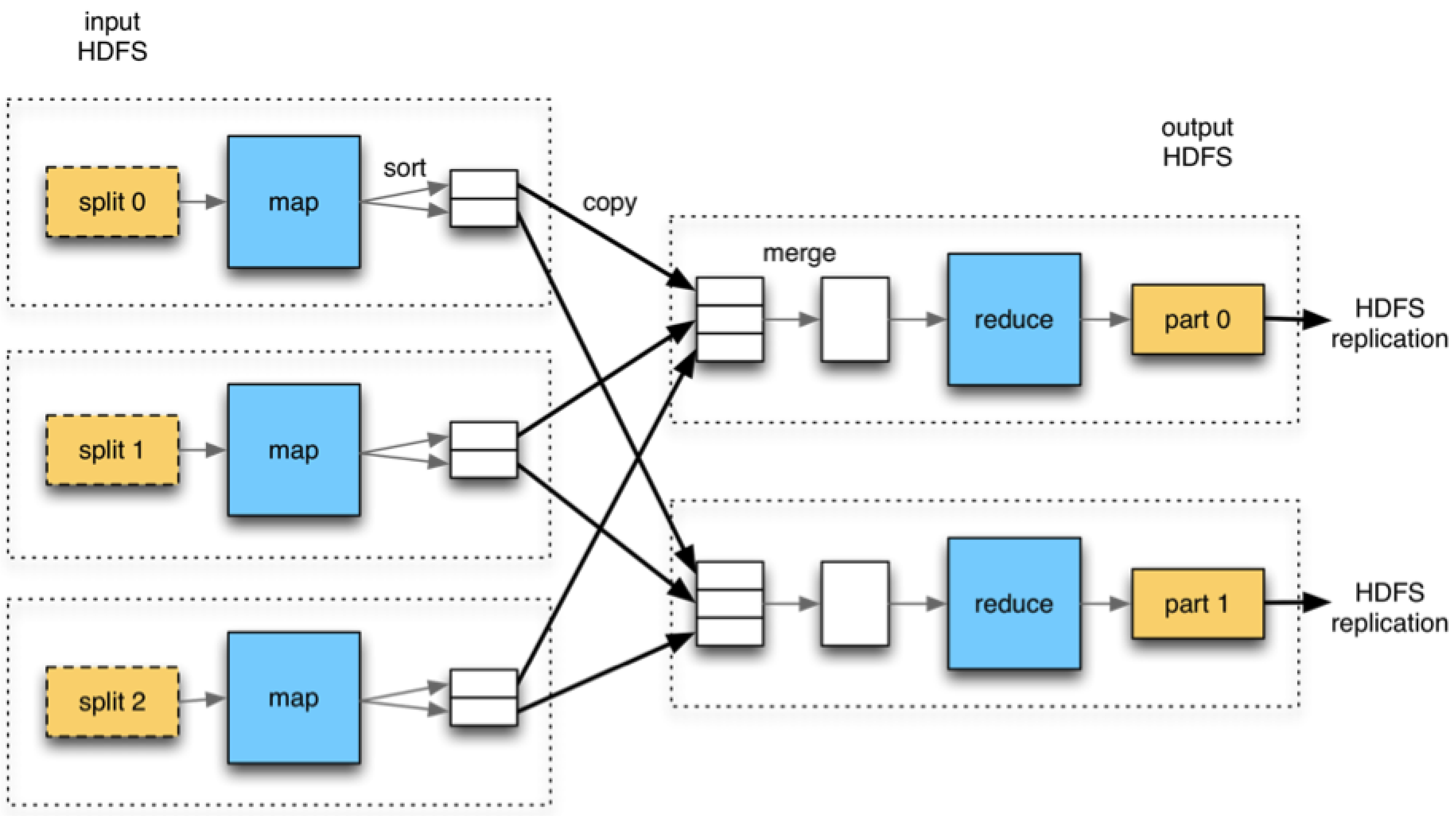
Map/Reduce II

Alexander Lazovik
a.lazovik@rug.nl

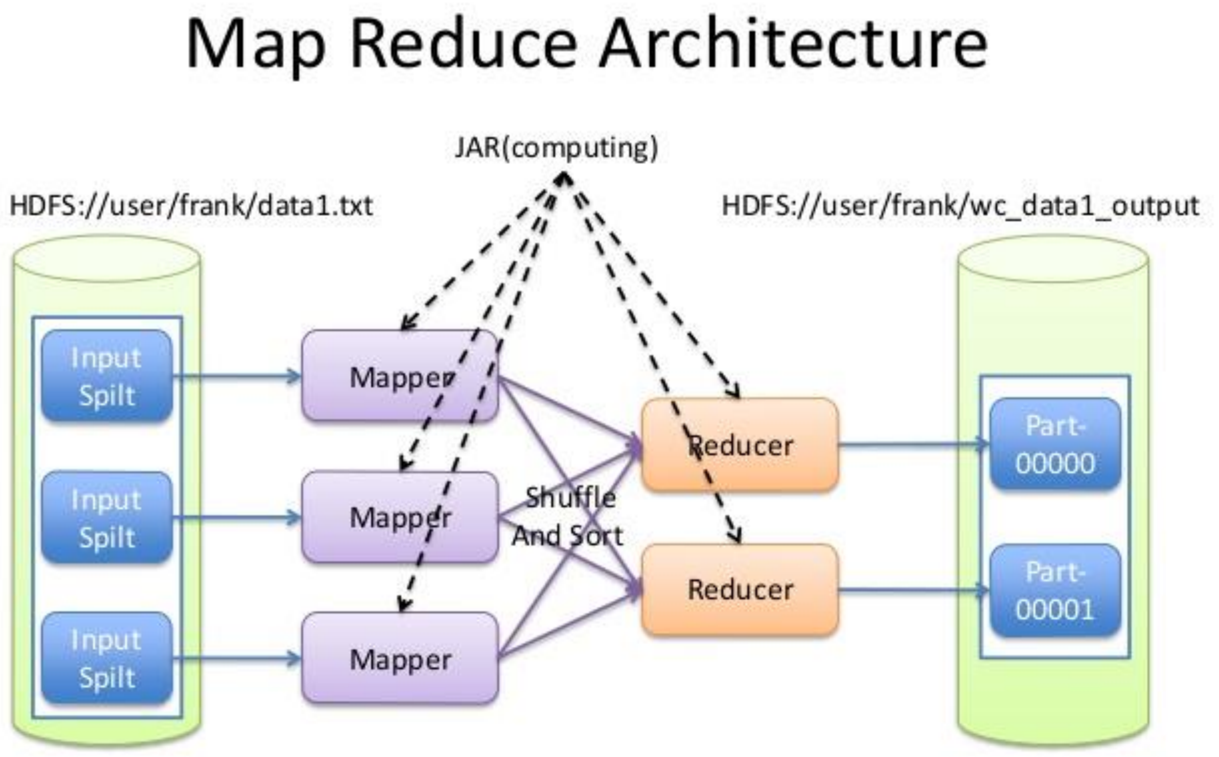
https://www.supinfo.com/articles/single/2807-introduction-to-the-mapreduce-life-cycle

Map/Reduce



https://www.supinfo.com/articles/single/2807-introduction-to-the-mapreduce-life-cycle

Map/Reduce



you, as a developer, have to represent your algorithm as a combination of map and reduce functions

Key/Value pairs

`map:` $(K_1, V_1) \rightarrow \text{list}\ (K_2, V_2)$

- Processes input key/value pair
  - not necessarily key/value as input
- Produces set of intermediate pairs

`reduce:` $(K_2, \text{list}\ V_2) \rightarrow (K_2, V_3)$

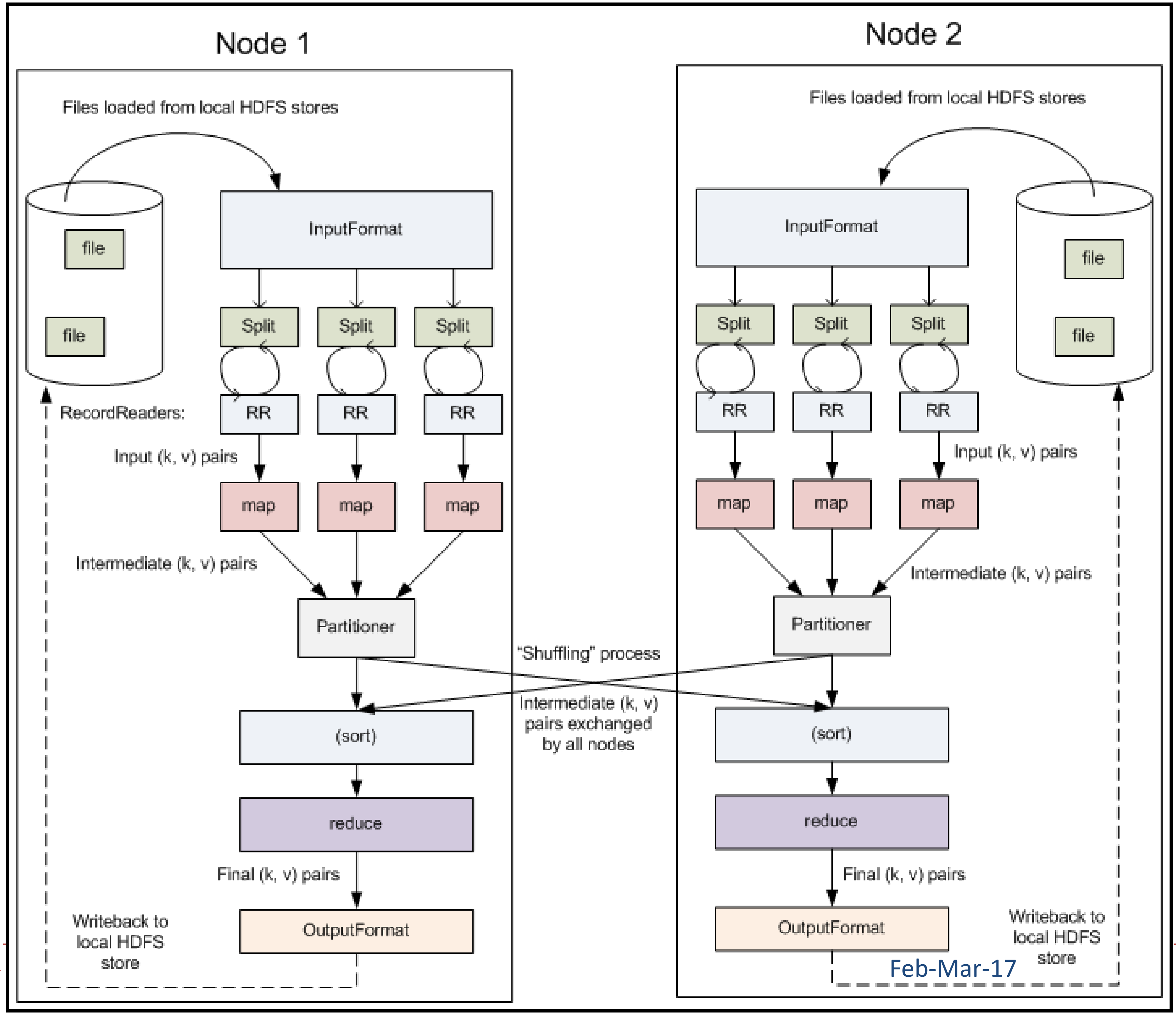
- Combines all intermediate values for a particular key
- Produces a merged output value (can also be a list)

Important observations w.r.t. map and reduce

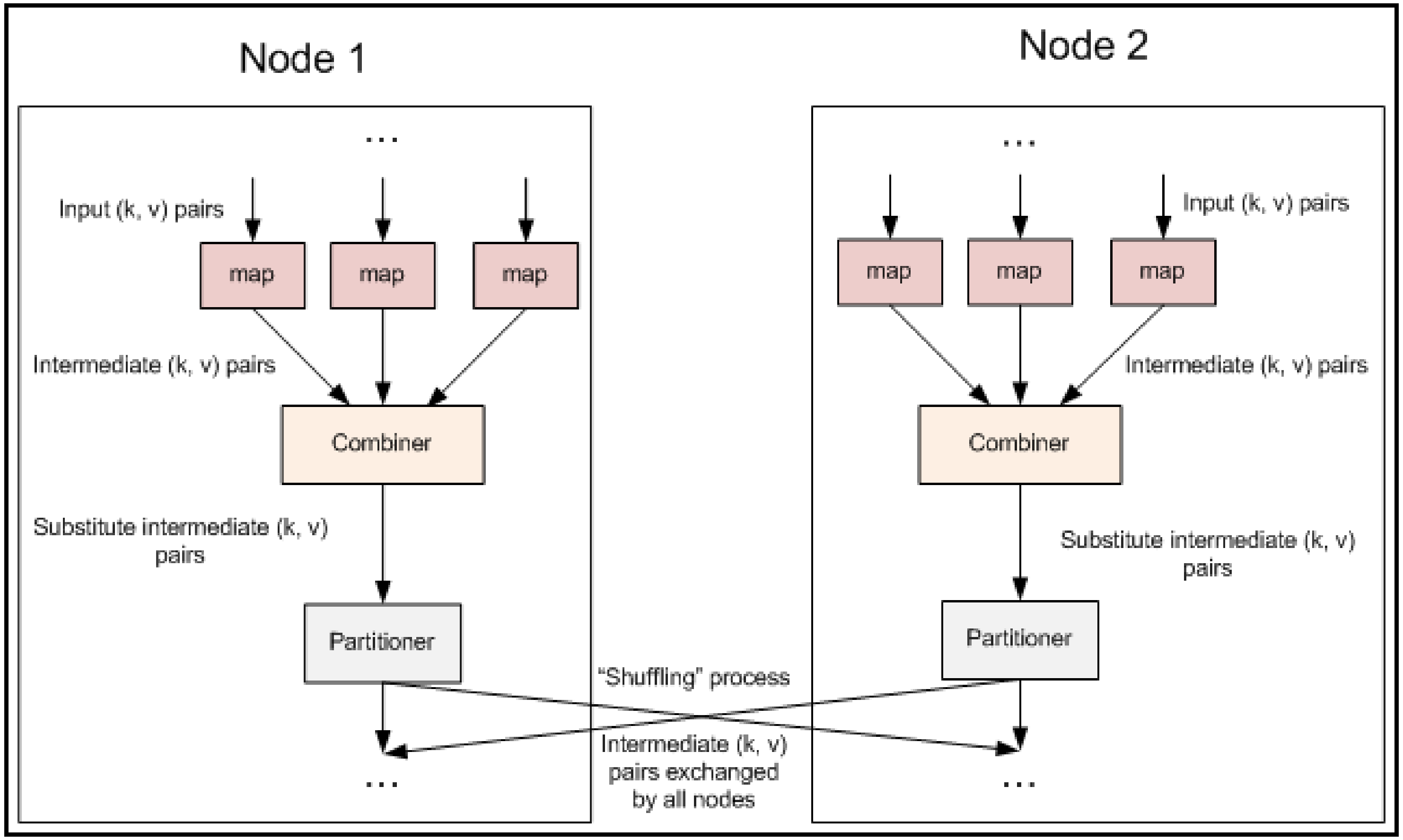
- map can be arbitrary function

- reduce function must be
  - commutative
    - a * b = b * a
    - the order of input does not matter
  - associative
    - (a * b) * c = a * (b * c)
    - reduce can be parallelized
  - existence of neutral element (over input values of reduce)
    - e * a = a * e = a
    - so that map function can generate no output (e element)
      - including situations when a particular map function does not have input

- your own implementation may drop any of the requirements, but then you have to ensure (1) order of input, (2) sequential reduce, (3) no data case


Node 1
Node 2
Files loaded from local HDFS stores
Files loaded from local HDFS stores
file
file
InputFormat
Split
Split
Split
RecordReaders:
RR
RR
RR
Input (k, v) pairs
map
map
map
Intermediate (k, v) pairs
Partitioner
"Shuffling" process
Intermediate (k, v) pairs exchanged by all nodes
(sort)
reduce
Final (k, v) pairs
OutputFormat
Writeback to local HDFS store

https://developer.yahoo.com/hadoop/tutorial/module4.html


Node 1
Node 2
Input (k, v) pairs
map
map
map
Intermediate (k, v) pairs
Combiner
Substitute intermediate (k, v) pairs
Partitioner
"Shuffling" process
Intermediate (k, v) pairs exchanged by all nodes
Input (k, v) pairs
map
map
map
Intermediate (k, v) pairs
Combiner
Substitute intermediate (k, v) pairs
Partitioner

Fault tolerance

- Workers send heartbeats to master
  - If worker fails
    - Re-execute completed and in-progress map tasks
    - Re-execute in-progress reduce tasks
- Master notices particular input key/values cause crashes in map, and skips those values on re-execution.
  - Effect: Can work around bugs in third-party libraries
- Master writes checkpoints periodically to database. If master fails a new master is started
  - Master is a single machine only, so failing is unlikely, easier just to restart the whole MapReduce task

Backup tasks

- The worst case is not when machine is dead, but when it is barely working, really slow
- Slow workers significantly lengthen completion time
  - Other jobs consuming resources on machine
  - Bad disks with soft errors transfer data very slowly
- Solution: Near end of phase, spawn backup copies of tasks
  - Whichever one finishes first "wins"
  - presumes referential transparency of both map and reduce
- Effect: shortens job completion time

Apache Spark

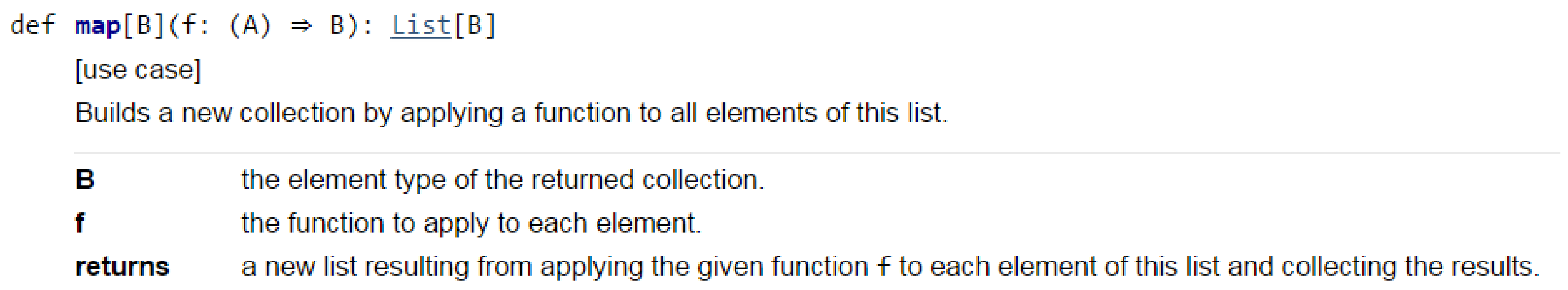
- Issues with Hadoop implementation:
  - low-level map reduce interface
    - a lot of boilerplate code
  - slow
    - intermediate steps are stored on disks

- Apache Spark:
  - better and easier-to-use API
  - faster than Hadoop (claims to be 10x-40x faster)
  - not only Map/Reduce

Apache Spark

- Has a number of high-level libraries / extensions:
  - Spark SQL
    - unifies access to structured data
  - Spark Streaming
    - makes it easy to build scalable fault-tolerant streaming applications
  - MLlib
    - scalable machine learning library
  - GraphX
    - API for graphs and graph-parallel computation

Idea (from functional programming)



```
def map[B](f: (A) ⇒ B): List[B]
```

[use case]

Builds a new collection by applying a function to all elements of this list.

B	the element type of the returned collection.
f	the function to apply to each element.
returns	a new list resulting from applying the given function `f` to each element of this list and collecting the results.

```
pre flatMap (line => line map (s => s -> 1))
```

each map phase "emits" a list of pairs: (key -> value)

In case of word counting, for each word we emit (word, 1) pair

as a result, we have list of list of pairs, which we flatten into a list of pairs

Idea (from functional programming)

```
def map[B](f: (A) ⇒ B): List[B]
```

[use case]

Builds a new collection by applying a function to all elements of this list.

B	the element type of the returned collection.
f	the function to apply to each element.
returns	a new list resulting from applying the given function `f` to each element of this list and collecting the results.

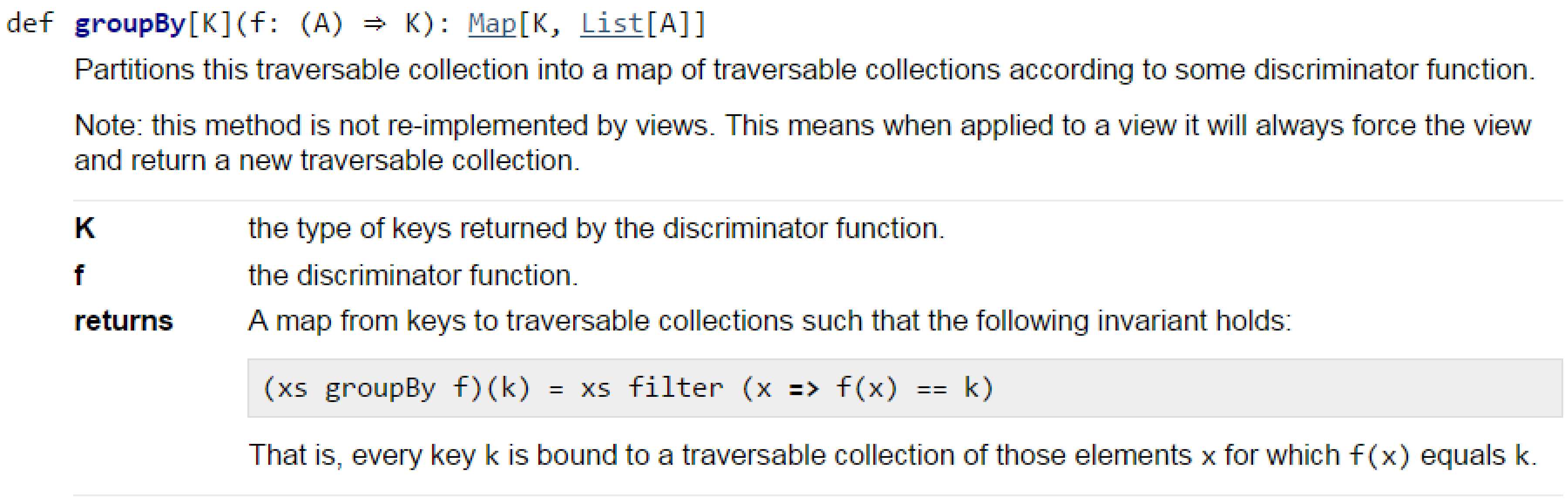
notice: each line is easily parallelizable, independent from others!

```
pre flatMap (line => line map (s => s -> 1))
```

```
for (
  line <- pre;
  s    <- line
) yield {
  s -> 1
}
```

```
(O,1), (let,1), (me,1), (teach,1), (you,1), (how,1), (to,1), (knit,1), (again,1), (This,1), (scattered,1), (corn,1), (into,1),
```

GroupBy



```
def groupBy[K](f: (A) ⇒ K): Map[K, List[A]]
```

Partitions this traversable collection into a map of traversable collections according to some discriminator function.

Note: this method is not re-implemented by views. This means when applied to a view it will always force the view and return a new traversable collection.

K	the type of keys returned by the discriminator function.
f	the discriminator function.
returns	A map from keys to traversable collections such that the following invariant holds:

```
(xs groupBy f)(k) = xs filter (x => f(x) == k)
```

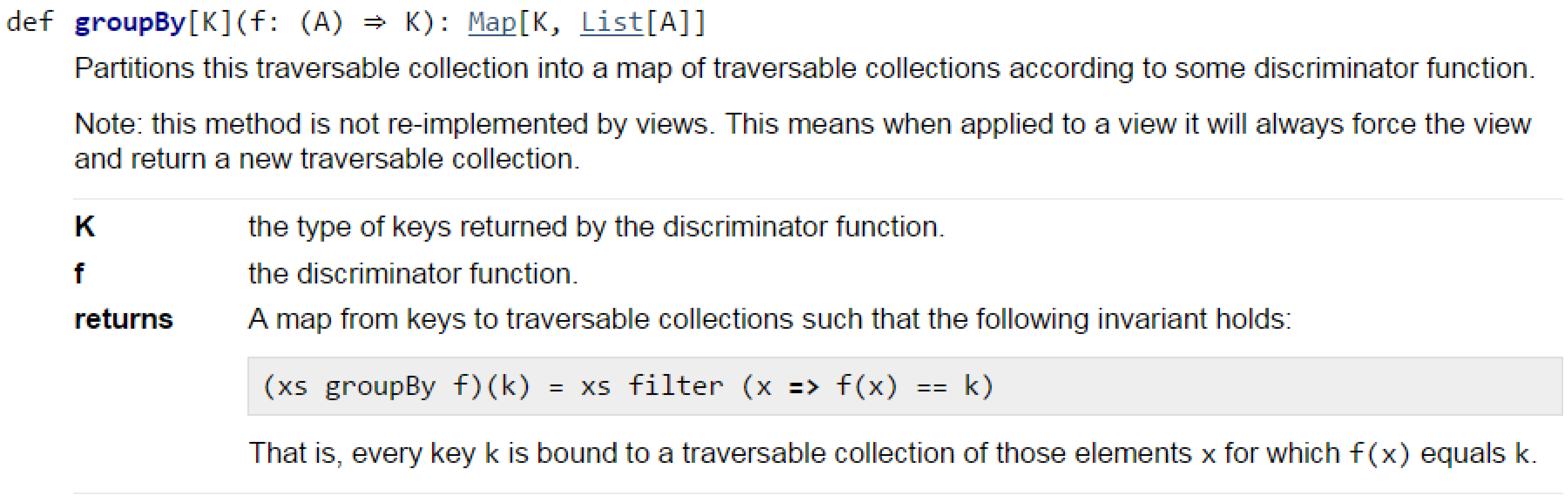
That is, every key k is bound to a traversable collection of those elements x for which `f(x)` equals k.

```
mp.groupBy(kv => kv._1)
```

we group our list of (word,1) by word

```
'Alas' -> List(('Alas',1)), coldness -> List((coldness,1), (coldness,1)),
```

GroupBy



```
def groupBy[K](f: (A) ⇒ K): Map[K, List[A]]
```

Partitions this traversable collection into a map of traversable collections according to some discriminator function.

Note: this method is not re-implemented by views. This means when applied to a view it will always force the view and return a new traversable collection.

K	the type of keys returned by the discriminator function.
f	the discriminator function.
returns	A map from keys to traversable collections such that the following invariant holds:

```
(xs groupBy f)(k) = xs filter (x => f(x) == k)
```

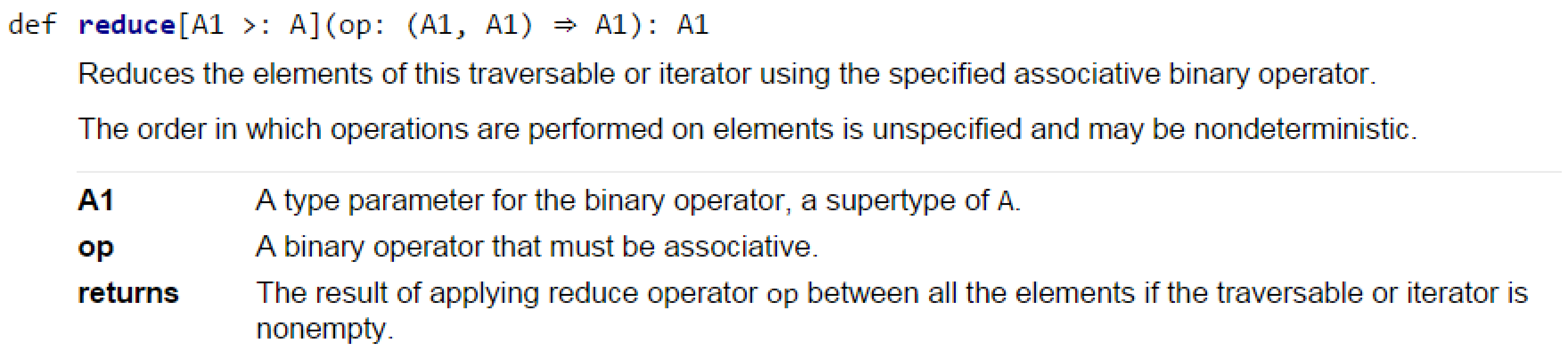
That is, every key k is bound to a traversable collection of those elements x for which `f(x)` equals k.

```
mp.groupBy(kv => kv._1)
'Alas' -> List(('Alas',1)), coldness -> List((coldness,1), (coldness,1)),
```

$\downarrow$

```
val gp  = mp.groupBy(kv => kv._1)
val gpf = gp.map{ case (word, list) => word -> list.map(kv => kv._2)}

, outcast -> List(1, 1), disloyal -> List(1, 1, 1, 1, 1, 1, 1, 1, 1, 1), germane -> List(1, 1),
```

Reduce



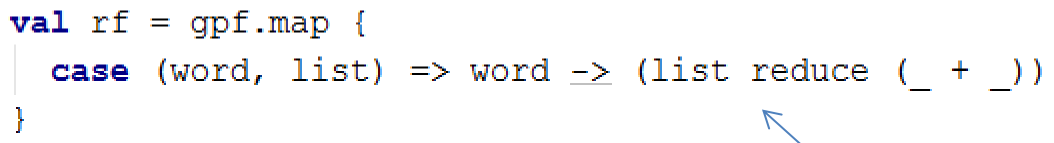
```
def reduce[A1 >: A](op: (A1, A1) ⇒ A1): A1
```

Reduces the elements of this traversable or iterator using the specified associative binary operator.

The order in which operations are performed on elements is unspecified and may be nondeterministic.

A1	A type parameter for the binary operator, a supertype of A.
op	A binary operator that must be associative.
returns	The result of applying reduce operator op between all the elements if the traversable or iterator is nonempty.

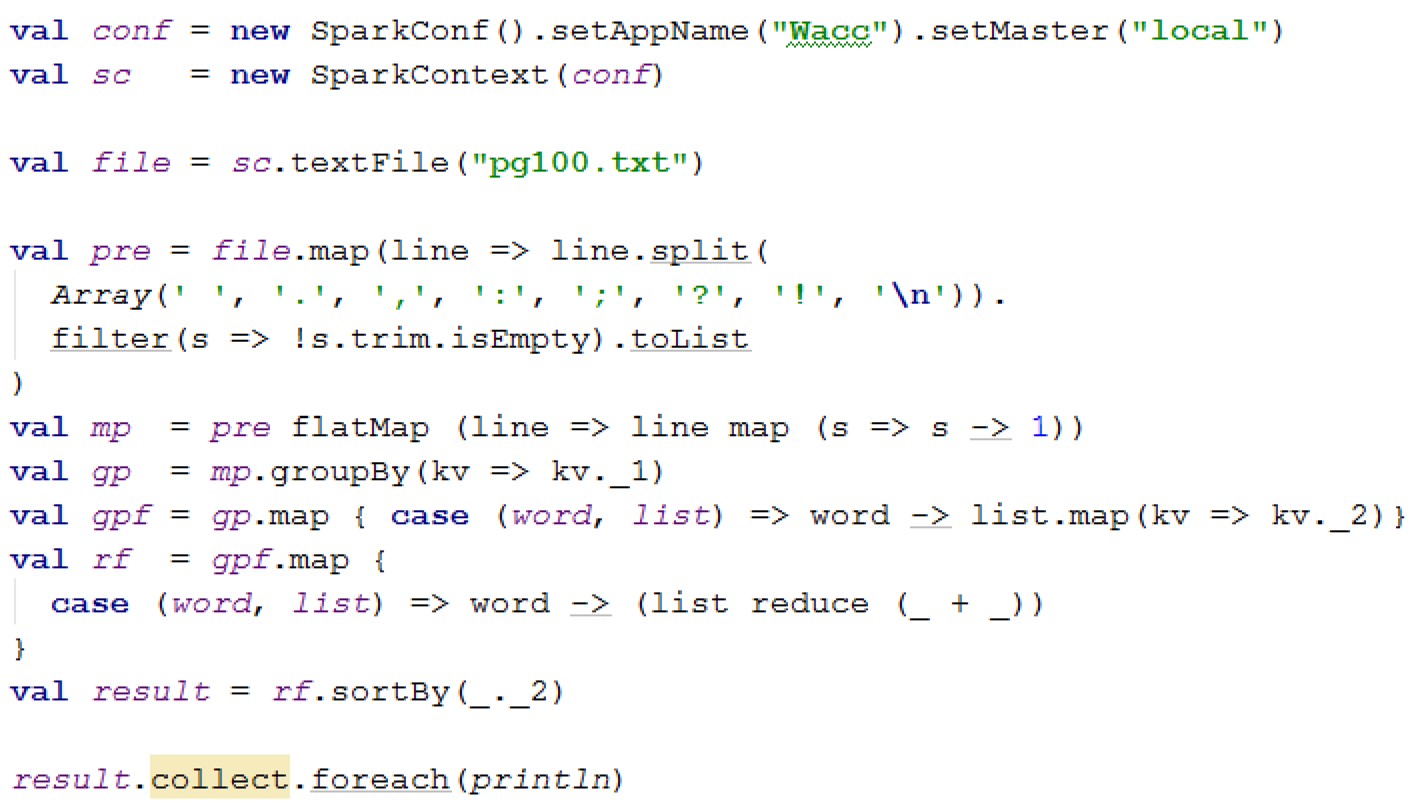
each (word, list) is independent => parallelizable!



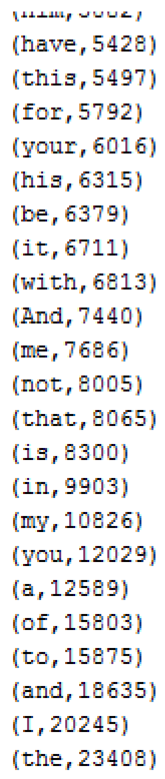
```
val rf = gpf.map {
  case (word, list) => word -> (list reduce (_ + _))
}
```

reduce step: transform (K, List of Values) into (K, Value)

Word Counting Revisited (Spark)



```
val conf = new SparkConf().setAppName("Wacc").setMaster("local")
val sc   = new SparkContext(conf)

val file = sc.textFile("pg100.txt")

val pre = file.map(line => line.split(
  Array(' ', '.', ',', ':', ';', '?', '!', '\n')).
  filter(s => !s.trim.isEmpty).toList
)
val mp  = pre flatMap (line => line map (s => s -> 1))
val gp  = mp.groupBy(kv => kv._1)
val gpf = gp.map { case (word, list) => word -> list.map(kv => kv._2)}
val rf  = gpf.map {
  case (word, list) => word -> (list reduce (_ + _))
}
val result = rf.sortBy(_._2)

result.collect.foreach(println)
```



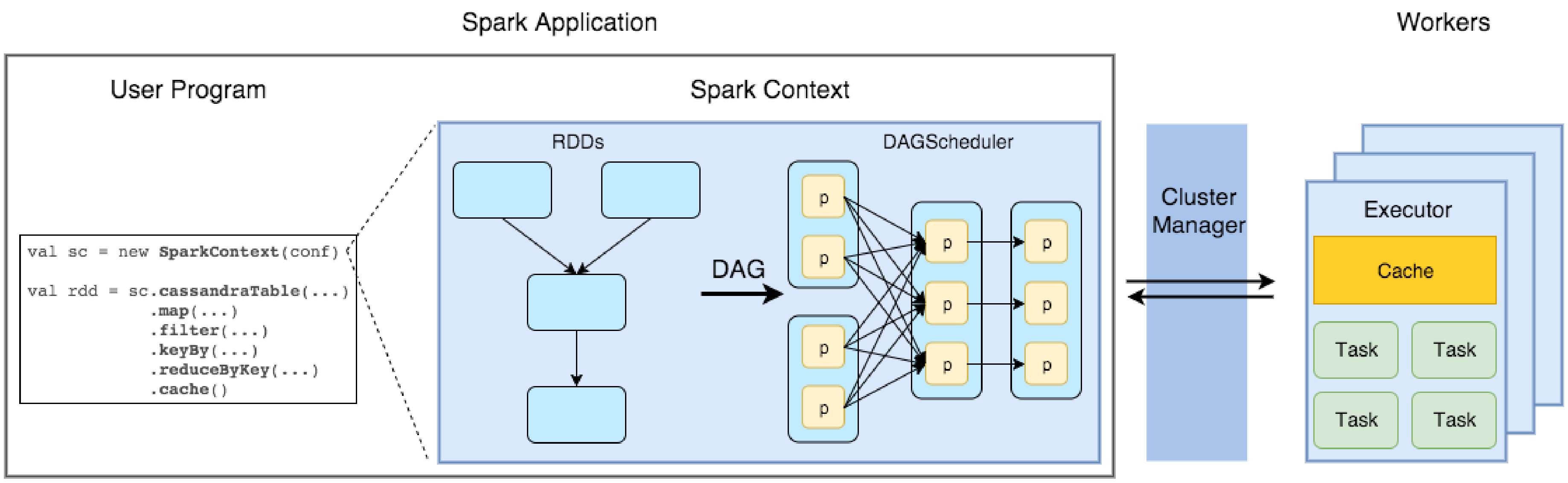
```
(have,5428)
(this,5497)
(for,5792)
(your,6016)
(his,6315)
(be,6379)
(it,6711)
(with,6813)
(And,7440)
(me,7686)
(not,8005)
(that,8065)
(is,8300)
(in,9903)
(my,10826)
(you,12029)
(a,12589)
(of,15803)
(to,15875)
(and,18635)
(I,20245)
(the,23408)
```

Map/Reduce algorithm:
map -> groupBy -> reduce

http://datastrophic.io/core-concepts-architecture-and-internals-of-apache-spark/

https://www.slideshare.net/akirillov/spark-workshop-internals-architecture-and-coding-59035491

Spark internals



```
val sc = new SparkContext(conf)

val rdd = sc.cassandraTable(...)
            .map(...)
            .filter(...)
            .keyBy(...)
            .reduceByKey(...)
            .cache()
```

- you data is represented via RDDs (resilient distributed dataset)
  - distributed (partitioned) immutable data
  - with API for iterator-like (lazy) transformations

- note: since recently, RDD API is not recommended
  - instead: Dataframe and Dataset APIs
  - too low level for query optimizations

http://datastrophic.io/core-concepts-architecture-and-internals-of-apache-spark/

https://www.slideshare.net/akirillov/spark-workshop-internals-architecture-and-coding-59035491

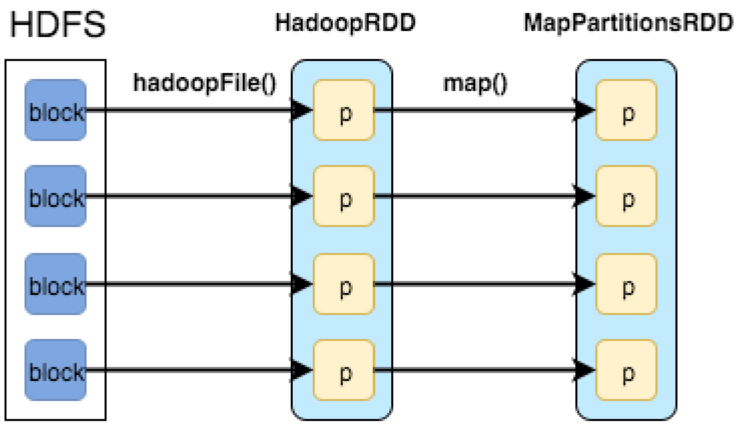
RDD API

```
//a list of partitions (e.g. splits in Hadoop)
def getPartitions: Array[Partition]

//a list of dependencies on other RDDs
def getDependencies: Seq[Dependency[_]]

//a function for computing each split
def compute(split: Partition, context: TaskContext): Iterator[T]


//(optional) a list of preferred locations to compute each split on
def getPreferredLocations(split: Partition): Seq[String] = Nil

//(optional) a partitioner for key-value RDDs
val partitioner: Option[Partitioner] = None
```

http://datastrophic.io/core-concepts-architecture-and-internals-of-apache-spark/

https://www.slideshare.net/akirillov/spark-workshop-internals-architecture-and-coding-59035491

sparkContext.textFile("hdfs://...")

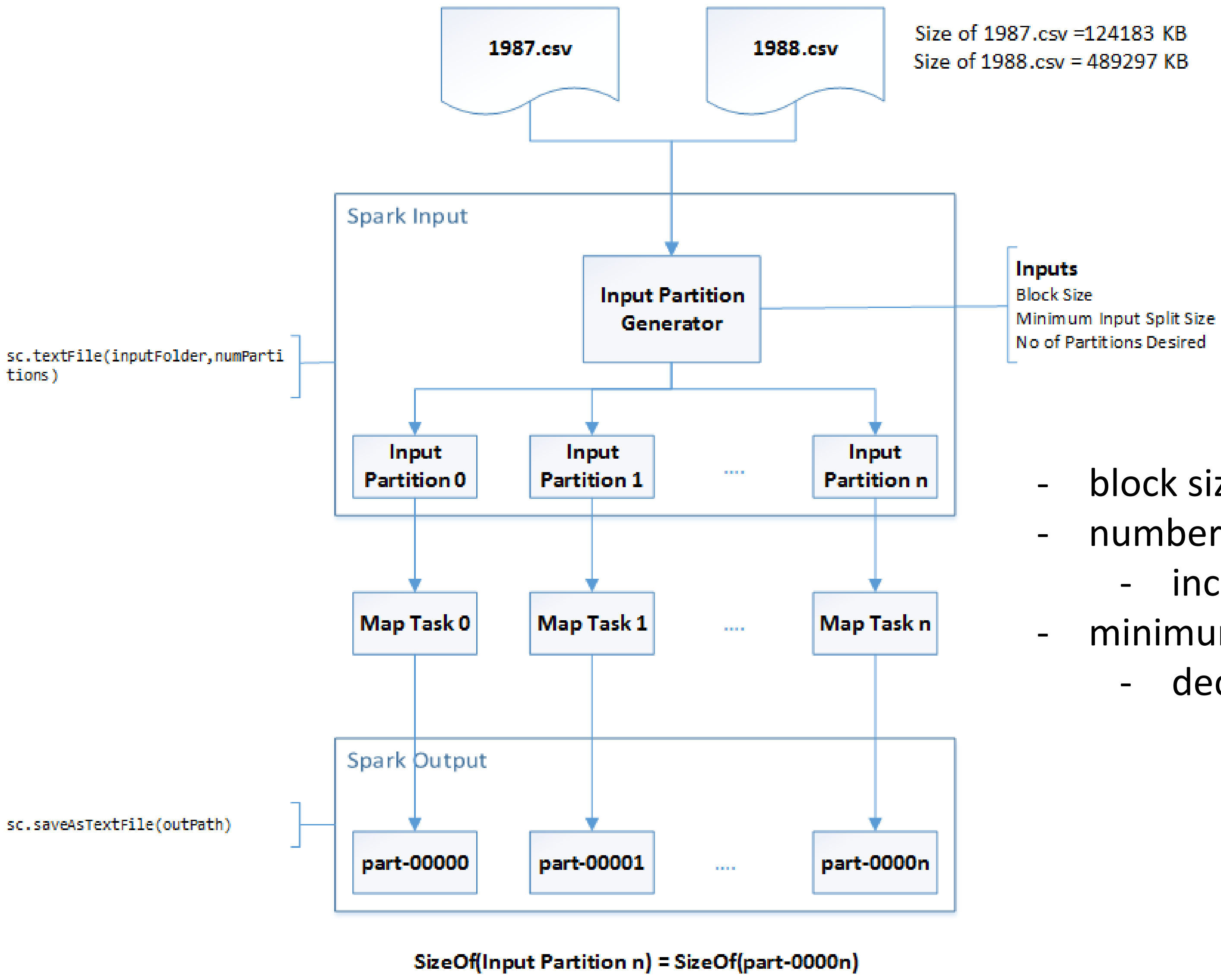


HadoopRDD:
- getPartitions = HDFS blocks
- getDependencies = None
- compute = load block in memory
- getPrefferedLocations = HDFS block locations
- partitioner = None

MapPartitionsRDD:
- getPartitions = same as parent
- getDependencies = parent RDD
- compute = compute parent and apply map()
- getPrefferedLocations = same as parent
- partitioner = None

http://www.bigsynapse.com/spark-input-output

Number of partitions

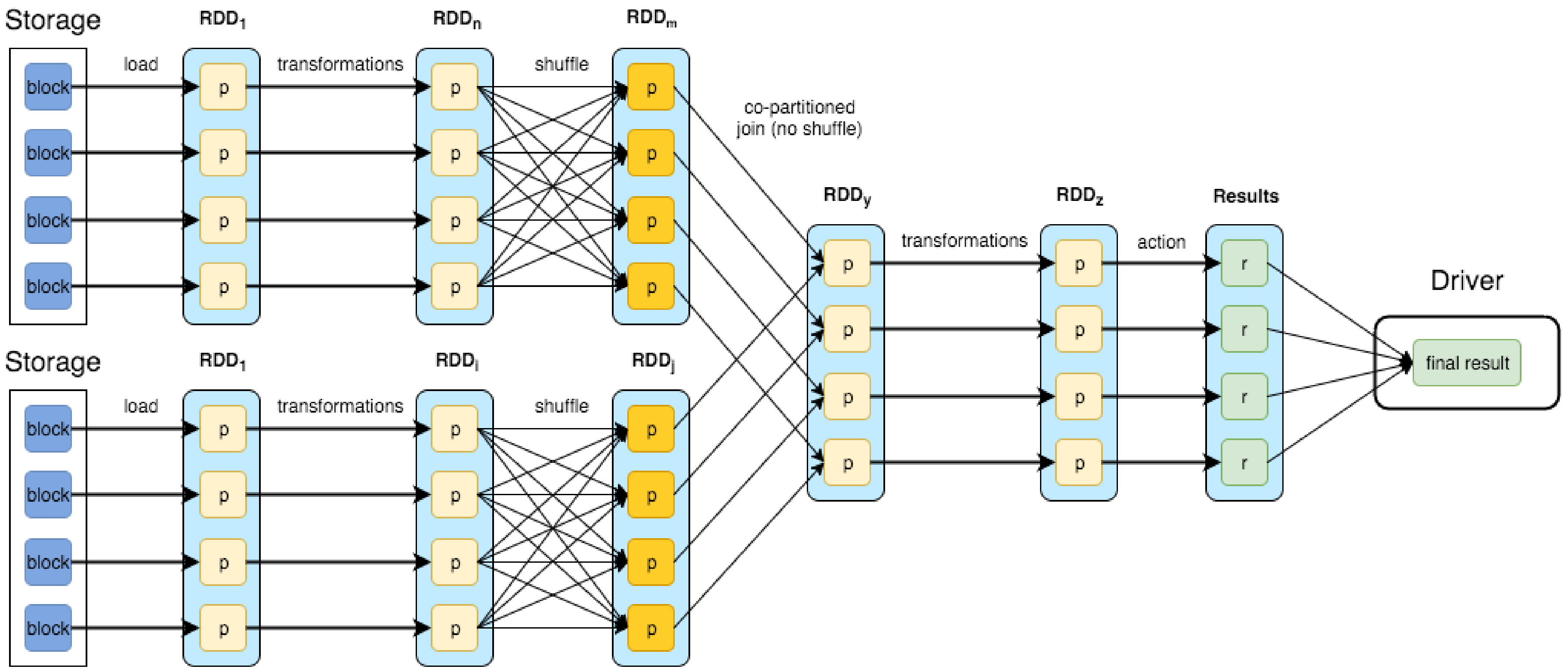


- block size -> default
- number of partitions
  - increase number
- minimum input split size
  - decrease number

http://datastrophic.io/core-concepts-architecture-and-internals-of-apache-spark/

https://www.slideshare.net/akirillov/spark-workshop-internals-architecture-and-coding-59035491

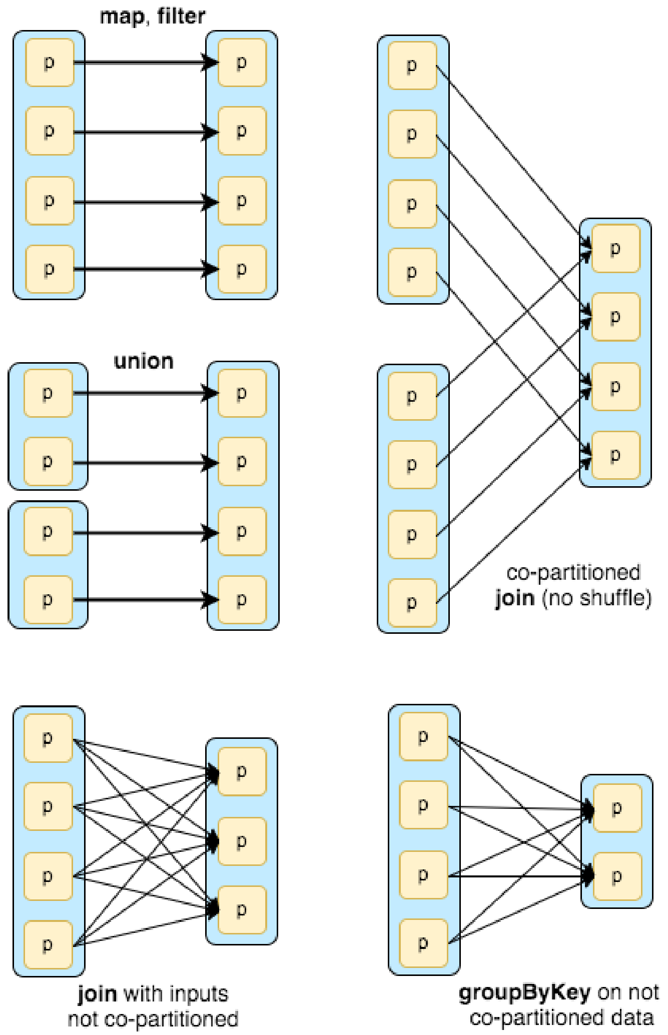
Execution DAG



http://datastrophic.io/core-concepts-architecture-and-internals-of-apache-spark/

https://www.slideshare.net/akirillov/spark-workshop-internals-architecture-and-coding-59035491

Staging



map, filter
union
co-partitioned join (no shuffle)
join with inputs not co-partitioned
groupByKey on not co-partitioned data
p

Narrow (pipeline-able):
- each partition of the parent RDD is used by at most one partition of the child RDD
- allow for pipelined execution on one cluster node
- failure recovery is more efficient as only lost parent partitions need to be recomputed

Wide (shuffle)
- multiple child partitions may depend on one parent partition
- require data from all parent partitions to be available and to be shuffled across the nodes
- if some partition is lost from all the ancestors a complete recomputation is needed

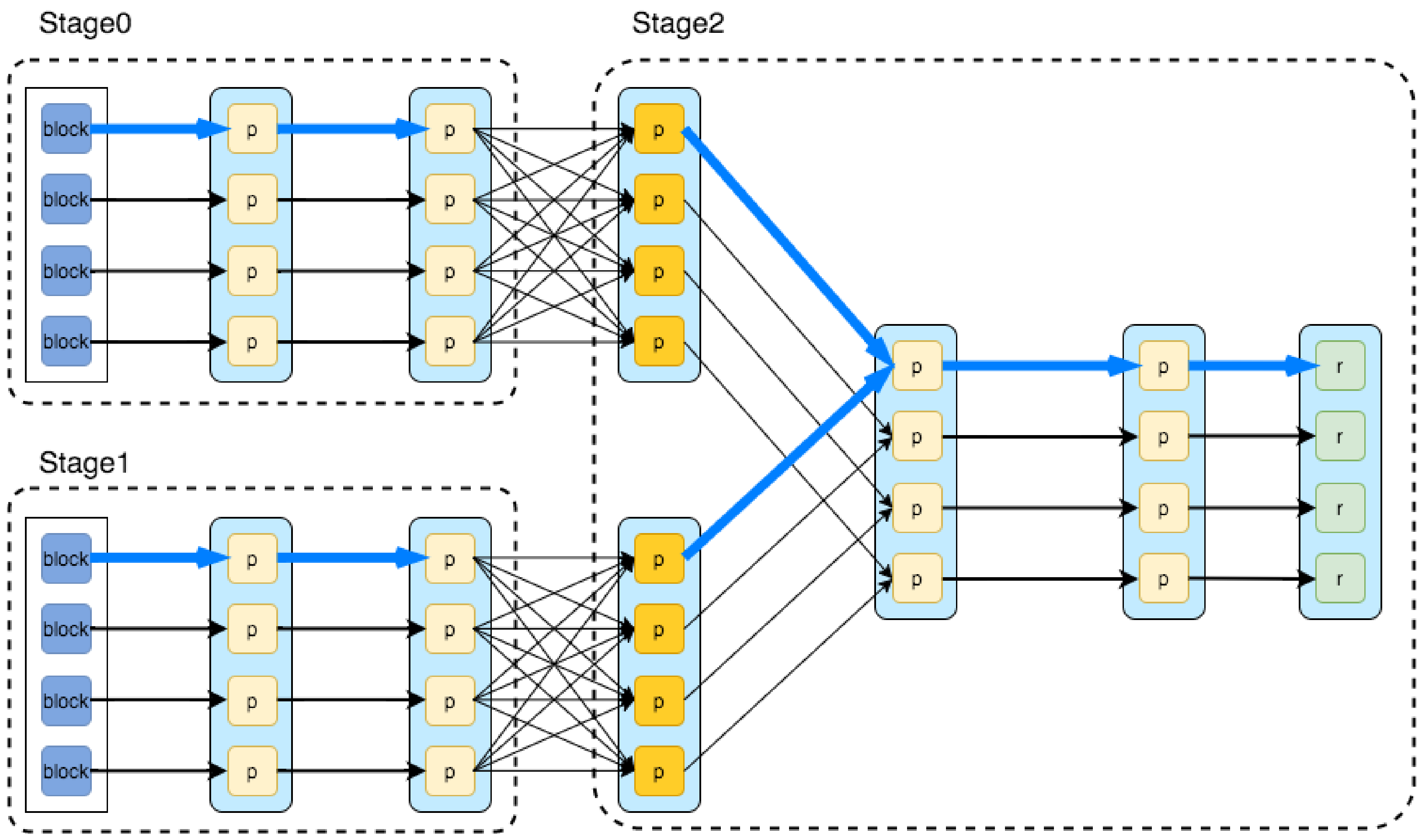
RDD operations with "narrow" dependencies, like map() and filter(), are pipelined together into one set of tasks in each stage

Operations with shuffle dependencies require multiple stages (one to write a set of map output files, and another to read those files after a barrier)

Every stage will have only shuffle dependencies on other stages http://datastrophic.io/core-concepts-architecture-and-internals-of-apache-spark/

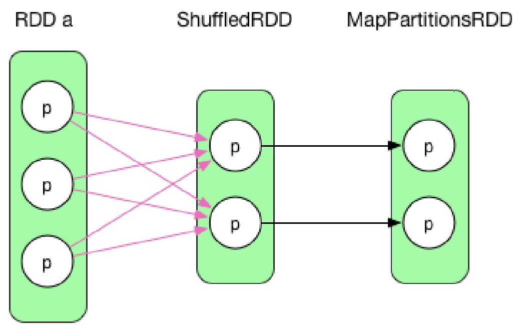
https://www.slideshare.net/akirillov/spark-workshop-internals-architecture-and-coding-59035491

Splitting DAG into stages

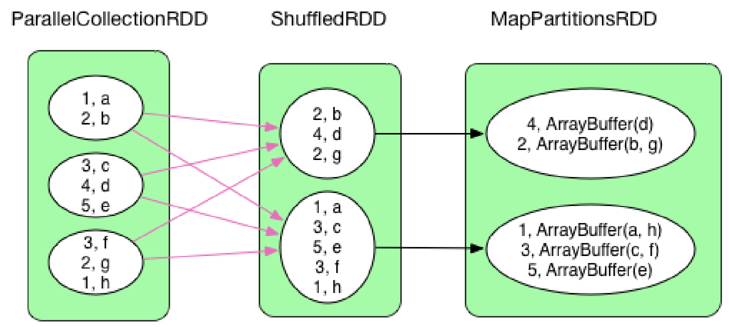


groupByKey vs reduceByKey

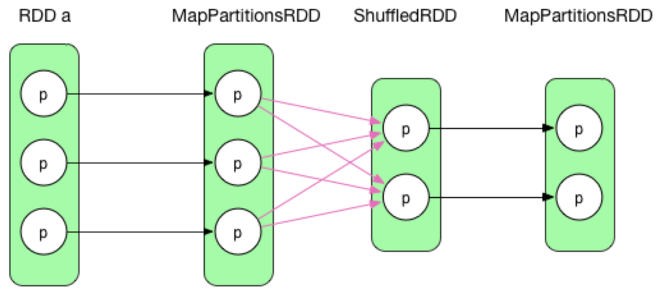
groupByKey(numPartitions)



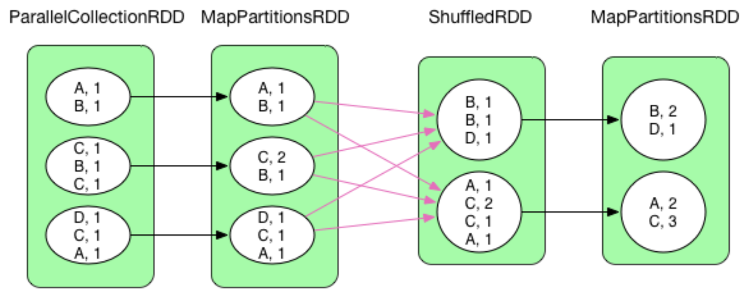
Example: groupByKey(2)



reduceByKey(f, numPartitions)

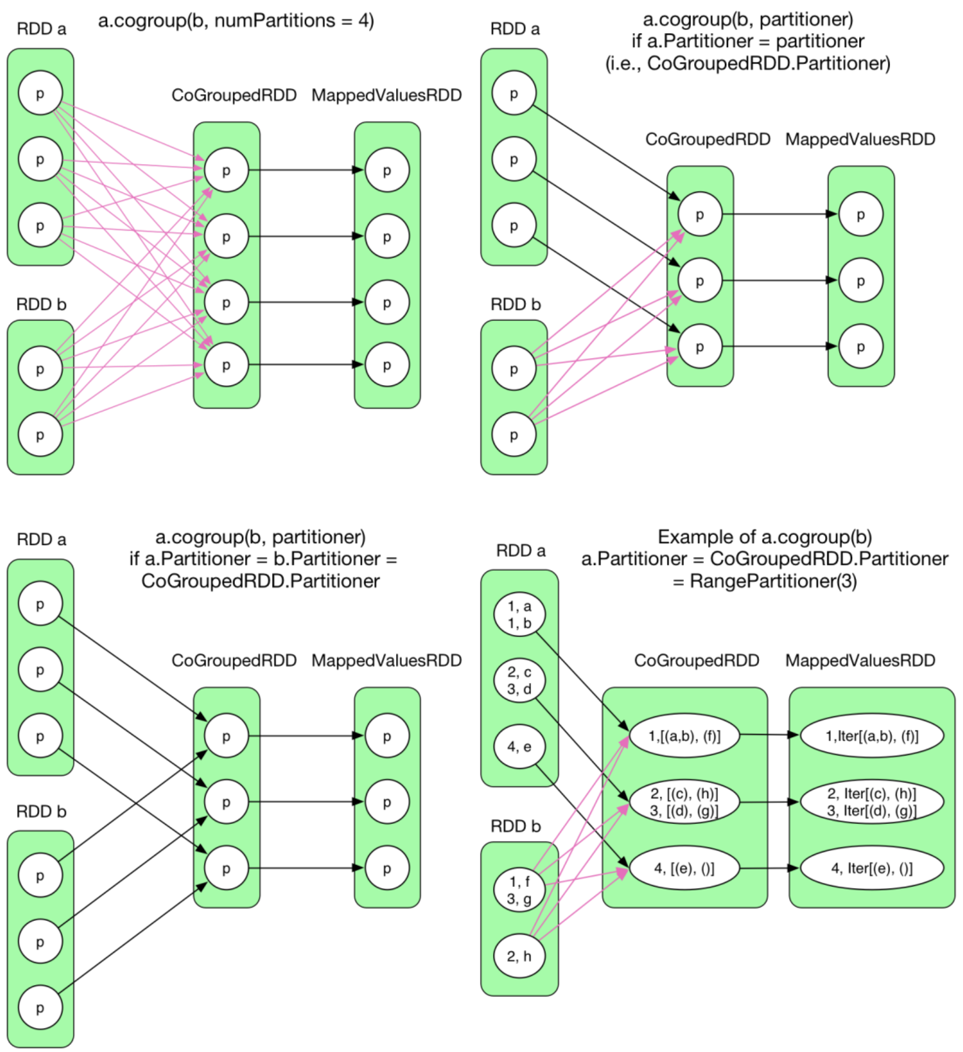


Example (WordCount): reduceByKey(_ + _, 2)

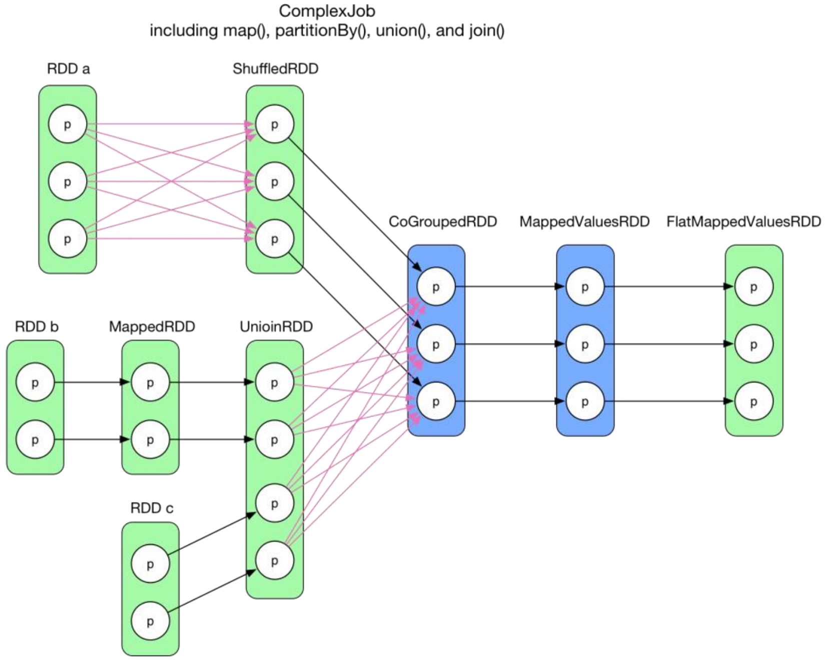


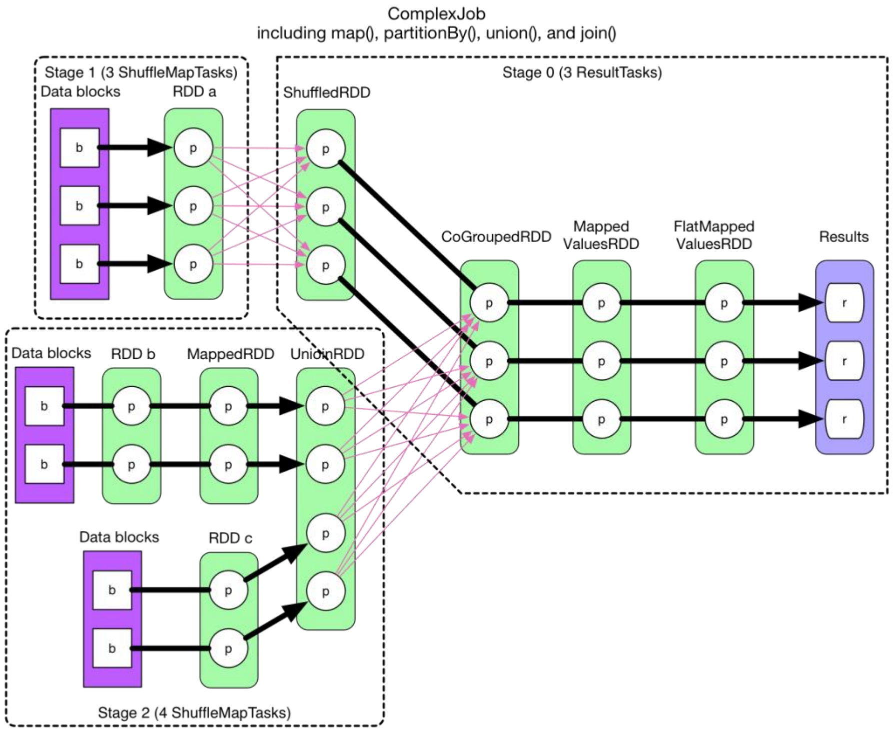
reduceByKey is "associative"-friendly

cogroup


a.cogroup(b, numPartitions = 4)
RDD a
RDD b
CoGroupedRDD
MappedValuesRDD
p
a.cogroup(b, partitioner)
if a.Partitioner = partitioner
(i.e., CoGroupedRDD.Partitioner)
a.cogroup(b, partitioner)
if a.Partitioner = b.Partitioner =
CoGroupedRDD.Partitioner
Example of a.cogroup(b)
a.Partitioner = CoGroupedRDD.Partitioner
= RangePartitioner(3)
1, a
1, b
2, c
3, d
4, e
1, f
3, g
2, h
1,[(a,b), (f)]
2, [(c), (h)]
3, [(d), (g)]
4, [(e), ()]
1,Iter[(a,b), (f)]
2, Iter[(c), (h)]
3, Iter[(d), (g)]
4, Iter[(e), ()]

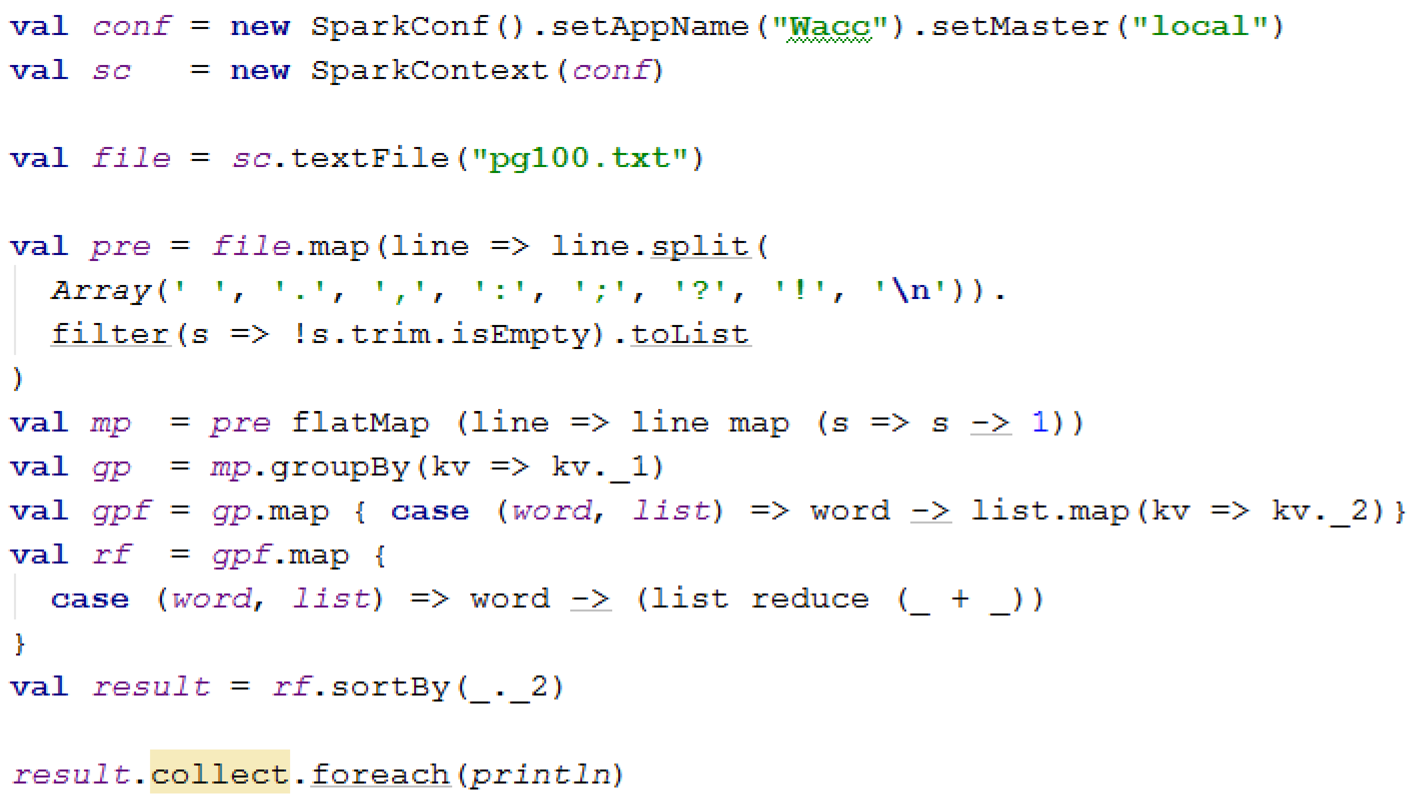
Physical execution plan



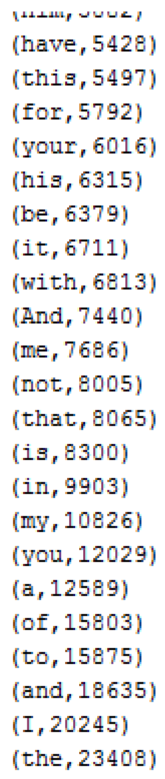


- executor examines the plan from right to left
- all dependencies (on stages) must be executed first
- inside of a stage, pipeline tasks are executed in parallel

Word Counting Revisited (Spark)



```scala
val conf = new SparkConf().setAppName("Wacc").setMaster("local")
val sc   = new SparkContext(conf)

val file = sc.textFile("pg100.txt")

val pre = file.map(line => line.split(
  Array(' ', '.', ',', ':', ';', '?', '!', '\n')).
  filter(s => !s.trim.isEmpty).toList
)
val mp  = pre flatMap (line => line map (s => s -> 1))
val gp  = mp.groupBy(kv => kv._1)
val gpf = gp.map { case (word, list) => word -> list.map(kv => kv._2)}
val rf  = gpf.map {
  case (word, list) => word -> (list reduce (_ + _))
}
val result = rf.sortBy(_._2)

result.collect.foreach(println)
```



```
(have,5428)
(this,5497)
(for,5792)
(your,6016)
(his,6315)
(be,6379)
(it,6711)
(with,6813)
(And,7440)
(me,7686)
(not,8005)
(that,8065)
(is,8300)
(in,9903)
(my,10826)
(you,12029)
(a,12589)
(of,15803)
(to,15875)
(and,18635)
(I,20245)
(the,23408)
```

Map/Reduce algorithm:
map -> groupBy -> reduce

Word Counting Revisited

```scala
val conf = new SparkConf().setAppName("Wacc").setMaster("local")
val sc   = new SparkContext(conf)

val file = sc.textFile("pg100.txt")

val pre = file.map(line => line.split(
  Array(' ', '.', ',', ':', ';', '?', '!', '\n')).
  filter(s => !s.trim.isEmpty).toList
)
val mp = pre flatMap (line => line map (s => s -> 1))
val rf = mp.reduceByKey((a, b) => a + b)

val result = rf.sortBy(_._2)

result.collect.foreach(println)
```

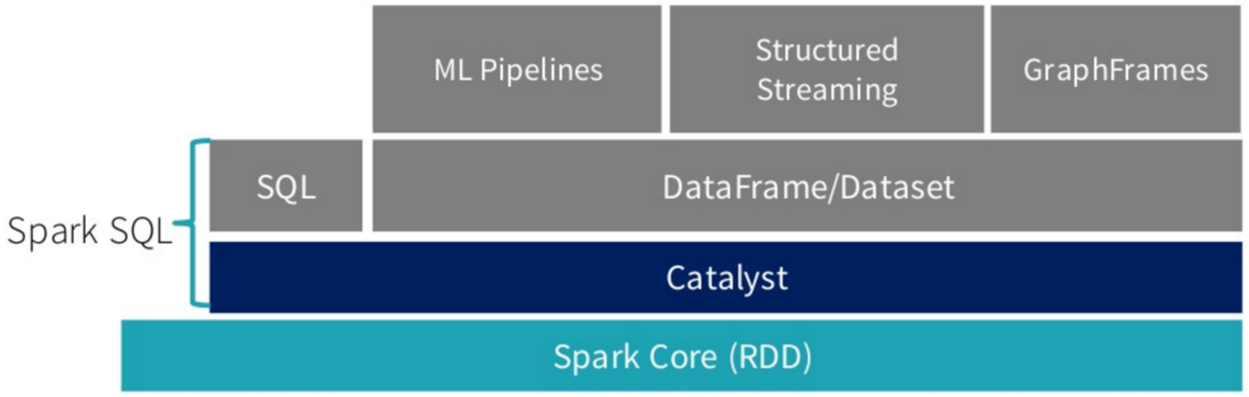
```
(him,5082)
(have,5428)
(this,5497)
(for,5792)
(your,6016)
(his,6315)
(be,6379)
(it,6711)
(with,6813)
(And,7440)
(me,7686)
(not,8005)
(that,8065)
(is,8300)
(in,9903)
(my,10826)
(you,12029)
(a,12589)
(of,15803)
(to,15875)
(and,18635)
(I,20245)
(the,23408)
```

Completed Stages (3)

Stage Id	Description
0	collect at SparkMain.scala:23
2	sortBy at SparkMain.scala:21
1	flatMap at SparkMain.scala:18

https://spark-summit.org/eu-2016/events/a-deep-dive-into-the-catalyst-optimizer/

Why not RDD?



- limiting of what can be expressed enables optimization
  - essentially, instead of using user generic functions, you do give a bit of insight into your query to Spark

Why not RDD?

RDD

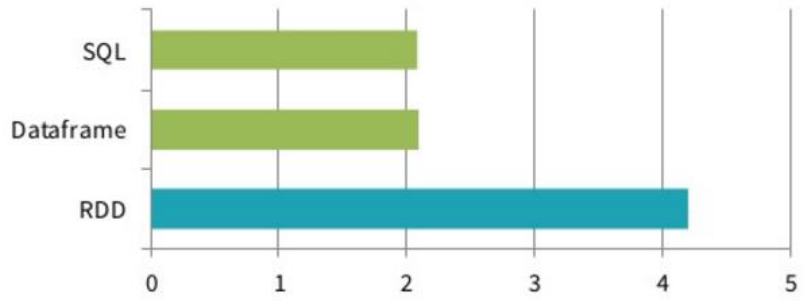
```
pdata.map { case (dpt, age) => dpt -> (age, 1) }
     .reduceByKey { case ((a1, c1), (a2, c2)) => (a1 + a2, c1 + c2)}
     .map { case (dpt, (age, c)) => dpt -> age/ c }
```

Dataframe

```
data.groupBy("dept").avg("age")
```

SQL

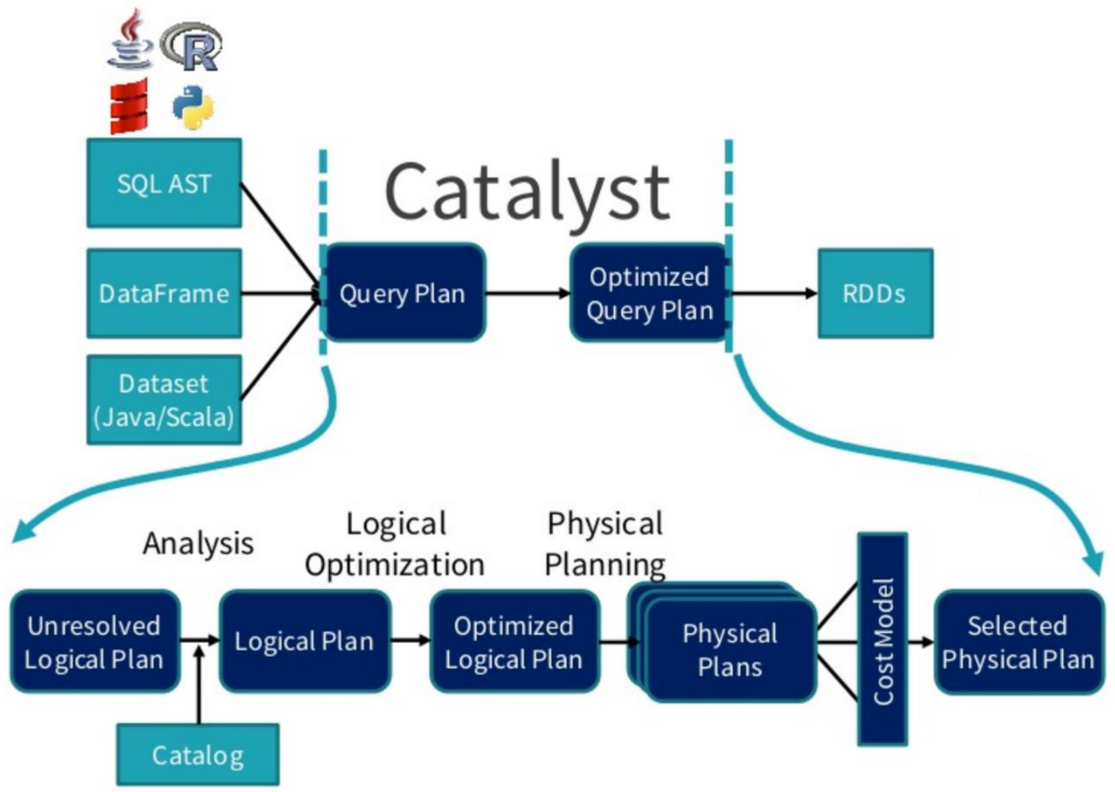
```
select dept, avg(age) from data group by 1
```



Runtime performance of aggregating 10 million int pairs (secs)

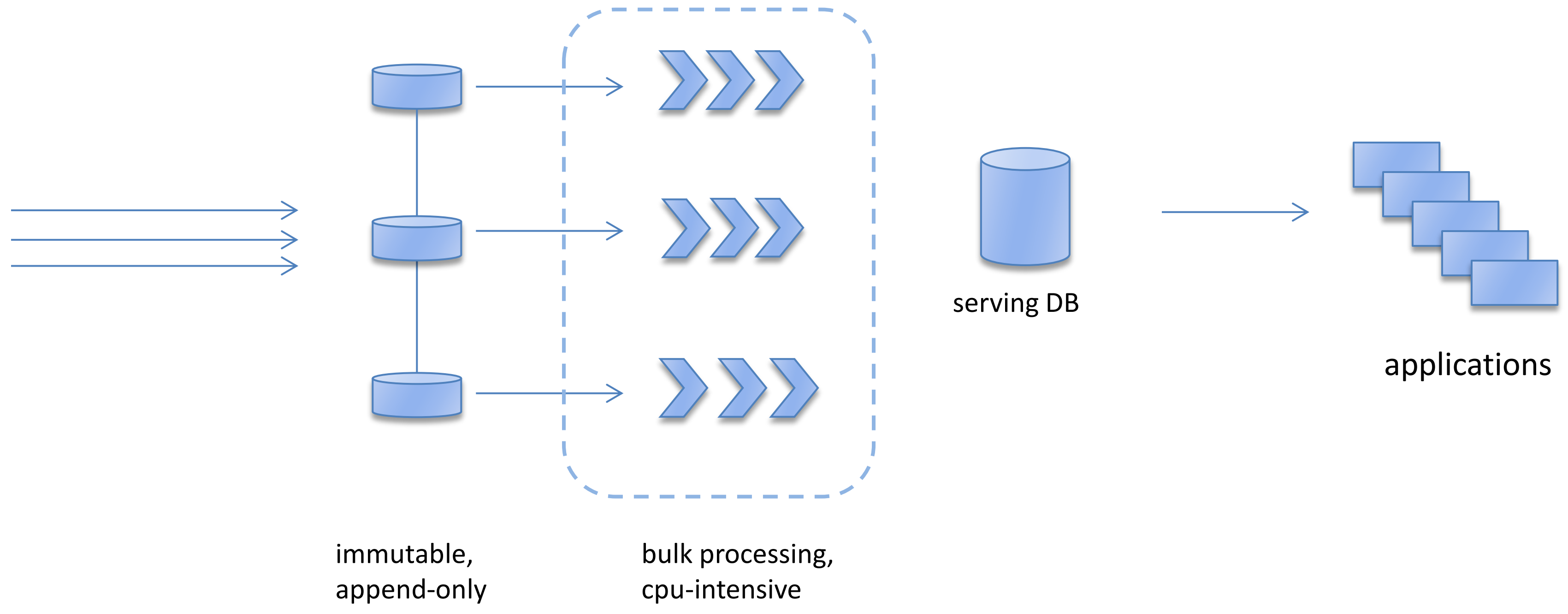
- query optimizer may come up with a better execution plan
  - similar to those of relational databases

Why not RDD?

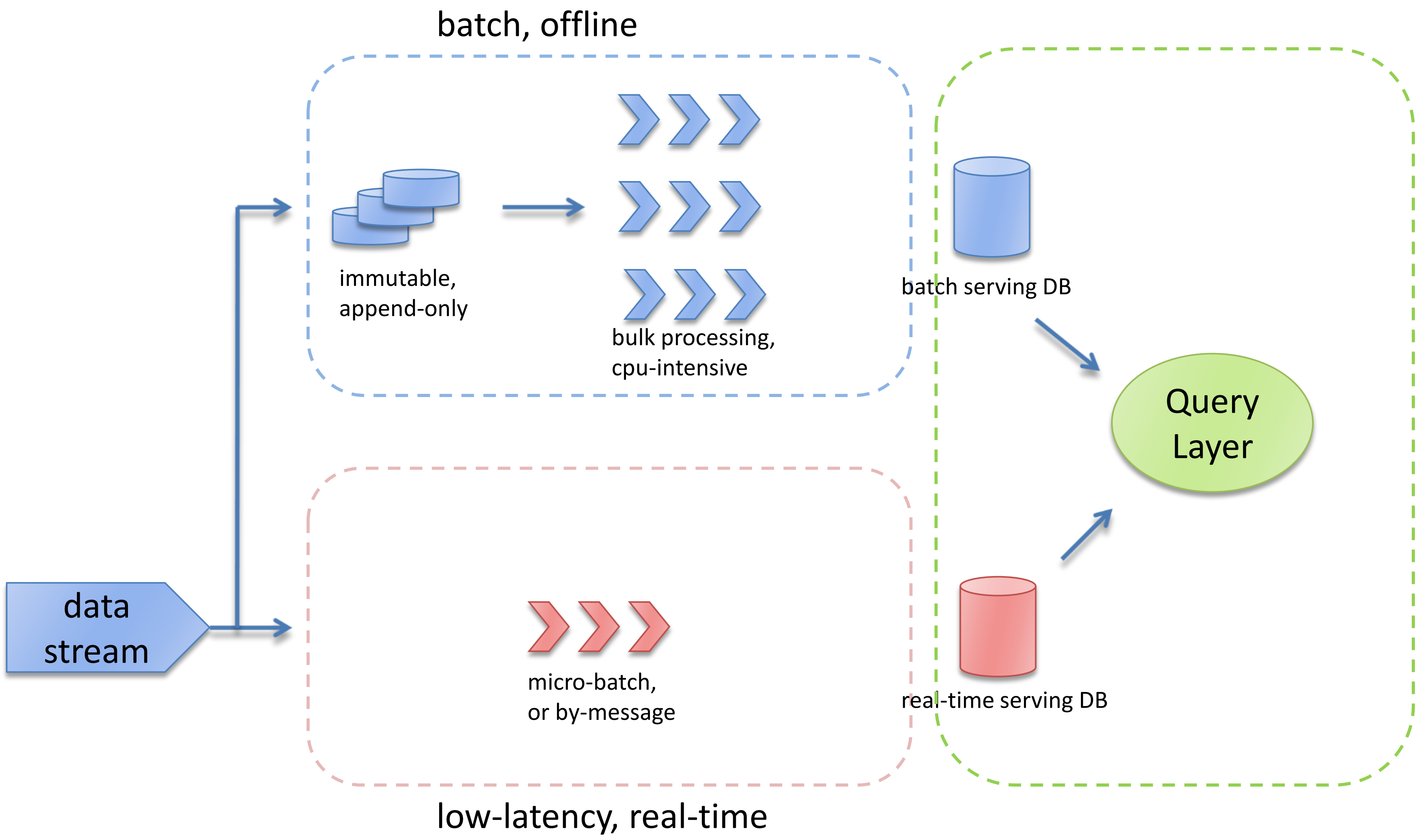


- you can write your own rules for optimization
  - rule is just a partial function that transforms your logical/physical plans
  - to exploit the specific structure of the problem

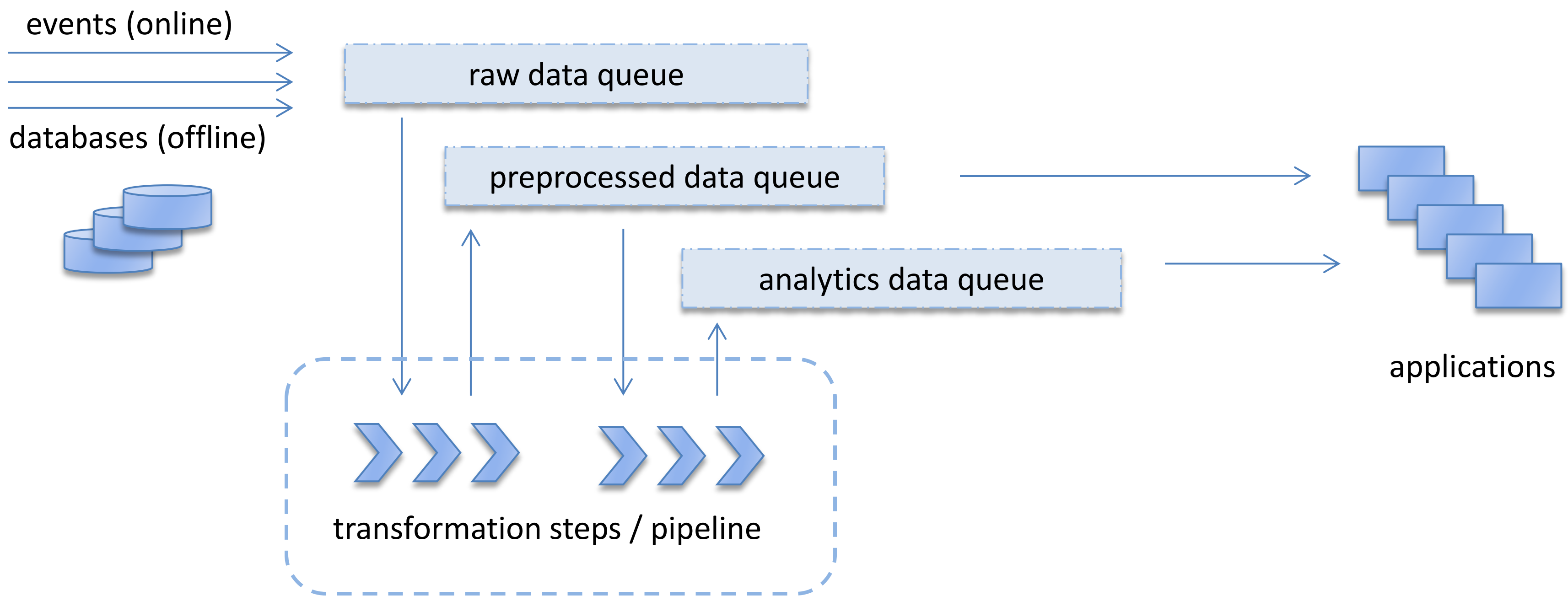
HDFS / “data lake”



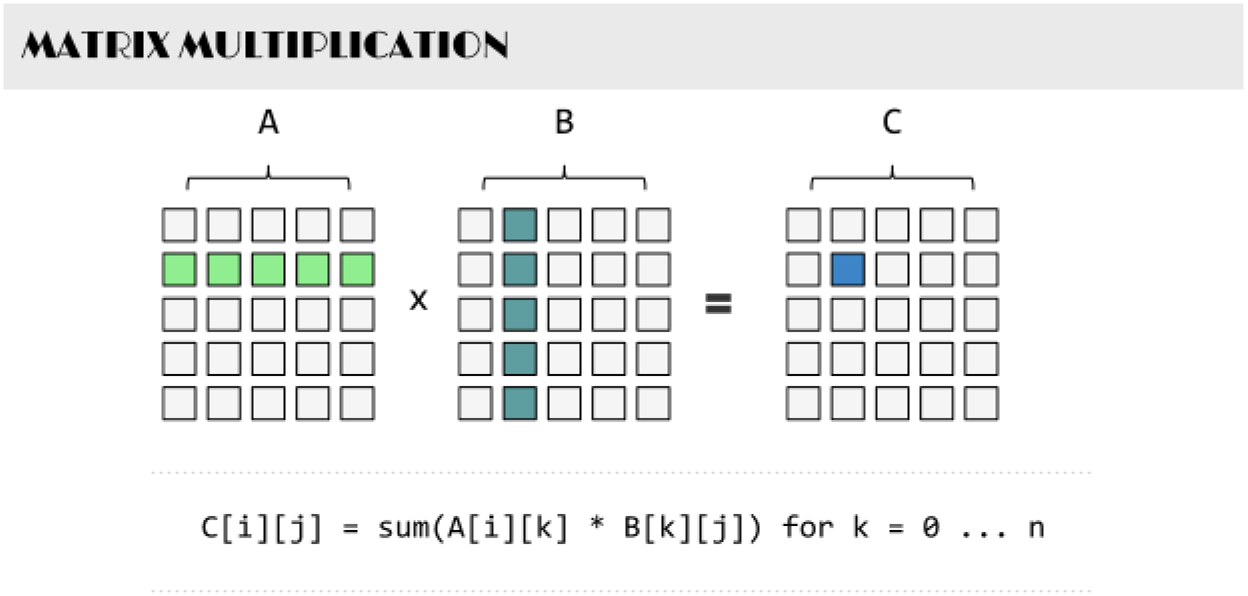
λ-Architecture



K-architecture



Matrix Multiplication



```
C[i][j] = sum(A[i][k] * B[k][j]) for k = 0 ... n
```

Matrix-Vector multiplication

- see section 2.3 at http://www.mmds.org

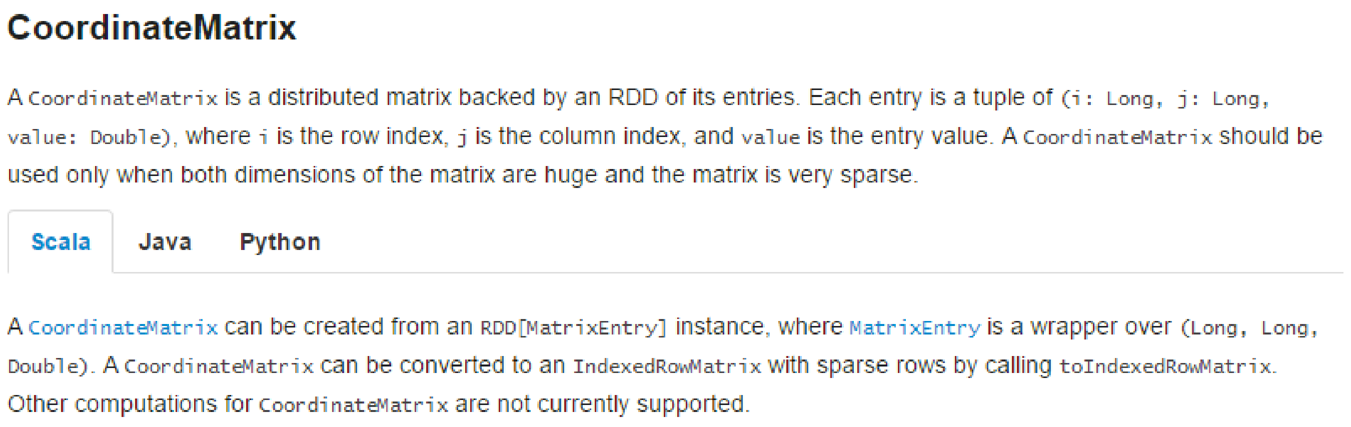
Sparse Matrices

a [large] matrix with a few non-zero elements

- recommendation systems (users vs. products)
- any graph (including web/social)
- …

matrix is represented by set of tuples (i, j, value) (representing non-zero elements)

from Spark documentation:

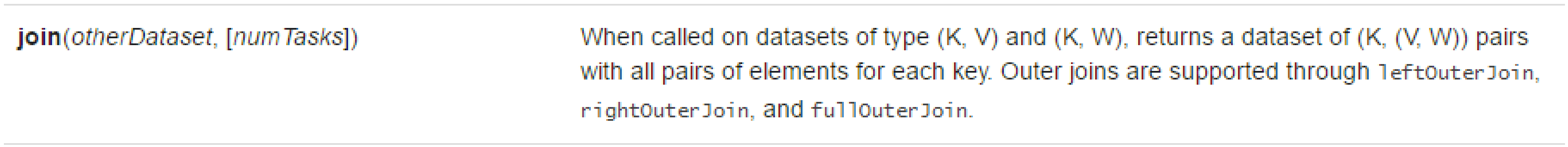


CoordinateMatrix

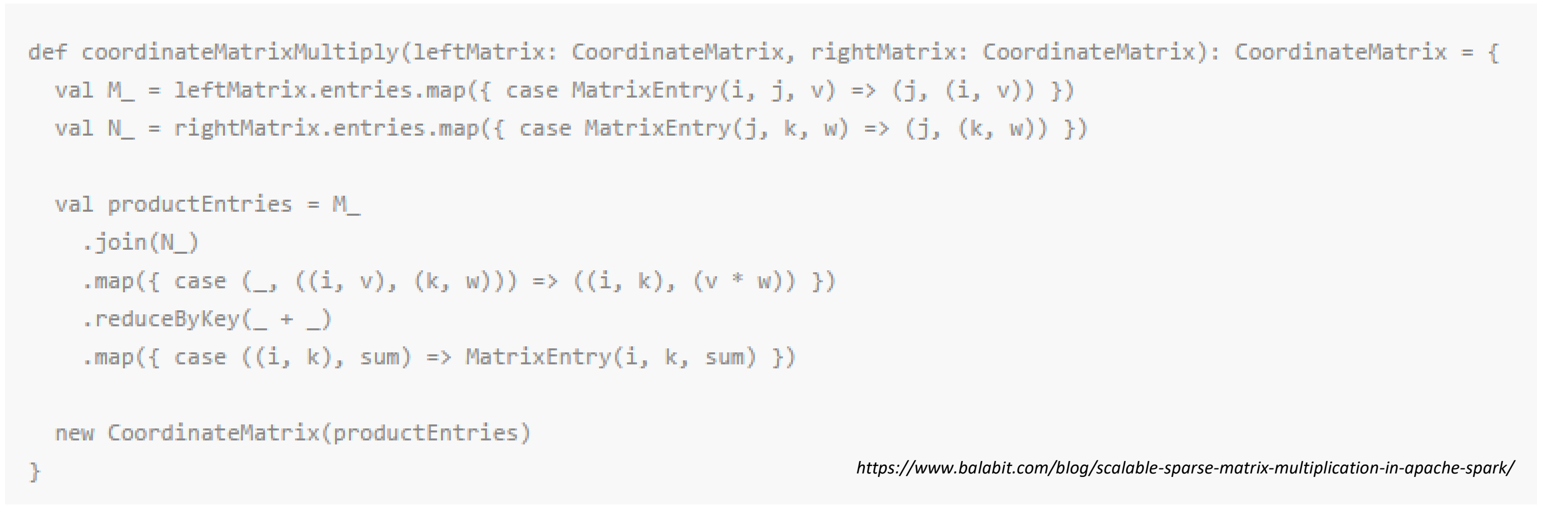
A `CoordinateMatrix` is a distributed matrix backed by an RDD of its entries. Each entry is a tuple of `(i: Long, j: Long, value: Double)`, where `i` is the row index, `j` is the column index, and `value` is the entry value. A `CoordinateMatrix` should be used only when both dimensions of the matrix are huge and the matrix is very sparse.

A `CoordinateMatrix` can be created from an `RDD[MatrixEntry]` instance, where `MatrixEntry` is a wrapper over `(Long, Long, Double)`. A `CoordinateMatrix` can be converted to an `IndexedRowMatrix` with sparse rows by calling `toIndexedRowMatrix`. Other computations for `CoordinateMatrix` are not currently supported.

Sparse Matrix multiplication



join(*otherDataset*, [*numTasks*])	When called on datasets of type (K, V) and (K, W), returns a dataset of (K, (V, W)) pairs with all pairs of elements for each key. Outer joins are supported through `leftOuterJoin`, `rightOuterJoin`, and `fullOuterJoin`.



```
def coordinateMatrixMultiply(leftMatrix: CoordinateMatrix, rightMatrix: CoordinateMatrix): CoordinateMatrix = {
  val M_ = leftMatrix.entries.map({ case MatrixEntry(i, j, v) => (j, (i, v)) })
  val N_ = rightMatrix.entries.map({ case MatrixEntry(j, k, w) => (j, (k, w)) })

  val productEntries = M_
    .join(N_)
    .map({ case (_, ((i, v), (k, w))) => ((i, k), (v * w)) })
    .reduceByKey(_ + _)
    .map({ case ((i, k), sum) => MatrixEntry(i, k, sum) })

  new CoordinateMatrix(productEntries)
}
```

https://www.balabit.com/blog/scalable-sparse-matrix-multiplication-in-apache-spark/

from the comment to the article above:
200 mil x 200 mil and a maximum number of non-zero per row or column of about 2k -> 44 min

Recursive multiplication

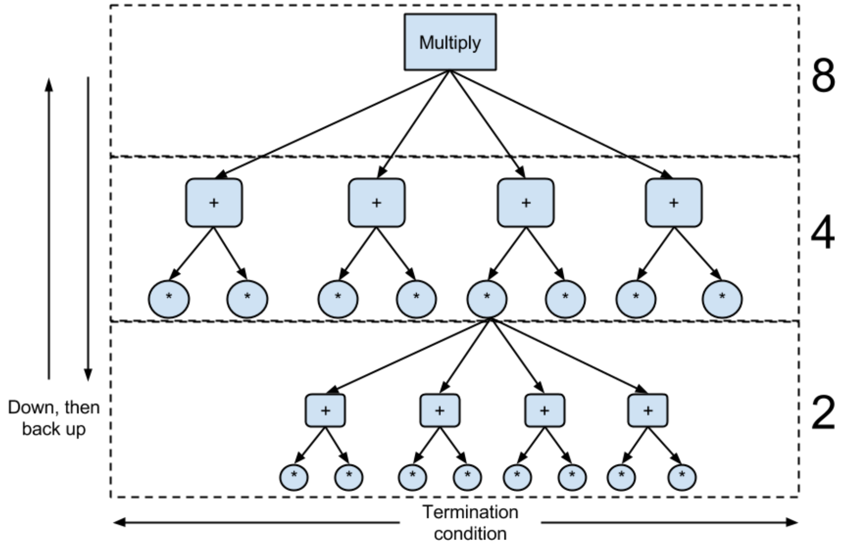
$$\left[\begin{array}{c|c} A11 & A12 \\ \hline A21 & A22 \end{array}\right] \cdot \left[\begin{array}{c|c} B11 & B12 \\ \hline B21 & B22 \end{array}\right] = \left[\begin{array}{c|c} \begin{matrix} A11.B11 \\ + \\ A12.B21 \end{matrix} & \begin{matrix} A11.B12 \\ + \\ A12.B22 \end{matrix} \\ \hline \begin{matrix} A21.B11 \\ + \\ A22.B21 \end{matrix} & \begin{matrix} A21.B12 \\ + \\ A22.B22 \end{matrix} \end{array}\right]$$

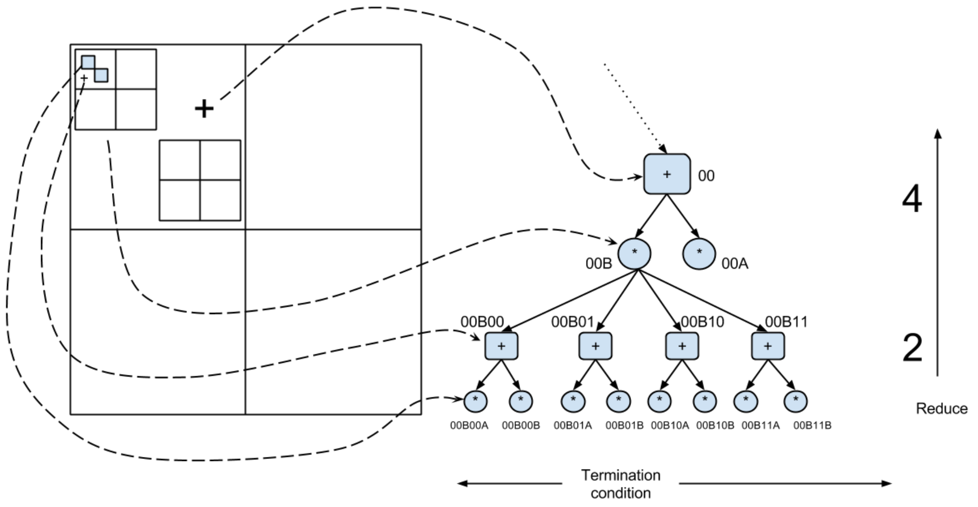
rules for matrix multiplication holds even if elements are matrices themselves

1) If one (or both) submatrix satisfies a "efficient calculation" criterion, calculate the product (hopefully, efficiently). Exit current context.
2) If a submatrix is small enough for its product to be calculated, calculate the product. Exit current context.
2) If both the submatrices is still too big, partition them into 4 partitions.
3) Go to step 1 for each partition.

Recursive multiplication

$$\begin{bmatrix} A11 & A12 \\ A21 & A22 \end{bmatrix} \cdot \begin{bmatrix} B11 & B12 \\ B21 & B22 \end{bmatrix} = \begin{bmatrix} A11.B11 + A12.B21 & A11.B12 + A12.B22 \\ A21.B11 + A22.B21 & A21.B12 + A22.B22 \end{bmatrix}$$





Map/Reduce Patterns: Collating

- **Problem Statement:** There is a set of items and some function of one item. It is required to save all items that have the same value of function into one file or perform some other computation that requires all such items to be processed as a group. The most typical example is building of inverted indexes.
- **Solution:** The solution is straightforward. Mapper computes a given function for each item and emits value of the function as a key and item itself as a value. Reducer obtains all items grouped by function value and process or save them. In case of inverted indexes, items are terms (words) and function is a document ID where the term was found.
- Applications:
  - Inverted Indexes