kell hogy legyen állítva.

Join operátorok

- Adatok összekapcsolása
- Stream-stream join

```
val stream1: DStream[String, String] = ...
val stream2: DStream[String, String] = ...
val joinedStream = stream1.join(stream2)
```

```
val windowedStream1 = stream1.window(Seconds(20))
val windowedStream2 = stream2.window(Minutes(1))
val joinedStream = windowedStream1.join(windowedStream2)
```

- További lehetőségek: leftOuterJoin, rightOuterJoin, fullOuterJoin

Join operátorok

- Adatok összekapcsolása
- Stream-dataset join

```
val dataset: RDD[String, String] = ...
val windowedStream = stream.window(Seconds(20))...
val joinedStream = windowedStream.transform { rdd => rdd.join(dataset) }
```

Output operátorok

- Eredmények kiírása

print()
saveAsTextFiles(*prefix*, [*suffix*])
saveAsObjectFiles(*prefix*, [*suffix*])
saveAsHadoopFiles(*prefix*, [*suffix*])
foreachRDD(*func*)

- foreachRDD: adat kimentése külső rendszerekben

```
dstream.foreachRDD { rdd =>
  val connection = createNewConnection()  // executed at the driver
  rdd.foreach { record =>
    connection.send(record) // executed at the worker
  }
}
```

Broadcast variable

- Csak olvasható változó, amelyet minden node csak egyszer kap meg

```
@volatile private var instance: Broadcast[Seq[String]] = null

def getInstance(sc: SparkContext): Broadcast[Seq[String]] = {
      if (instance == null) {
        synchronized {
          if (instance == null) {
            val wordBlacklist = Seq("a", "b", "c")
            instance = sc.broadcast(wordBlacklist)
          }
        }
      }
      instance
}
```

Accumulator variable

- Csak növelhető változó
- Hasznos a folyamat állapotszámlálónak

```
@volatile private var instance: Accumulator[Long] = null

def getInstance(sc: SparkContext): Accumulator[Long] = {
    if (instance == null) {
        synchronized {
            if (instance == null) {
                instance = sc.accumulator(0L, "WordsInBlacklistCounter")
            }
        }
    }
    instance
}
```

Használatuk WordCount-nál

```
wordCounts.foreachRDD((rdd: RDD[(String, Int)], time: Time) => {
        // Get or register the blacklist Broadcast
    val blacklist = WordBlacklist.getInstance(rdd.sparkContext)
        // Get or register the droppedWordsCounter Accumulator
    val droppedWordsCounter = DroppedWordsCounter.getInstance(rdd.sparkContext)
        // Use blacklist to drop words and use droppedWordsCounter to count them

    val counts = rdd.filter { case (word, count) =>
        if (blacklist.value.contains(word)) {
            droppedWordsCounter += count
            false
        } else {
            true
        }
     }.collect()
     val output = "Counts at time " + time + " " + counts
})
```

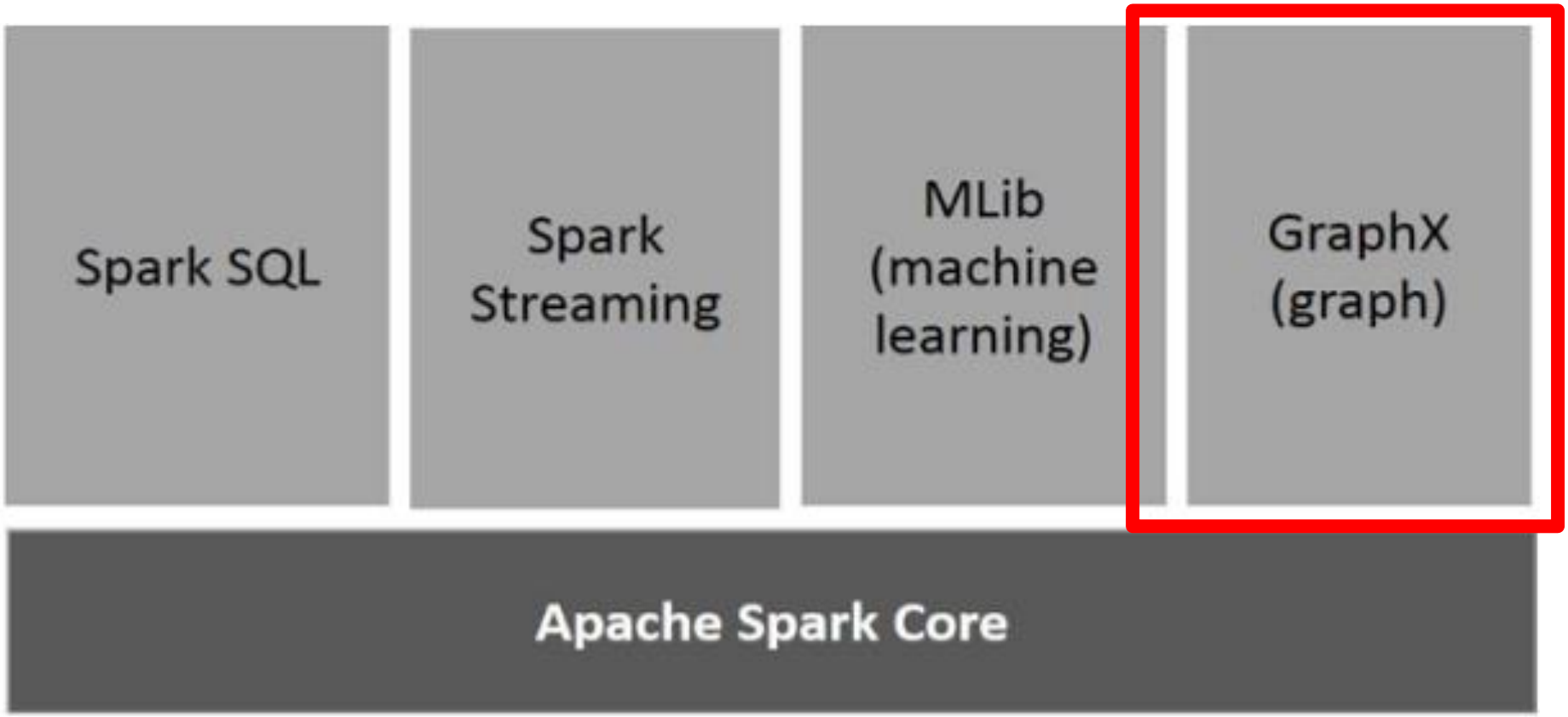
Checkpointing

- Az adatok folyamatosak, így hiba esetén nincs visszaállítási lehetőség
- A spark-nak checkpointokra van szüksége, hogy visszatudjon állítani egy korábbi állapotot
- Típusai:
  - Metadata szintű: a stream számításhoz szükséges információkat tárolja (driver hibákra jó)
  - Adat szintű: korábbi RDD adatok tárolása (statefull adat hibákra jó)
- Checkpoint beállítása:
  - egy fault-tolerant rendszeren mappa megadása (pl HDFS)
  - ssc.checkpoint(checkpointDirectory)

Alkalmazás felépítése

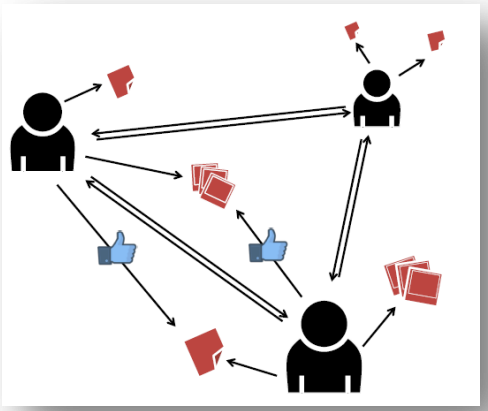
1. SparkConf, StreamingContext
2. Input source megadása
3. Stream műveletek elkészítése
4. streamContext elindítása
5. Végtelen futtatás vagy streamcontext leállítása

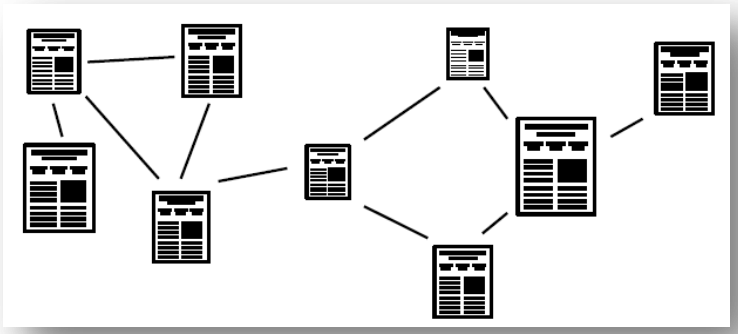
Spark eszközök

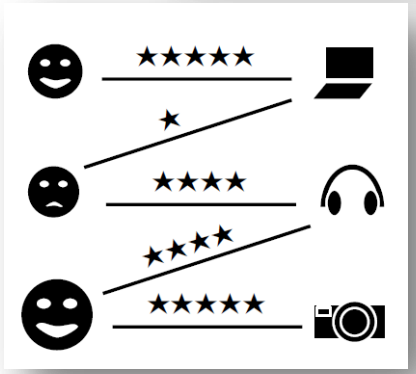


Gráfok

- Nagy méretű gráfok mindenhol
  - Web: weboldalak, hivatkozások
  - Közösségi hálózatok
  - Felhasználó-Termék gráf







Gráf feldolgozási feladatok

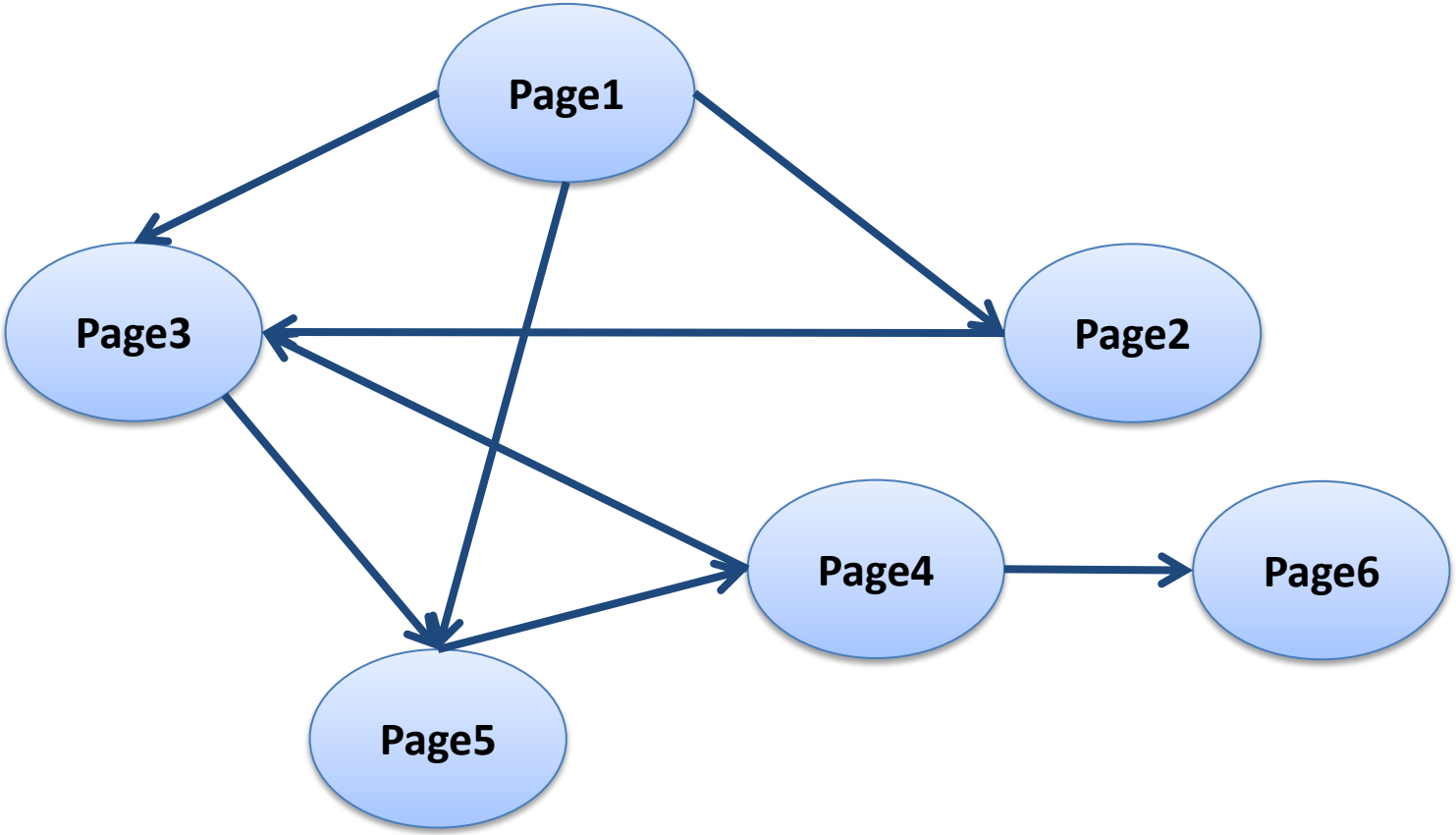
- Alap statisztikák
  - Pl: Bemenő/kimenő élek
- Származtatott tulajdonságok a gráfból
  - Klaszterezési együttható
- Legrövidebb út keresés
  - Internet (routing)
- Párosítás
  - Randi oldalak (match.com)
- …

Gráf feldolgozási feladatok

- PageRank
  - Node fontosság meghatározás
- Háromszögek keresése
  - Társaságok megtalálása

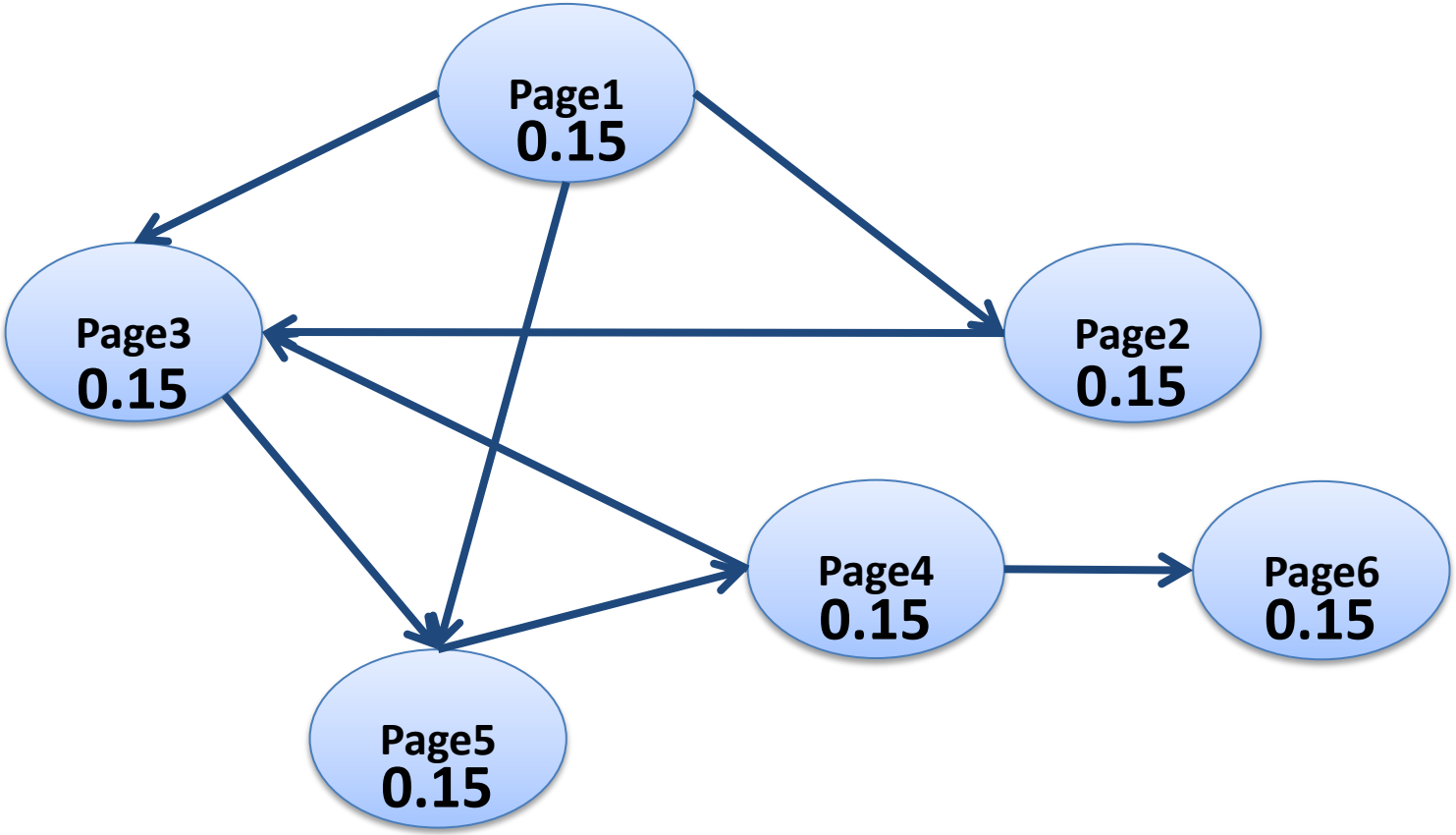
PageRank példa

ResetProb: 0.15



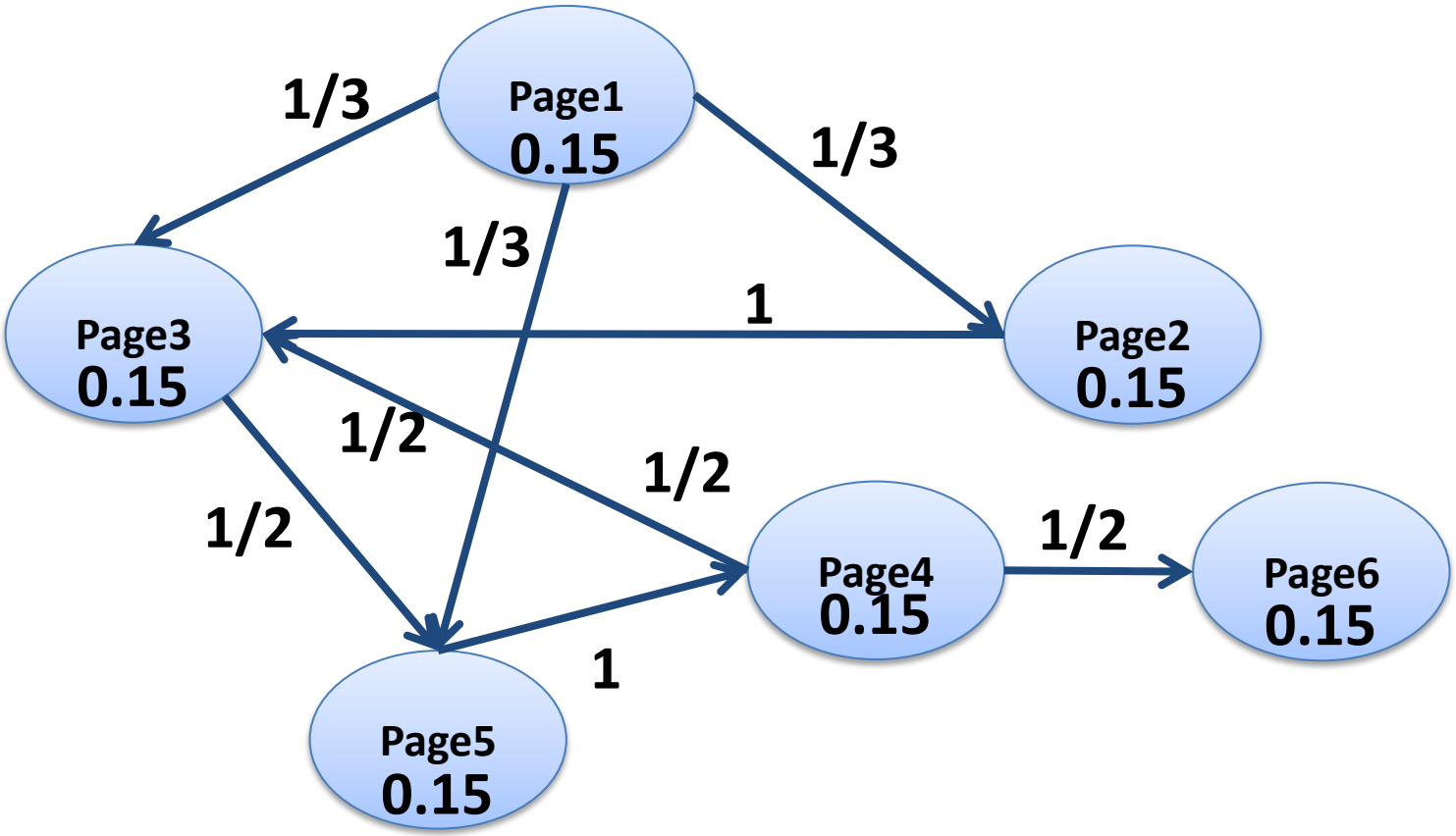
PageRank példa

ResetProb: 0.15



PageRank példa

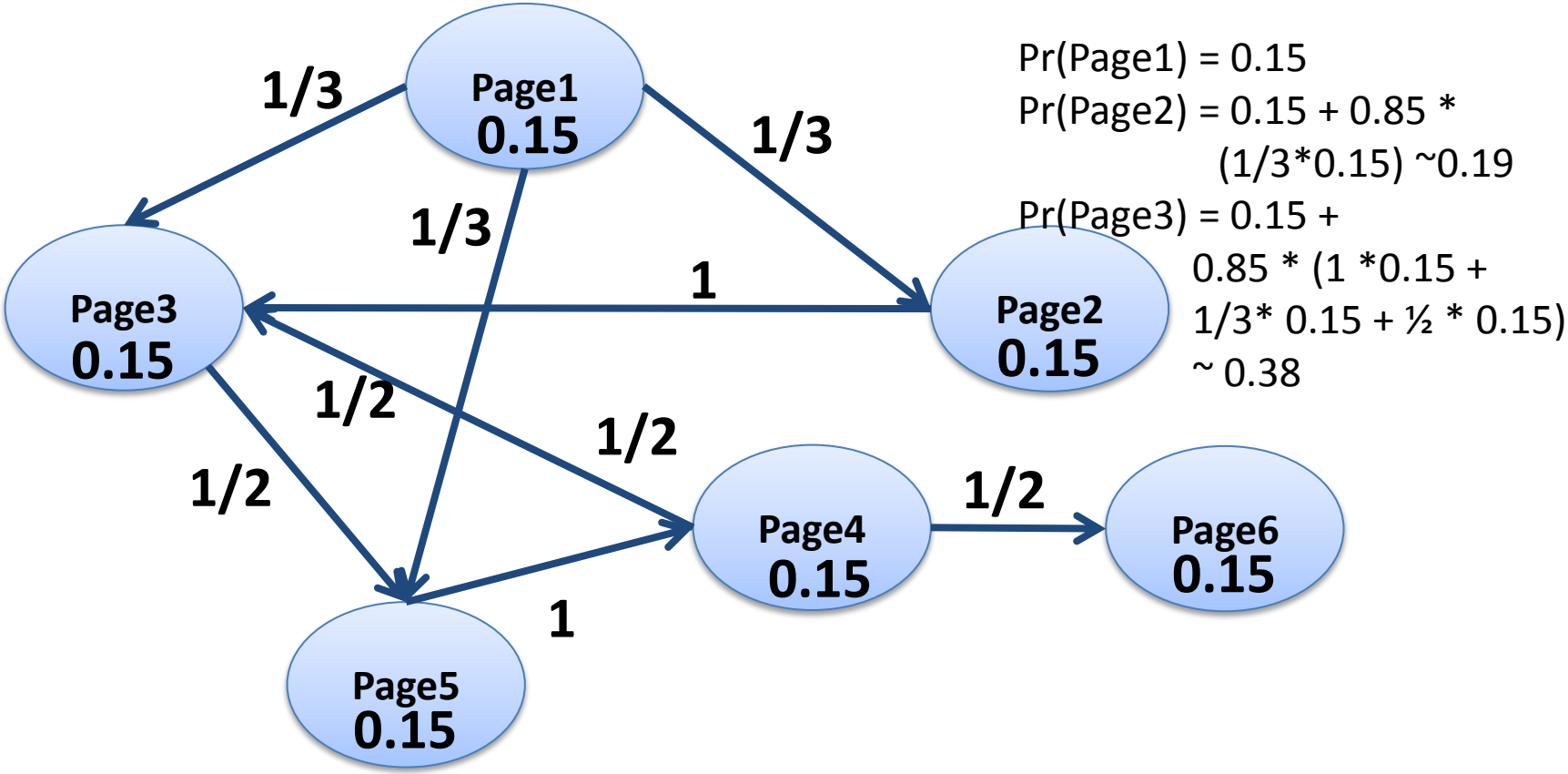
ResetProb: 0.15



PageRank példa

ResetProb: 0.15

Pr(Page) = resetProb + (1-resetProb) * szumAll(NeighPage * Edge)



Pr(Page1) = 0.15

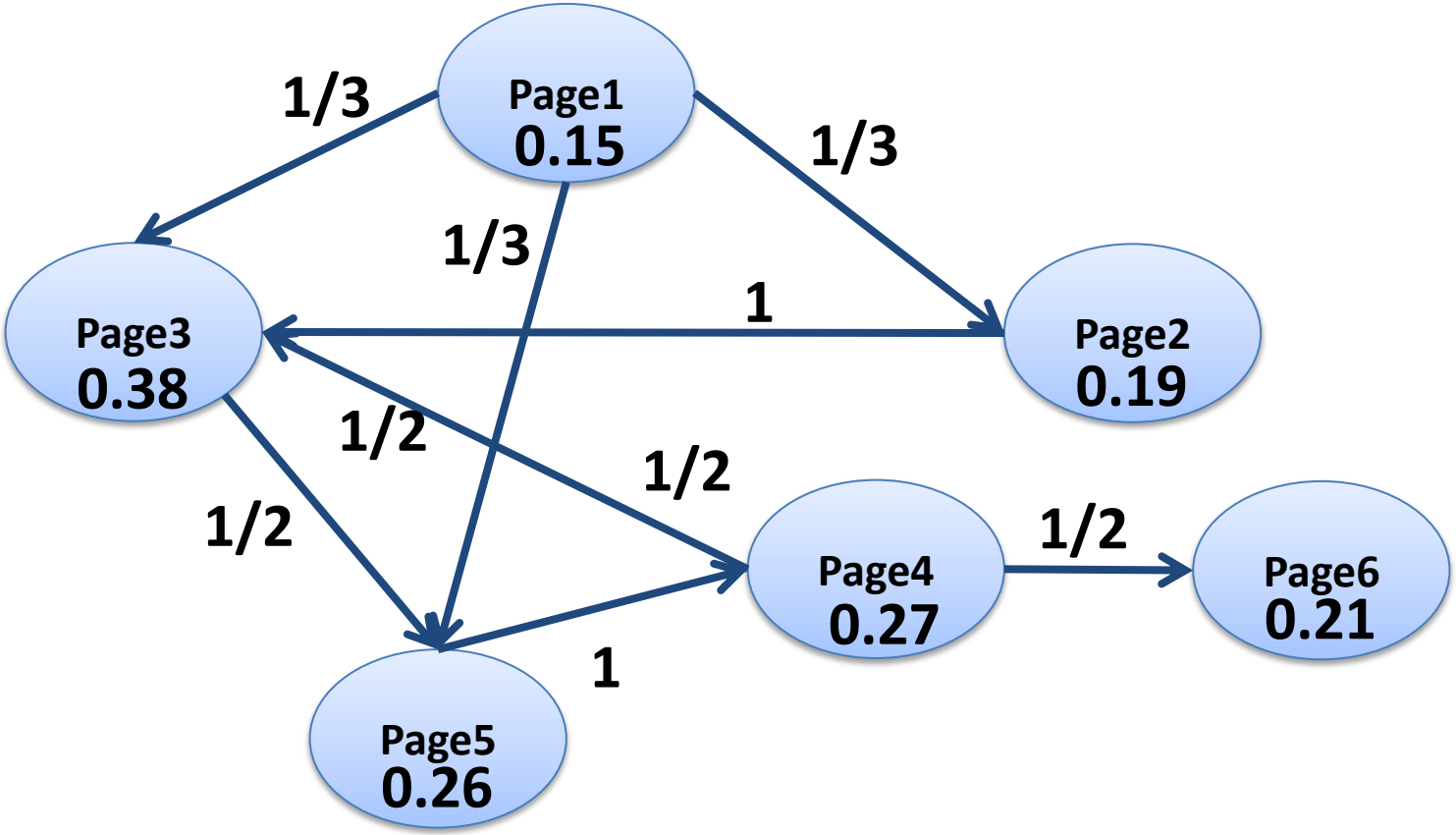
Pr(Page2) = 0.15 + 0.85 * (1/3*0.15) ~0.19

Pr(Page3) = 0.15 + 0.85 * (1 *0.15 + 1/3* 0.15 + ½ * 0.15) ~ 0.38

PageRank példa

ResetProb: 0.15

Pr(Page) = resetProb + (1-resetProb) * szumAll(NeighPage * Edge)



Kihívások

- Gráfok nagyok
  - Milliós nagyságrendű csúcsok és élek
- Tárolás, feldolgozás egy serveren:
  - Memória limit
  - Lassú
- Tárolás elosztva:
  - Sok kommunikáció a gépek között

Tipikus gráf feldolgozás

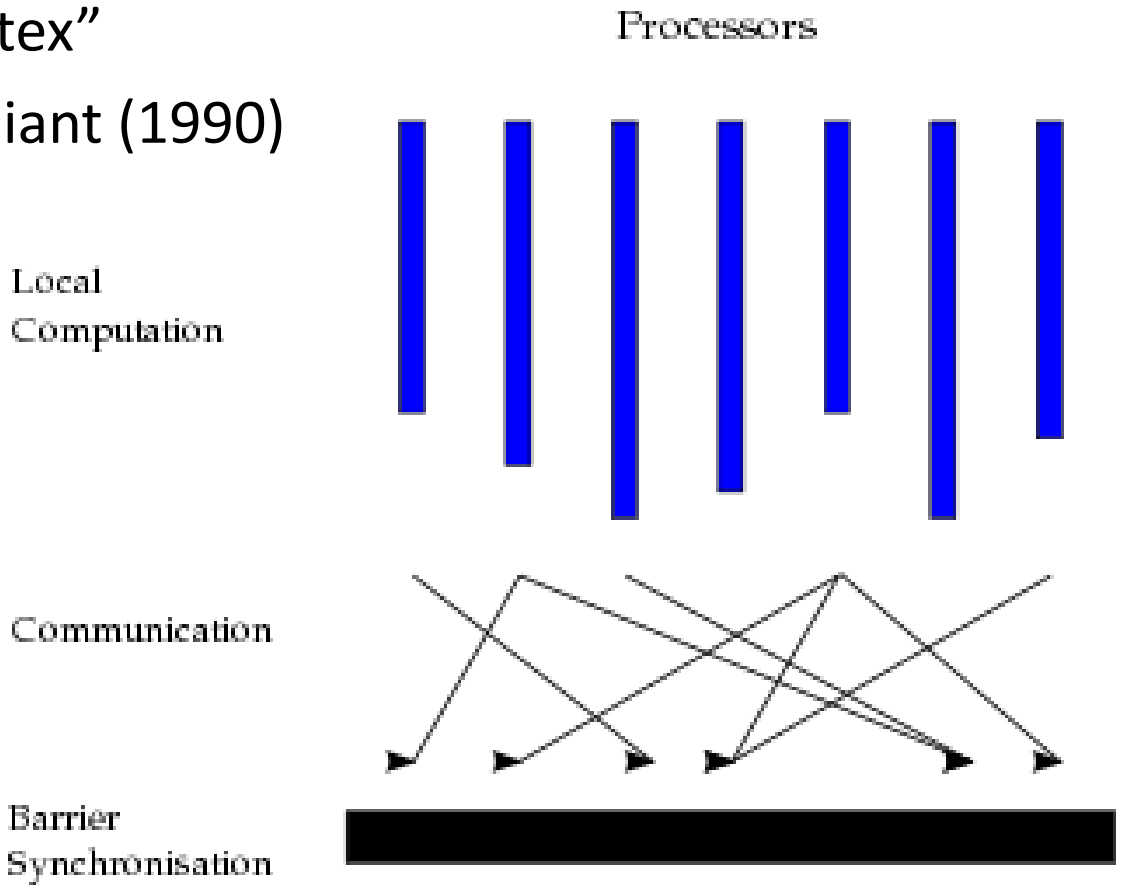
- Iteratív
- Minden csúcs rendelkezik értékkel
- Minden iterációban:
  - Újraszámolja a saját értékét a szomszédok és az élek alapján
- Iteráció vége:
  - Fix számú iteráció
  - Nincs új érték a gráfban

MapReduce?

- 1 job = 1 iteráció
- A reduce kulcsa nodeid, értéke az üzenetek
- Előny:
  - Ismert
- Hátrány:
  - Minden csúcs átmegy a hálózaton minden iterációban
  - Minden csúcs ki lesz írva a HDFS-re
  - Lassú, nem hatékony

Bulk Synchronous Parallel(BSP) Model

- „Think like a vertex”
- Originally by Valiant (1990)



Elosztott gráf feldolgozás

- „Think like a vertex”
- Minden csúcs legyen egy szerveren
- Így a szerverek csúcshalmazt tárolnak
- Egy iteráció lépése:
  - Gather: megkapjuk a szomszéd csúcsok értékét
  - Apply: Kiszámoljuk az új értéket
  - Scatter: elküldjük az új értéket a szomszédoknak

Csúcs tárolás

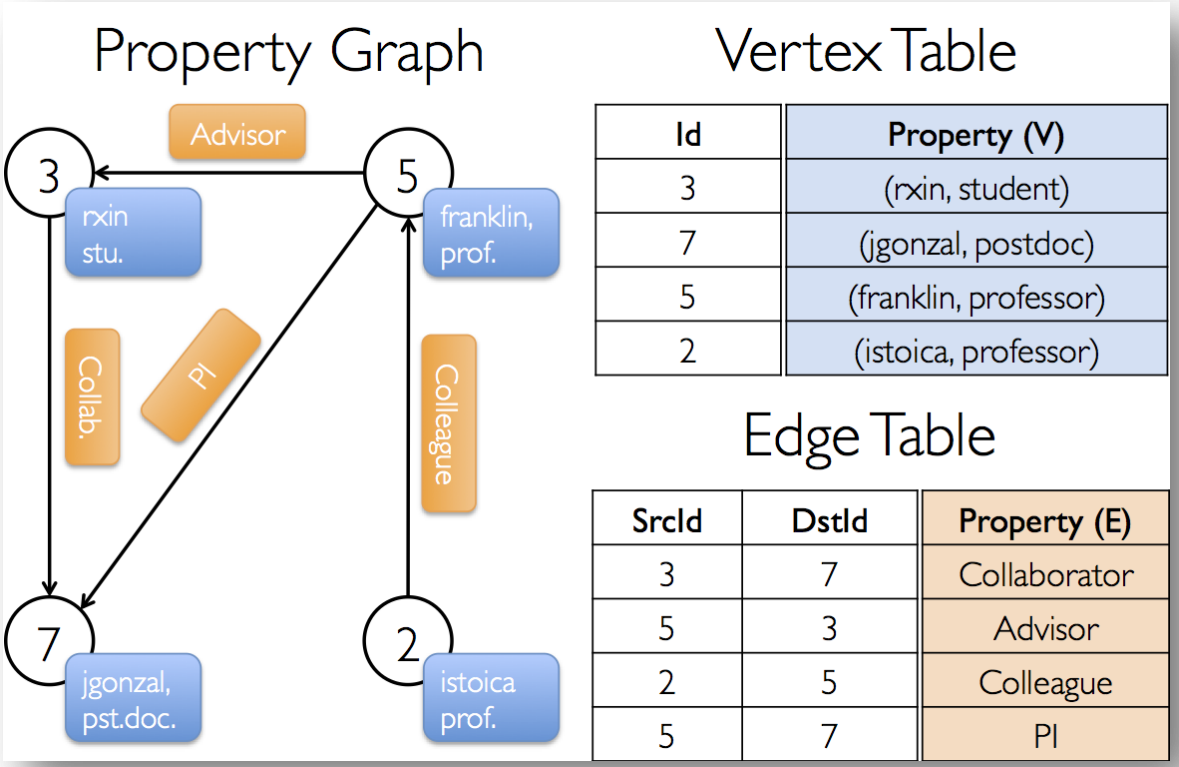
- Honnan tudjuk melyik csúcs melyik szerveren tárolódik?
- Lehetőségek:
  - Hash-alapú:
    - Hash(csúcs id) mod num(server)
    - Hasonló: P2P
  - Lokalitás-alapú
    - A szomszédos csúcsokat próbáljuk egy szerveren tárolni
    - Csökken a szerver-szerver közötti kommunikáció

GraphX

- Optimális particionálás és indexelés a csúcsok és élek tárolására
- Alapstruktúra:
  - Property graph
- API
  - Bejárás
  - Elérés
- Alap függvények
  - PageRank
  - Connected components
  - Tringle counting

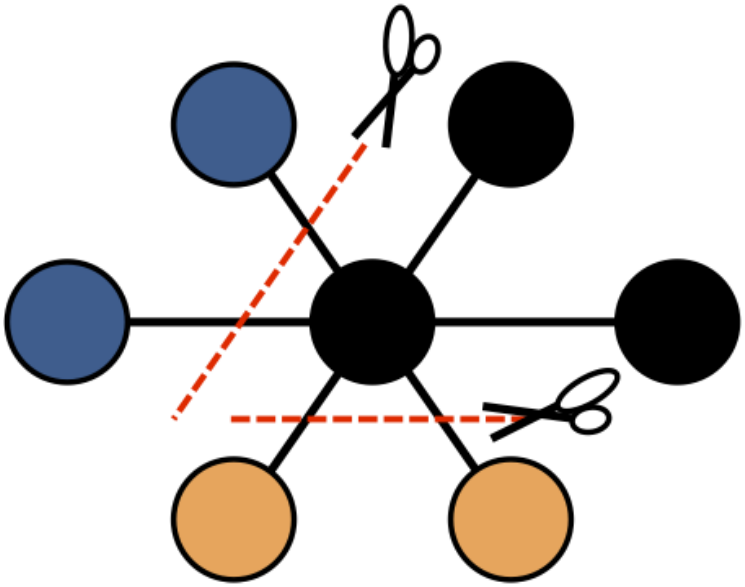
Property graph

- Felhasználói struktúrák a csúcsokhoz és élekhez



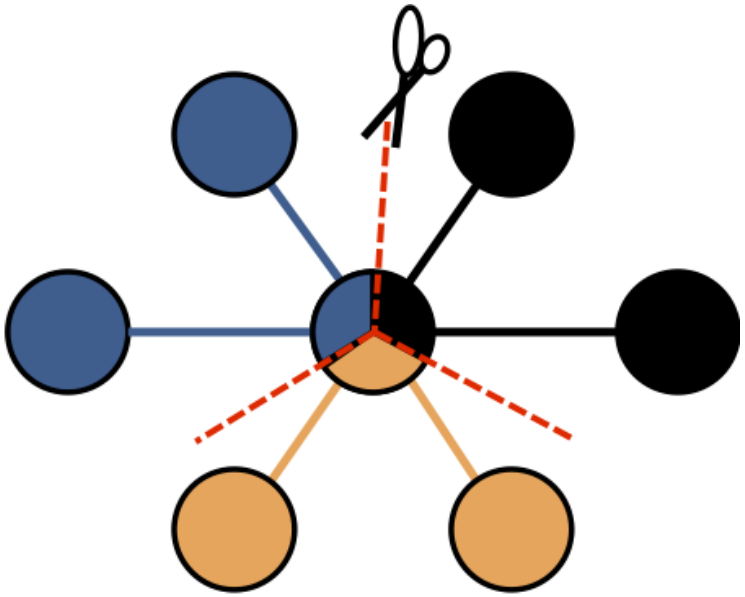
Optimális gráftárolás

Általában



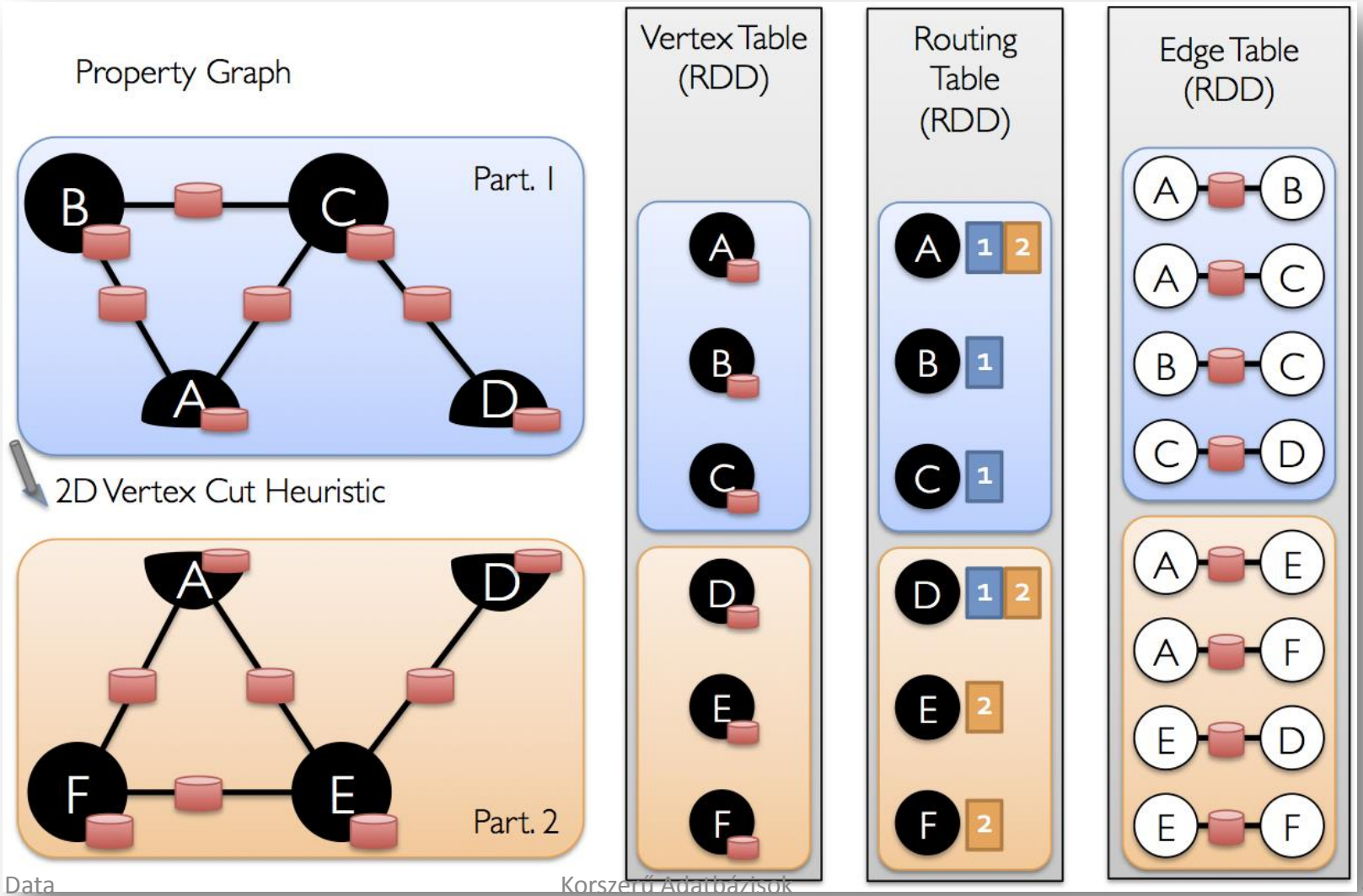
Edge Cut

GraphX



Vertex Cut

Optimális adattárolás



Gráf operátorok

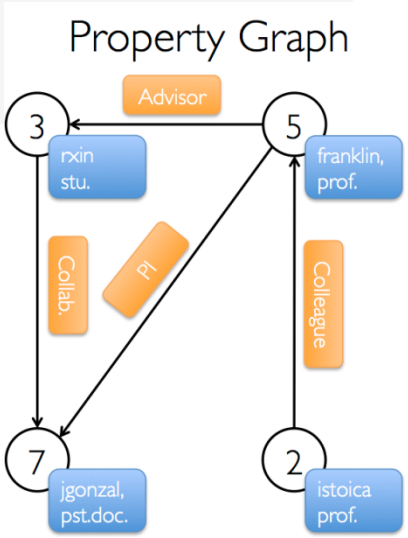
```scala
// Information about the Graph ===================================================================
val numEdges: Long
val numVertices: Long
val inDegrees: VertexRDD[Int]
val outDegrees: VertexRDD[Int]
val degrees: VertexRDD[Int]
// Views of the graph as collections =============================================================
val vertices: VertexRDD[VD]
val edges: EdgeRDD[ED]
val triplets: RDD[EdgeTriplet[VD, ED]]
// Functions for caching graphs ==================================================================
def persist(newLevel: StorageLevel = StorageLevel.MEMORY_ONLY): Graph[VD, ED]
def cache(): Graph[VD, ED]
def unpersistVertices(blocking: Boolean = true): Graph[VD, ED]
// Change the partitioning heuristic  ============================================================
def partitionBy(partitionStrategy: PartitionStrategy): Graph[VD, ED]
// Transform vertex and edge attributes ==========================================================
def mapVertices[VD2](map: (VertexID, VD) => VD2): Graph[VD2, ED]
def mapEdges[ED2](map: Edge[ED] => ED2): Graph[VD, ED2]
def mapEdges[ED2](map: (PartitionID, Iterator[Edge[ED]]) => Iterator[ED2]): Graph[VD, ED2]
def mapTriplets[ED2](map: EdgeTriplet[VD, ED] => ED2): Graph[VD, ED2]
def mapTriplets[ED2](map: (PartitionID, Iterator[EdgeTriplet[VD, ED]]) => Iterator[ED2])
  : Graph[VD, ED2]
```

Gráf operátorok

```scala
// Modify the graph structure ==========================================================
def reverse: Graph[VD, ED]
def subgraph(
    epred: EdgeTriplet[VD,ED] => Boolean = (x => true),
    vpred: (VertexID, VD) => Boolean = ((v, d) => true))
  : Graph[VD, ED]
def mask[VD2, ED2](other: Graph[VD2, ED2]): Graph[VD, ED]
def groupEdges(merge: (ED, ED) => ED): Graph[VD, ED]
// Join RDDs with the graph ============================================================
def joinVertices[U](table: RDD[(VertexID, U)])(mapFunc: (VertexID, VD, U) => VD): Graph[VD, ED]
def outerJoinVertices[U, VD2](other: RDD[(VertexID, U)])
    (mapFunc: (VertexID, VD, Option[U]) => VD2)
  : Graph[VD2, ED]
// Aggregate information about adjacent triplets =======================================
def collectNeighborIds(edgeDirection: EdgeDirection): VertexRDD[Array[VertexID]]
def collectNeighbors(edgeDirection: EdgeDirection): VertexRDD[Array[(VertexID, VD)]]
def aggregateMessages[Msg: ClassTag](
    sendMsg: EdgeContext[VD, ED, Msg] => Unit,
    mergeMsg: (Msg, Msg) => Msg,
    tripletFields: TripletFields = TripletFields.All)
  : VertexRDD[A]
```

GraphX

```scala
// Assume the SparkContext has already been constructed
val sc: SparkContext
// Create an RDD for the vertices
val users: RDD[(VertexId, (String, String))] =
  sc.parallelize(Array((3L, ("rxin", "student")), (7L, ("jgonzal", "postdoc")),
                       (5L, ("franklin", "prof")), (2L, ("istoica", "prof"))))
// Create an RDD for edges
val relationships: RDD[Edge[String]] =
  sc.parallelize(Array(Edge(3L, 7L, "collab"),    Edge(5L, 3L, "advisor"),
                       Edge(2L, 5L, "colleague"), Edge(5L, 7L, "pi")))
// Define a default user in case there are relationship with missing user
val defaultUser = ("John Doe", "Missing")
// Build the initial Graph
val graph = Graph(users, relationships, defaultUser)
```



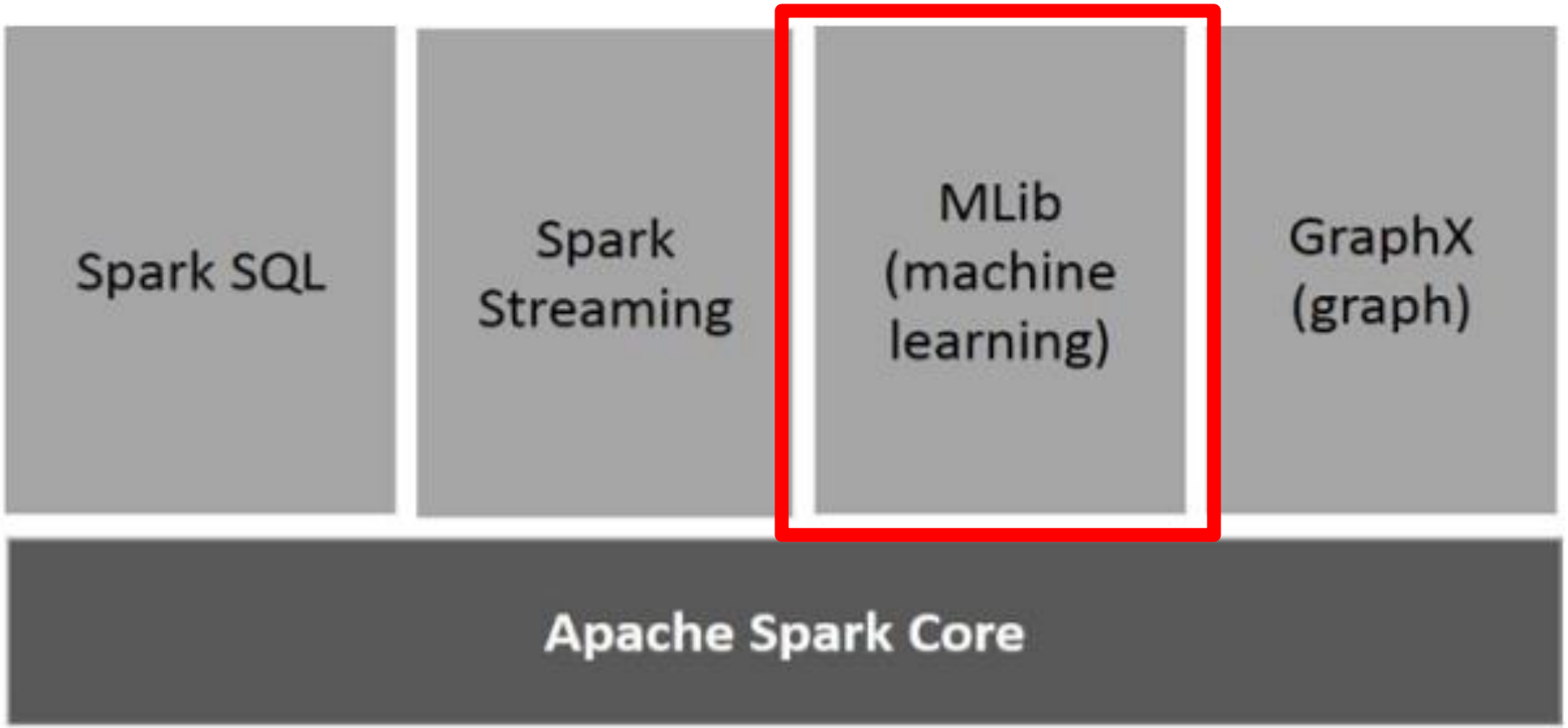
Vertex Table

Id	Property (V)
3	(rxin, student)
7	(jgonzal, postdoc)
5	(franklin, professor)
2	(istoica, professor)

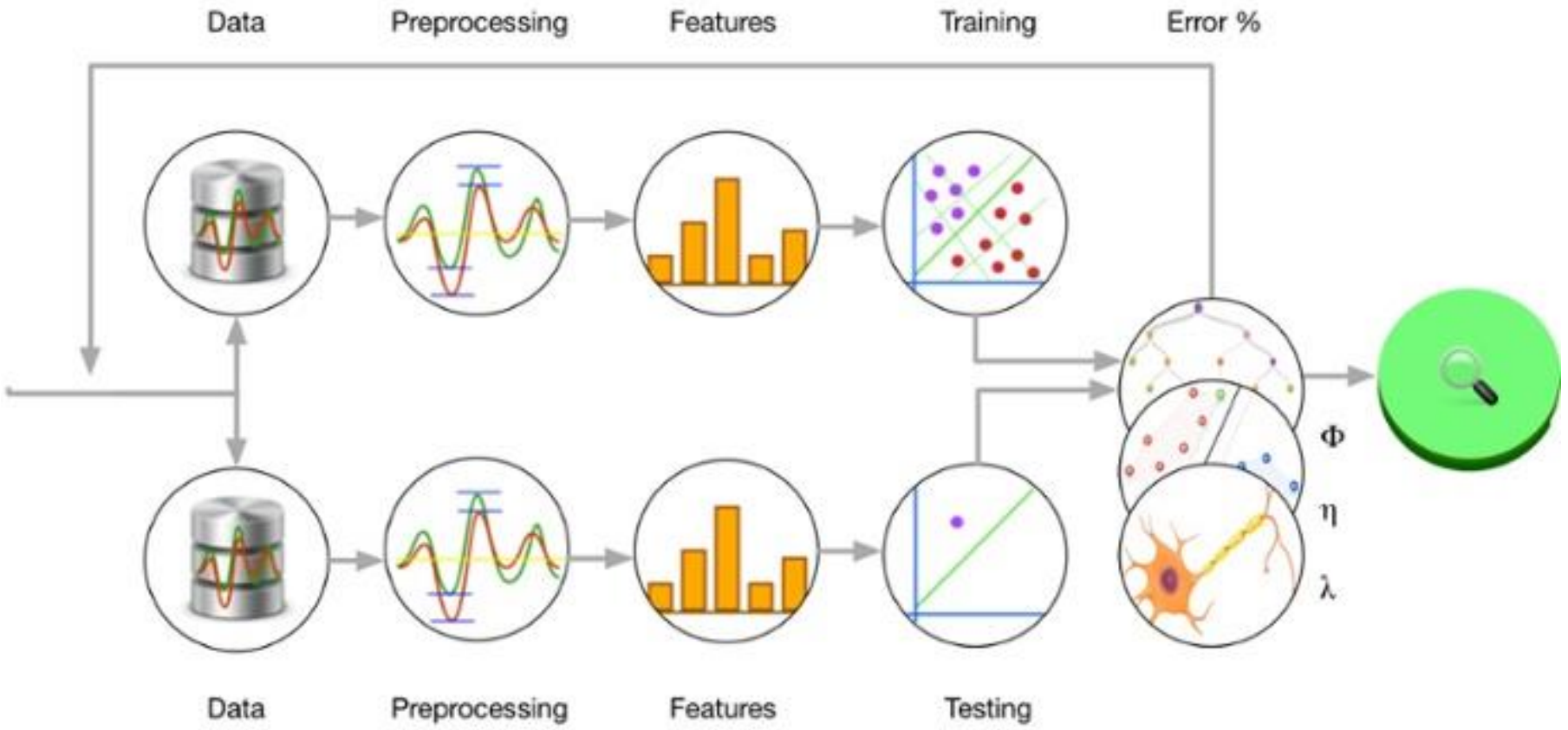
Edge Table

SrcId	DstId	Property (E)
3	7	Collaborator
5	3	Advisor
2	5	Colleague
5	7	PI

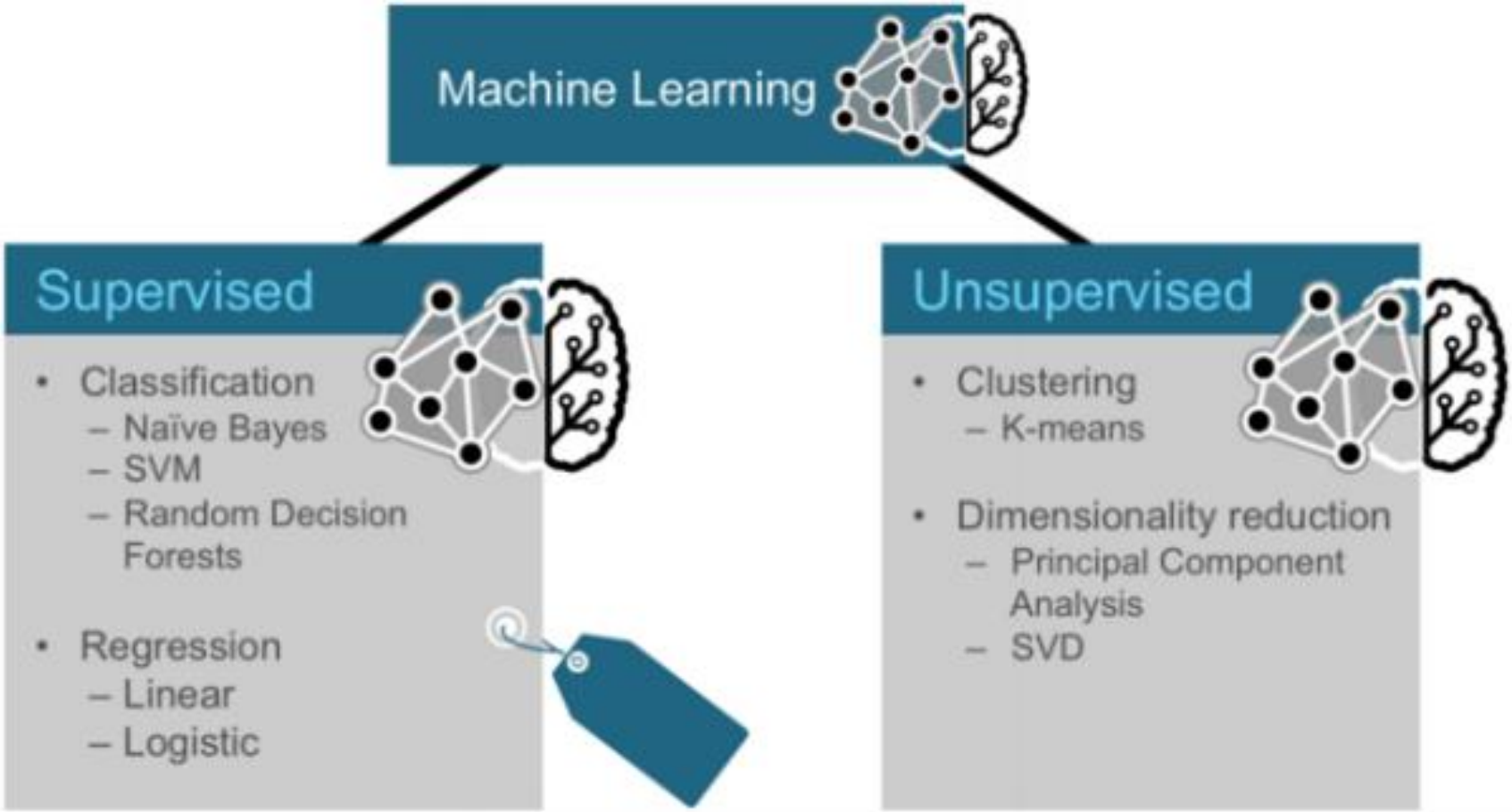
Spark eszközök



Machine Learning



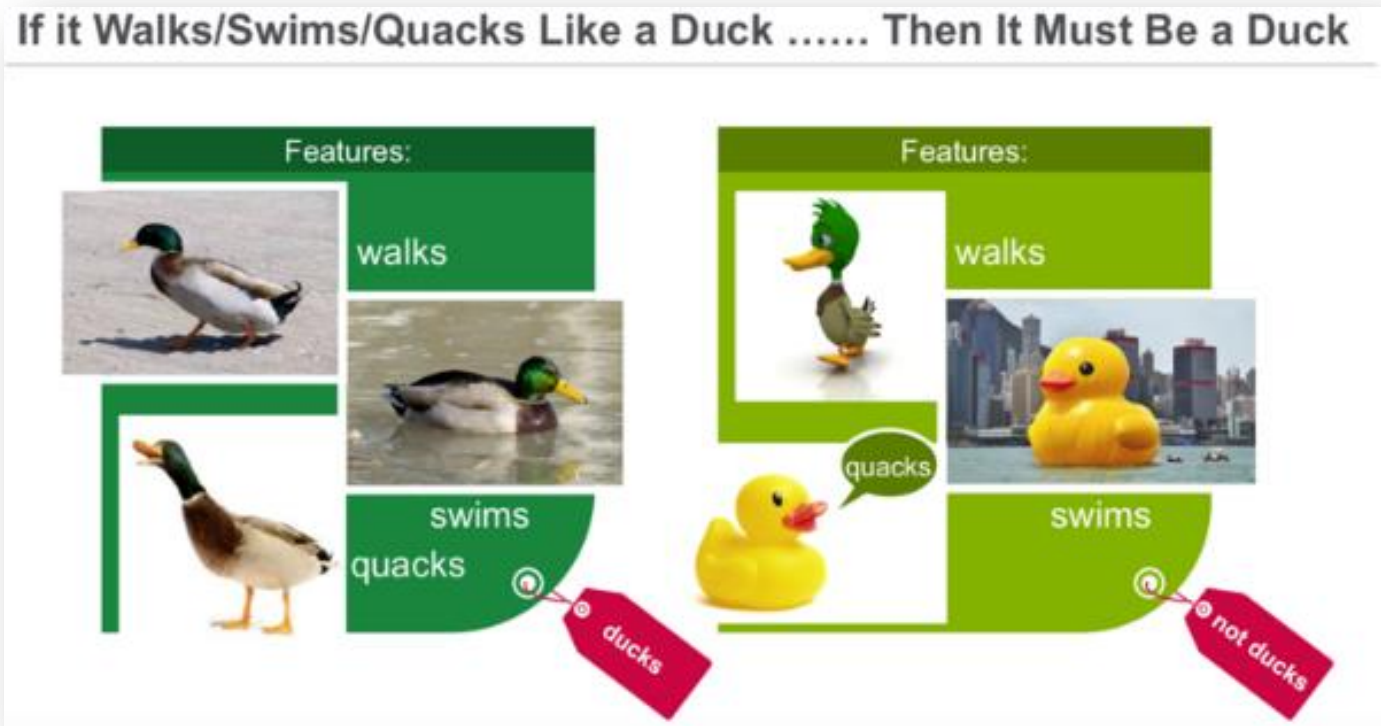
Machine Learning Library (MLlib)



Classification

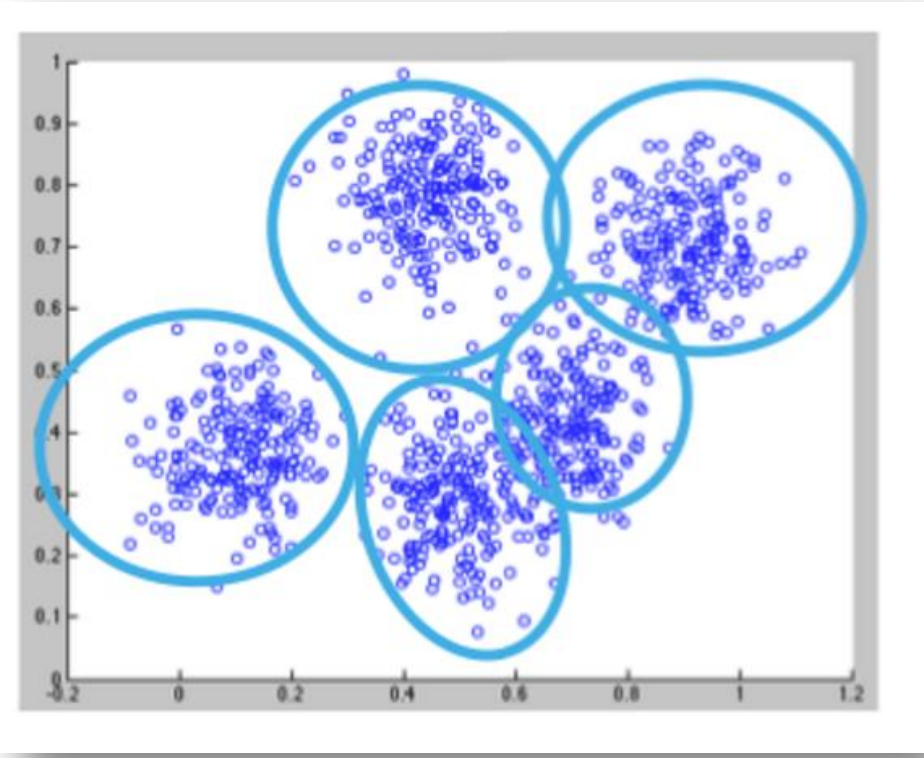
- Adatok címkézése / osztályokba sorolásaű

- pl:
  - spam



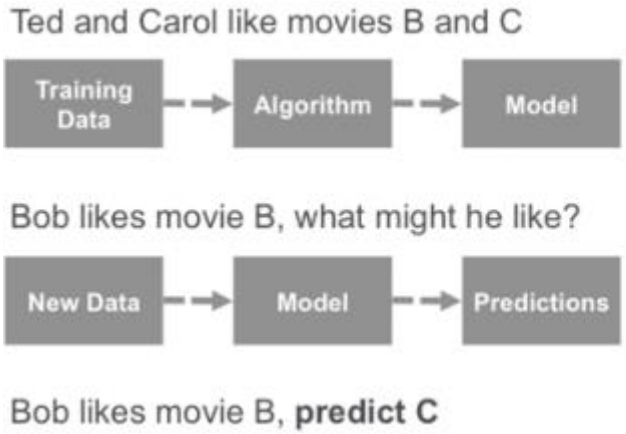
Clustering

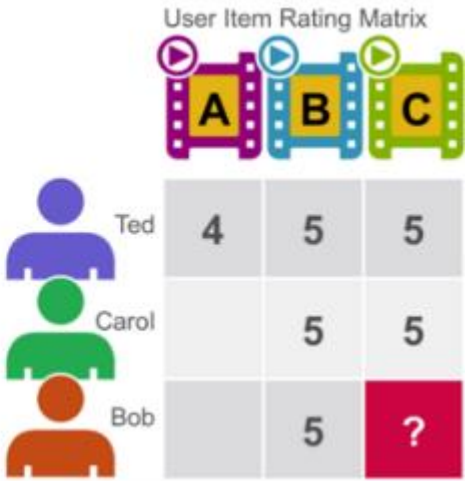
- Csoportok keresése
- pl:
  - Vevők csoportosítása
  - Anomália detektálás
  - Szöveg kategorizálása



Collaborative Filtering

- Ajánló rendszerek
  - A múltban hasonló termékeket kedvelő userek a jövőben is kedveli fognak hasonló terméket.





Machine Learning Library (MLlib)

- 2 csomagban érhetőek el:
  - spark.mllib:
    - RDD-ken dolgozik
    - alacsonyabb szintű
  - spark.ml
    - DataFrame-ken dolgozik
    - magasabbszintű
    - pipeline-ok készítése

spark.ml package

- Építő elemi:
  - Transformer:
    - DataFrame-ből DataFrame-be alakít át
  - Estimator:
    - Egy algoritmus ami rátanul az adatra (egy model-t készít)
  - Pipeline:
    - Transformerek és Estimator-ok láncolt sorozata
  - Parameter
    - Különböző paraméterek a Trandformereknek, Estimatoroknek

Transformer

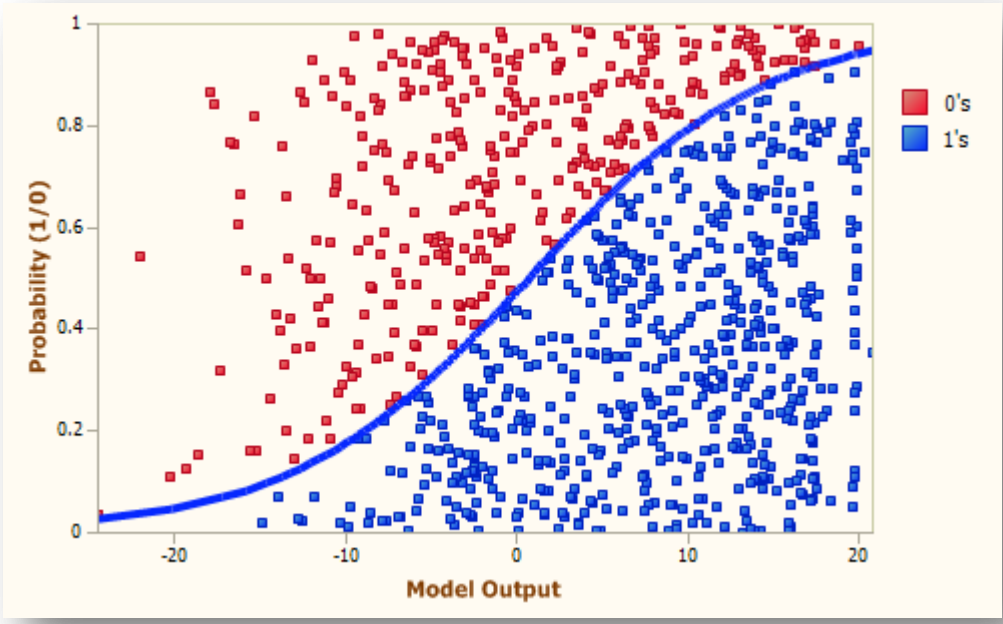
- példa:
  - olvass be egy oszlopot („text”) készíts egy új oszlopot („features vector”) a term-frequency alapján (hányszor fordult elő a szó a dokumentumban)

```
val hashingTF = new HashingTF()
    .setInputCol(„text”)
    .setOutputCol("features")
```

Estimator

- példa:
  - Logisztikus regresszió

```
val lr = new LogisticRegression()
    .setMaxIter(10)
    .setRegParam(0.01)
```



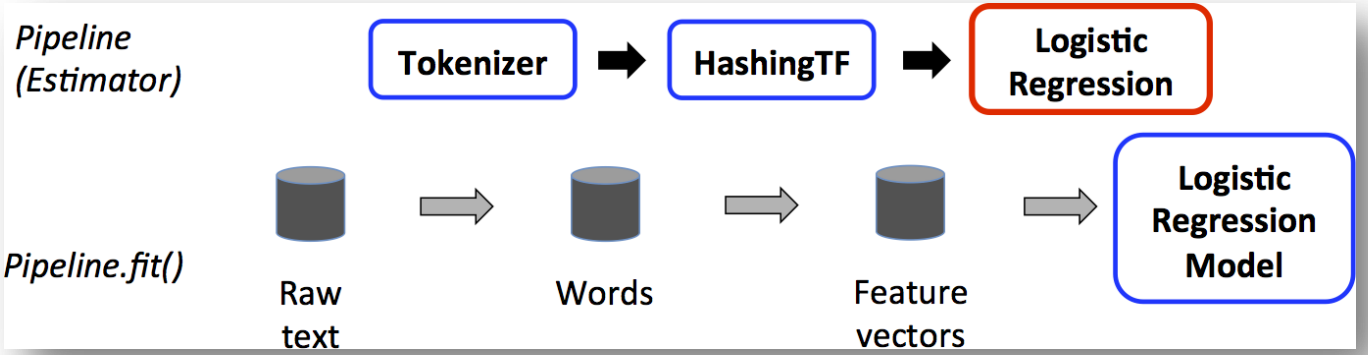
Pipeline

- Az egyes komponenseket egy sorba rakjuk
  - Alakítsd a szövegeket dokumentummá
  - alakítsd a szavakat számmal rendelkező feature vektorrá
  - Tanulj rá az adathalmazra

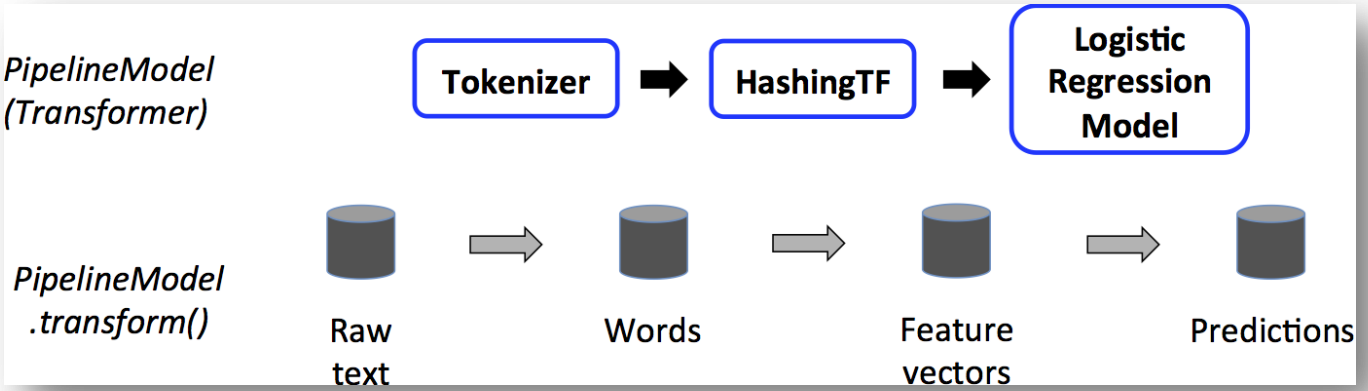
```
val pipeline = new Pipeline()
    .setStages(Array(tokenizer, hashingTF, lr))

  val model = pipeline.fit(training)
```

Pipeline

Training



Test



Parameters

- Két lehetőség paraméterek átadására
  1. Az instance-nak adjuk meg
     - logreg.setMaxIter(10)
  2. Készítünk egy ParamMap-et
     - ‚maxIter’ -> 10
     - Ezt a ParamMap-et minden instance használni fogja
     - Ez felülírja a setter-ekkel beállított értéket

Model selection via Cross-Validation

```
// We use a ParamGridBuilder to construct a grid of parameters to search over.
// With 3 values for hashingTF.numFeatures and 2 values for lr.regParam,
// this grid will have 3 x 2 = 6 parameter settings for CrossValidator to choose from.
val paramGrid = new ParamGridBuilder()
  .addGrid(hashingTF.numFeatures, Array(10, 100, 1000))
  .addGrid(lr.regParam, Array(0.1, 0.01))
  .build()
```

```
// Note that the evaluator here is a BinaryClassificationEvaluator and
// its default metric is areaUnderROC.
val cv = new CrossValidator()
  .setEstimator(pipeline)
  .setEvaluator(new BinaryClassificationEvaluator)
  .setEstimatorParamMaps(paramGrid)
  .setNumFolds(2) // Use 3+ in practice
```

Köszönöm a figyelmet!