Bigdata Infrastructure
Part II

F. Toumani

Institut d'Informatique, LIMOS, UCA

October 6, 2022

- Massively parallel programs
  - Simple parallel programming model
  - Scalability
  - Fault tolerance
- Moving programs not data !!

Exercise

You have a collection of 1000 000 000 documents of objects observed in the Sky. The Sky is organized in a set of predefined zones and each object is associated with a given zone. An object description includes a zone attribute and a weight attribute that give respectively a zone of the sky where the object is located and the weight of the object.

- **Problem:** compute the total weights of objects in each zone of the sky using a cluster of 1000 nodes.

Exercise

You have a collection of 1000 000 000 documents of objects observed in the Sky. The Sky is organized in a set of predefined zones and each object is associated with a given zone. An object description includes a zone attribute and a weight attribute that give respectively a zone of the sky where the object is located and the weight of the object.

- **Problem:** compute the total weights of objects in each zone of the sky using a cluster of 1000 nodes.
- The node's failure rate is around 20%. What is the impact on your program?

Exercise

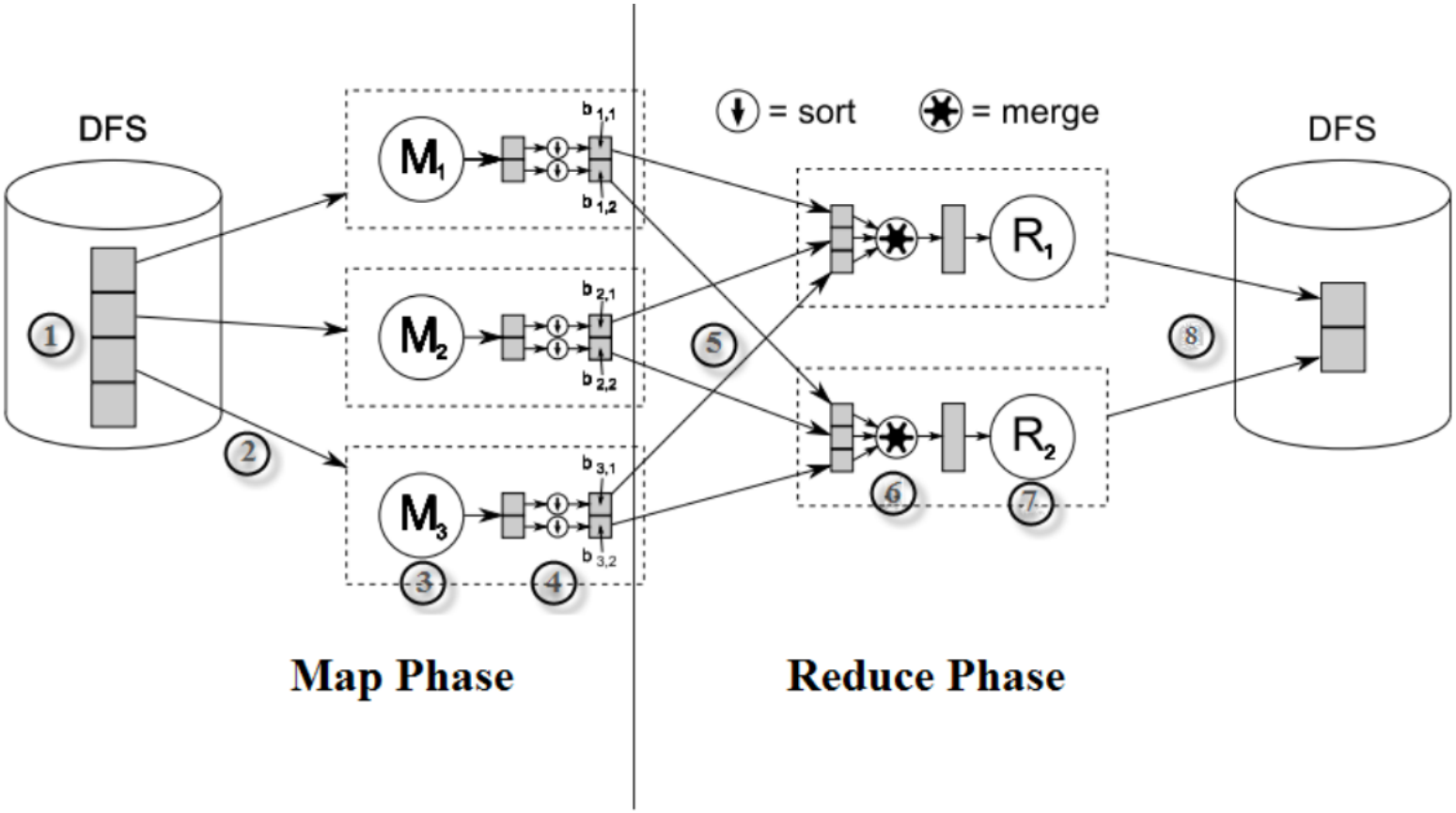
You have a collection of 1000 000 000 documents of objects observed in the Sky. The Sky is organized in a set of predefined zones and each object is associated with a given zone. An object description includes a zone attribute and a weight attribute that give respectively a zone of the sky where the object is located and the weight of the object.

- **Problem:** compute the total weights of objects in each zone of the sky using a cluster of 1000 nodes.
- The node's failure rate is around 20%. What is the impact on your program?
- When you come to execute your program, you observe that there are 1000 additional nodes which are available in the cluster. How to adapt your program to run it in this new context?

MapReduce framework

- A process in two steps
  - A Map step : execution of a user provided map function
  - A Reduce step : execution of a user provided reduce function
- Main advantages/drawbacks
  - (+) Make parallelism transparent to the programmer
    - Muliple instances of the map function are executed in parallel
    - Muliple instances of the reduce function are executed in parallel
  - (+) Massively parallel model
    - Fault tolerance: failures have *local effects*
    - Horizontal scalability
  - (-) Suitable only for simple problems (highly parallelizable)
  - (-) Initialization cost

MapReduce framework



- the input file is splitted into several pieces (a split or chunk)
- each node hosting a map task reads the content of the corresponding input split from the distributed file system
- each mapper converts the content of its input split into a sequence of key-value pairs and calls the user-defined Map function for each (k, v) pair. The produced intermediate pairs (k', v') are buffered in memory.
- periodically, the buffered intermediate key-value pairs are written to r local intermediate files, called segment files, where r is the number of reducer nodes. The partitioning function ensures that pairs with the same key are always allocated to the same segment file.

- on the completion of a map task, the reducers (i.e., nodes executing the reduce function), will pull over their corresponding segments.
- when a reducer has read all intermediate data, it sorts it by the intermediate keys so that all occurrences of the same key are grouped together. If the amount of intermediate data is too large to fit in memory, an external sort is used. The reducer then merges the data to produce for each intermediate key k' a single pair (k', list(v'))
- each reducer iterates over the sorted intermediate data and passes each pair (k', list(v')) to the user's reduce function.
- each reducer writes its final results to the distributed file system.

Design of MapReduce programs

- Map function
  - tuple at a time funcion
  - returns <key, value> pairs
- Reduce function
  - Takes as input a paire <key, list(values)>

- Map function

```
var mapFunction1 = function() {
emit(this.country, 1); };
```

- Reduce function

```
var reduceFunction1 = function(k, v) {
return Array.sum(v); };
```

- Call of a MapReduce program

```
db.orders.mapReduce(
        mapFunction1,
        reduceFunction1,
        { out: "my-map-reduce-result" } )
```

MongoDB MapReduce

```
db.runCommand( {
    mapReduce: <collection>,
    map: <function>,
    reduce: <function>,
    finalize: <function>,
    out: <output>,
    query: <document>,
    sort: <document>,
    limit: <number>,
    scope: <document>,
    jsMode: <boolean>,
    verbose: <boolean>,
    bypassDocumentValidation: <boolean>,
    collation: <document> } )
```

{ _id: ObjectId("50a8240b927d5d8b5891743c"),
cust_id: "abc123",
ord_date: new Date("Oct 04, 2012"),
status: 'A',
price: 25,
items: [{ sku: "mmm", qty: 5, price: 2.5 },
{ sku: "nnn", qty: 5, price: 2.5 }] }

- Return the Total Price Per Customer
- Calculate Order and Total Quantity with Average Quantity Per Item

- Return the Total Price Per Customer

```
db.orders.mapReduce(
    function() {emit(this.cust_id, this.price);},
    function(keyCustId, valuesPrices)
        {return Array.sum(valuesPrices)},
    { out: "myResult" } )
```

- Calculate Order and Total Quantity with Average Quantity Per Item

```
var mapFunction2 = function() {
   for (var idx = 0; idx < this.items.length; idx++) {
         var key = this.items[idx].sku;
         var value = {
                  count: 1,
                  qty: this.items[idx].qty };
         emit(key, value); } };
```

```
var reduceFunction2 = function(keySKU, countObjVals) {
    reducedVal = { count: 0, qty: 0 };
    for (var idx = 0; idx < countObjVals.length; idx++) {
        reducedVal.count += countObjVals[idx].count;
        reducedVal.qty += countObjVals[idx].qty; }
    return reducedVal; };

var finalizeFunction2 = function (key, reducedVal) {
    reducedVal.avg = reducedVal.qty/reducedVal.count;
    return reducedVal; };
```

```
db.orders.mapReduce( mapFunction2,
      reduceFunction2,
      {
            out: { merge: "map_reduce_example" },
            query: { ord_date:
                  { $gt: new Date('01/01/2012')}
            },
      finalize: finalizeFunction2
} )
```

- In the map function, reference the current document as **this** within the function
- The map function should not access the database for any reason
- The map function should not have side effects
- The map function may optionally call **emit(key,value)** any number of times to create an output document associating key with value

- The reduce function should not access the database
- The reduce function should not have side effects
- The type of the return object must be identical to the type of the value emitted by the map function (multiple invocation of the reduce function for the same key)
- The reduce function must be associative
- The reduce function must be idempotent
- The reduce function should be commutative

Aggregation pipeline

- A multi-stage pipeline that transforms the documents into an aggregated result

$\Rightarrow$ Preferred solution for aggregation tasks

```
db.orders.aggregate([
{ $match: { status: "A" } },
{ $group: {_id: "$cust_id", total: {$sum: "$amount" } } },
{ $sort: { total: -1 } } ])
```

- Stage operators
  - $project, $match, $sort, $sortBycount, $group, $count
  - $limit , $skip
  - $geoNear
  - $redact : restricts the contents of the documents based on information stored in the documents themselves (using the system variables $$DESCEND, $$PRUNE, or $$KEEP)
  - $lookup: performs a left outer join to another collection in the same database to filter in documents from the "joined" collection for processing
  - $unwind: deconstructs an array field from the input documents to output a document for each element
  - ...
- boolean operators: $and, $or, $not
- other operators: arithmetic, comparaison, set, ...

System Variable	Description
$$DESCEND	$redact returns the fields at the current document level, excluding embedded documents. To include embedded documents and embedded documents within arrays, apply the $cond expression to the embedded documents to determine access for these embedded documents
$$PRUNE	$redact excludes all fields at this current document/embedded document level, without further inspection of any of the excluded fields. This applies even if the excluded field contains embedded documents that may have different access levels
$$KEEP	$redact returns or keeps all fields at this current document/embedded document level, without further inspection of the fields at this level. This applies even if the included field contains embedded documents that may have different access levels

db.books.aggregate([{$project : { title : 1 , author : 1 } }])

db.books.aggregate([{ $project : { "author.first" : 0,
"lastModified" : 0 } }])

{ "_id": 1,"item" :"ABC1", sizes: ["S", "M", "L"] }

db.inventory.aggregate([{ $unwind : "$sizes" }])

{ "_id" : 1, "item" : "ABC1", "sizes" : "S" }
{ "_id" : 1, "item" : "ABC1", "sizes" : "M" }
{ "_id" : 1, "item" : "ABC1", "sizes" : "L" }

Stage operators
Examples

```
{
 \_id: 1,
 title: "123 Department Report",
 tags: [ "G", "STLW" ],
 year: 2014,
 subsections: [
  {
   subtitle: "Section 1: Overview",
   tags: [ "SI", "G" ],
   content:  "Section 1: content of section 1. »   },
  {
   subtitle: "Section 2: Analysis",
   tags: [ "STLW" ],
   content: "Section 2: content of section 2."
  },
  {
   subtitle: "Section 3: Budgeting",
   tags: [ "TK" ],
   content: {
    text: "Section 3: content of section3.",
    tags: [ "HCS" ]     }    }
 ]
}
```

```
{
 "_id" : 1,
 "title" : "123 Department Report",
 "tags" : [ "G", "STLW" ],
 "year" : 2014,
 "subsections" : [
  {
   "subtitle" : "Section 1: Overview",
   "tags" : [ "SI", "G" ],
   "content" : "Section 1: content of section 1."
  },
  {
   "subtitle" : "Section 2: Analysis",
   "tags" : [ "STLW" ],
   "content" : "Section 2: content of section 2."
  }
 ]
}
```

userAccess : "STLW" or "G"

```
db.forecasts.aggregate( [
   { $match: { year: 2014 } },
   { $redact: {
      $cond: {
      if: { $gt: [ { $size: { $setIntersection:
            [ "$tags", userAccess ] } }, 0 ] },
         then: "$$DESCEND",
         else: "$$PRUNE"
} } } ] );
```

Stage operators
Examples

```
{
"_id" : 1,
"title" : "123 Department Report",
"tags" : [ "G", "STLW" ],
"year" : 2014,
"subsections" : [
   {
      "subtitle" : "Section 1: Overview",
      "tags" : [ "SI", "G" ],
      "content" : "Section 1: This is the content of section 1."
   },
   {
      "subtitle" : "Section 2: Analysis",
      "tags" : [ "STLW" ],
      "content" : "Section 2: This is the content of section 2."
   }
   ] }
```

Stage operators
Examples

Orders

{ "_id" : 1, "item" : "abc", "price" : 12, "quantity" : 2 }

Inventory

{ "_id" : 1, "sku" : "abc", description: "product 1",
"instock" : 120 }

```
db.orders.aggregate([
   {
      $lookup:
         {
            from: "inventory",
            localField: "item",
            foreignField: "sku",
            as: "inventory_docs"
         }
   } ])
```

Expression of operators

Expression of operators

{ <operator>: [<argument1>, <argument2> ...] }

{ <operator>: <argument>}

```
db.orders.aggregate([
{ $match: { status: "A" } },
{ $group: {_id: "$cust_id", total: {$sum: "$amount" } } },
{ $sort: { total: -1 } } ])
```

- $sum, $avg, $first, $last, $max, $min
- $stdDevPop: Returns the population standard deviation of the input values
- $stdDevSamp: Returns the sample standard deviation of the input values

- when used in the $group stage, maintain their state (e.g. totals, maximums, minimums, and related data) as documents progress through the pipeline
- When used in the $project stage, the accumulators do not maintain their state.

```
{
    _id : "jane",
    joined : ISODate("2011-03-02"),
    likes : ["golf", "racquetball"]
}
```

- Return user names in upper case and in alphabetical order
- Return Usernames Ordered by Join Month
- Return Total Number of Joins per Month
- Return the Five Most Common "Likes"

Return user names in upper case and in alphabetical order

```
db.users.aggregate(
   [
      { $project : { name:{$toUpper:"$_id"} , _id:0 } },
      { $sort : { name : 1 } }
   ]
)
```

Return Usernames Ordered by Join Month

```
db.users.aggregate(
  [
    { $project :
      {
        month_joined : { $month : "$joined" },
        name : "$_id",
        _id : 0
      }
    },
    { $sort : { month_joined : 1 } }
  ] )
```

Return Total Number of Joins per Month

```
db.users.aggregate(
   [
      {
         $project : { month_joined : { $month : "$joined" } }
      } ,
      {
         $group : { _id : {month_joined:"$month_joined"} ,
         number : { $sum : 1 } }
      },
      { $sort : { "_id.month_joined" : 1 } }
   ]
)
```

Return the Five Most Common "Likes"

```
db.users.aggregate(
   [       { $unwind : "$likes" },
      { $group : { _id : "$likes" , number : { $sum : 1 } } },
      { $sort : { number : -1 } },
      { $limit : 5 }
   ]
)
```

Aggregation Pipeline Optimization

- Projection

  If only a subset of the fields in the documents are required, the aggregation pipeline will only use those required fields
- Reordering the sequences
  - $sort + $match
  - $skip + $limit
  - $redact + $match

    When possible, when the pipeline has the $redact stage immediately followed by the $match stage, the aggregation can sometimes add a portion of the $match stage before the $redact stage
  - $project + $skip or $limit

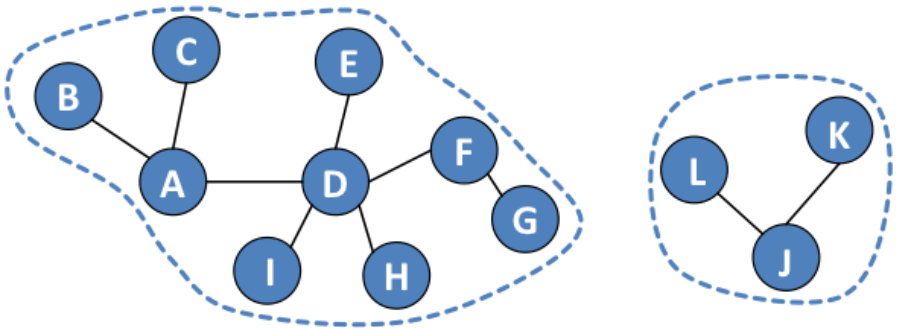
- Pipeline Coalescence Optimization
  - $sort + $limit Coalescence
  - $limit + $limit Coalescence
  - $skip + $skip Coalescence
  - $match + $match Coalescence
  - $lookup + $unwind Coalescence
  - Coalescence

Exercise

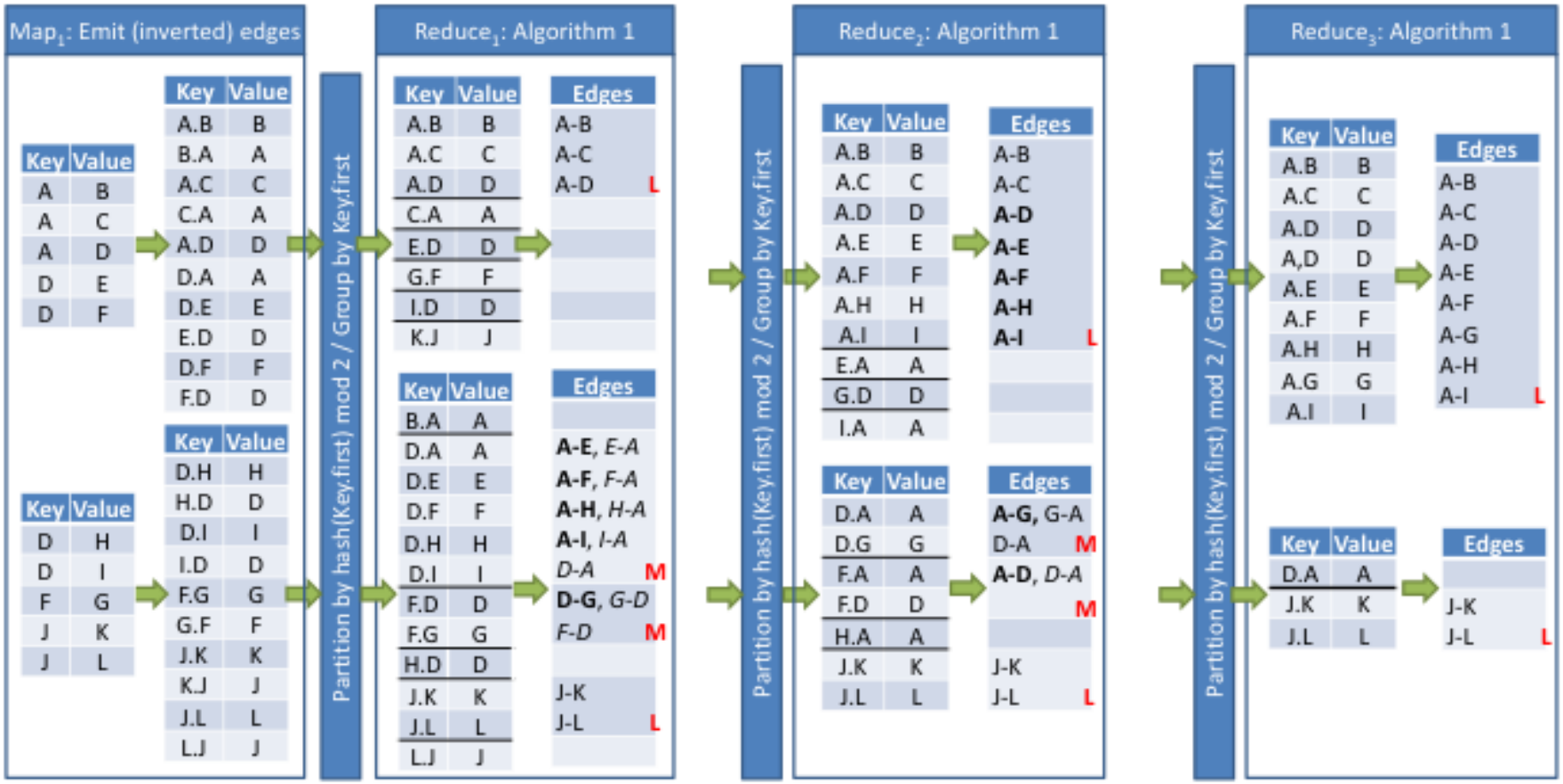
Consider a social network graph where individuals are represented as nodes and friendship relationships are represented as edges. Write a MapReduce program to compute the groups of nodes that are related to each other by the friendship relationship.

Exercise

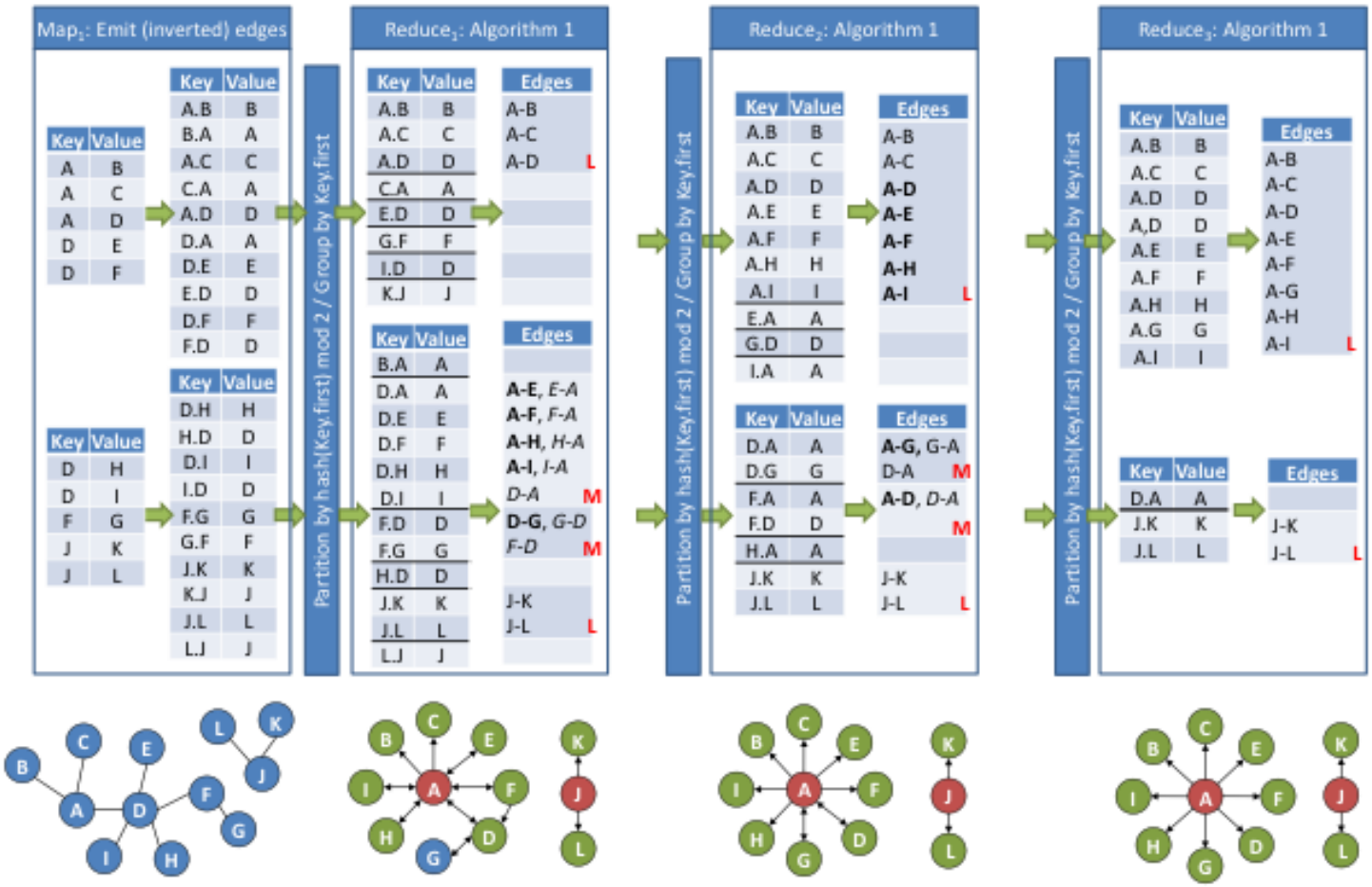
Consider a social network graph where individuals are represented as nodes and friendship relationships are represented as edges. Write a MapReduce program to compute the groups of nodes that are related to each other by the friendship relationship.



Exercise


Map1: Emit (inverted) edges
Key Value
A B
A C
A D
D E
D F
Key Value
A.B B
B.A A
A.C C
C.A A
A.D D
D.A A
D.E E
E.D D
D.F F
F.D D
Key Value
D H
D I
F G
J K
J L
Key Value
D.H H
H.D D
D.I I
I.D D
F.G G
G.F F
J.K K
K.J J
J.L L
L.J J
Partition by hash(Key.first) mod 2 / Group by Key.first
Reduce1: Algorithm 1
Key Value
A.B B
A.C C
A.D D
C.A A
E.D D
G.F F
I.D D
K.J J
Edges
A-B
A-C
A-D L
Key Value
B.A A
D.A A
D.E E
D.F F
D.H H
D.I I
F.D D
F.G G
H.D D
J.K K
J.L L
L.J J
Edges
A-E, E-A
A-F, F-A
A-H, H-A
A-I, I-A
D-A M
D-G, G-D
F-D M
J-K
J-L L
Partition by hash(Key.first) mod 2 / Group by Key.first
Reduce2: Algorithm 1
Key Value
A.B B
A.C C
A.D D
A.E E
A.F F
A.H H
A.I I
E.A A
G.D D
I.A A
Edges
A-B
A-C
A-D
A-E
A-F
A-H
A-I L
Key Value
D.A A
D.G G
F.A A
F.D D
H.A A
J.K K
J.L L
Edges
A-G, G-A
D-A M
A-D, D-A
M
J-K
J-L L
Partition by hash(Key.first) mod 2 / Group by Key.first
Reduce3: Algorithm 1
Key Value
A.B B
A.C C
A.D D
A,D D
A.E E
A.F F
A.H H
A.G G
A.I I
Edges
A-B
A-C
A-D
A-E
A-F
A-G
A-H
A-I L
Key Value
D.A A
J.K K
J.L L
Edges
J-K
J-L L

Exercise


Map$_1$: Emit (inverted) edges
Reduce$_1$: Algorithm 1
Reduce$_2$: Algorithm 1
Reduce$_3$: Algorithm 1
Partition by hash(Key.first) mod 2 / Group by Key.first
Partition by hash(Key.first) mod 2 / Group by Key.first
Partition by hash(Key.first) mod 2 / Group by Key.first
Key Value
Edges
A-E, E-A
A-F, F-A
A-H, H-A
A-I, I-A
D-A M
D-G, G-D
F-D M
A-G, G-A
D-A M
A-D, D-A M
J-K
J-L L

Beyond MapReduce

- Need to cover
  - More complex, multi-stages applications
  - Interactive ad-hoc queries
- Limitations of preexisting technology
  - Lack of abstractions for leveraging **distributed memory**
  - Reusing intermediate results across **multiple computations**
- Notion of RDD (Resilient Distributed Data)
  - High level operators
  - Distributed execution based on MapReduce
  - Notion of **RDD** (Resilient Distributed Datasets)

Designing RDD

- A need for a programming interface that can provide fault tolerance efficiently
- Example of abstractions for in-memory storage on clusters: distributed shared memory, key-value stores, databases, ...
  - Interface based on fine-grained updates to mutable state (e.g., cells in a table)
  - Fault tolerance achieved using replication of data across machines or logging updates across machines

    $\Rightarrow$ Expensive for data-intensive workloads

RDD: Resilient Distributed Data

- Fault-tolerant, parallel data structures
- Enable users to explicitly control:
  - Persistence: users define explicit storage strategy (e.g., in-memory storage)
  - Partitioning: user can control RDD partitioning to optimize data placement
- User can manipulate them using a rich set of operators
- An interface based on coarse-grained transformations (e.g., map, filter and join) that apply the same operation to many data items
  - $\Rightarrow$ Fault tolerance achieved by lineage: logging the transformations used to build a dataset (its lineage) rather than the actual data just that partition
  - $\Rightarrow$ Lost data can be recovered without requiring costly replication

RDD: example

```
val data = Array(1, 2, 3, 4, 5)
val myRdd = sc.parallelize(data)        - - sc.parallelize(data, 30)
myRdd.getNumPartitions

val myrddtext = sc.textFile("iliad.mb.txt")
print(myrddtext.getNumPartitions)
val myrddtext2= myrddtext.repartition(4)
print(myrddtext2.getNumPartitions)

val lLengths = myrddtext.map(line => line.length)
val totalLength = lLengths.reduce((x, y) => x + y)

val lLengths = myrddtext.map(line => line.length)
lLengths.persist()                      - - lLengths.cache()
val totalLength = lLengths.reduce((x, y) => x + y)
```

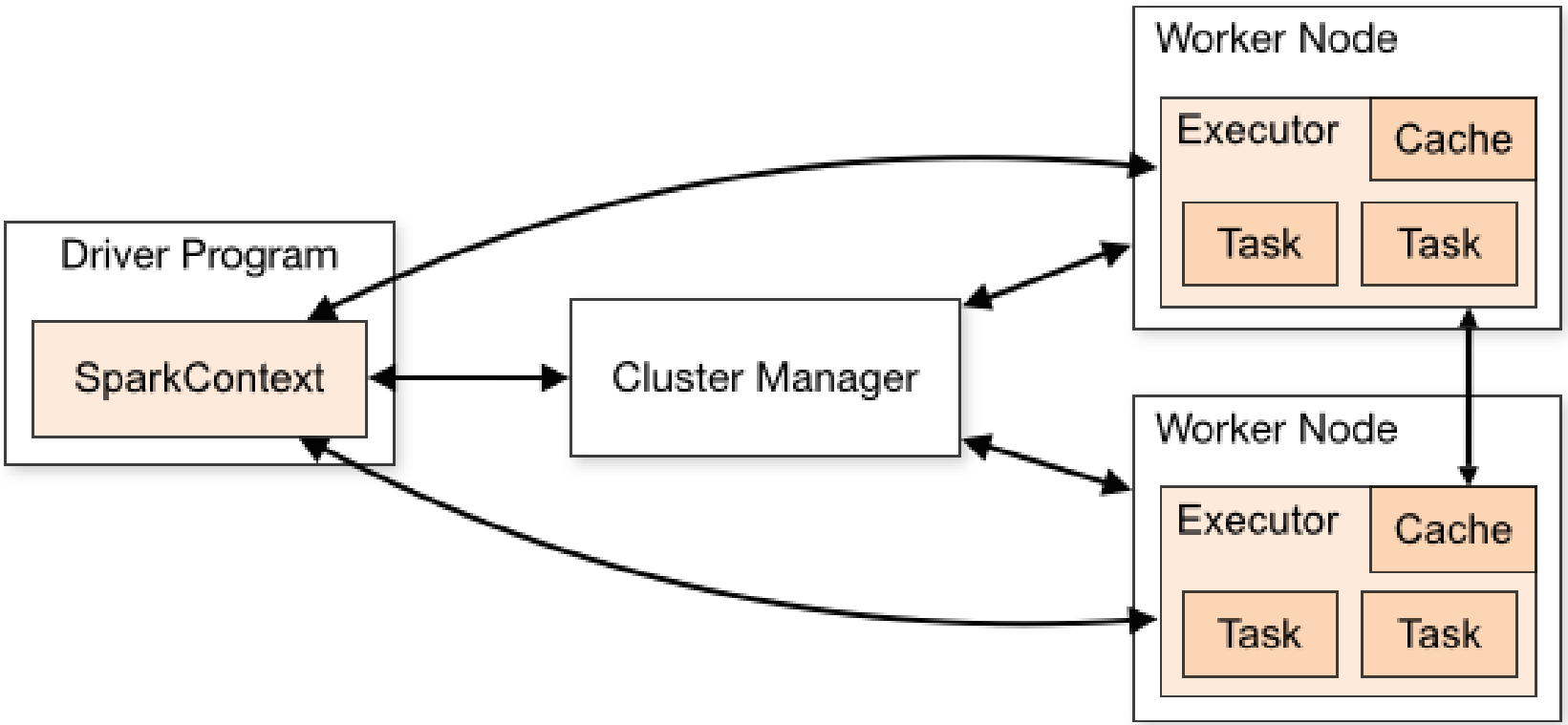
RDD abstraction

- A read-only collection of objects partioned across a set of machines
- Can only be created through deterministic operations on either data in stable storage or other RDDs
- Can be cached in memory across machine
- Can be reused in many MapReduce-like operation
- Fault tolerance based on *lineage*: can be rebuilt if a partition is lost
- Lazy computation of RDDs
- Implemented in the system Apache Spark

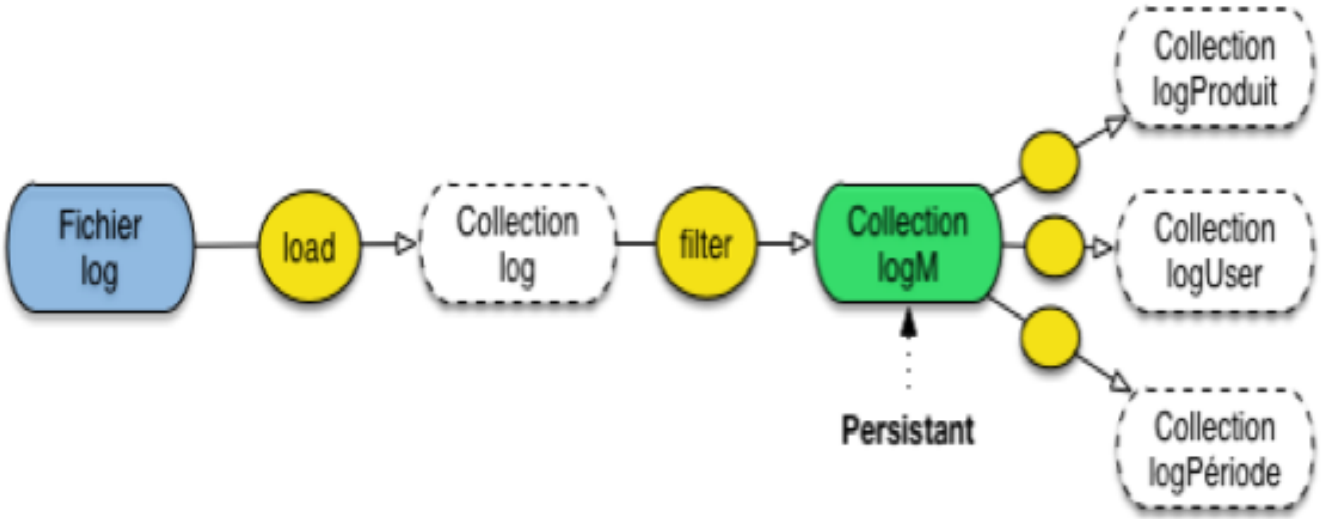
Apache Spark architecture

Spark applications run as independent sets of processes on a cluster coordinated by SparkContext objects

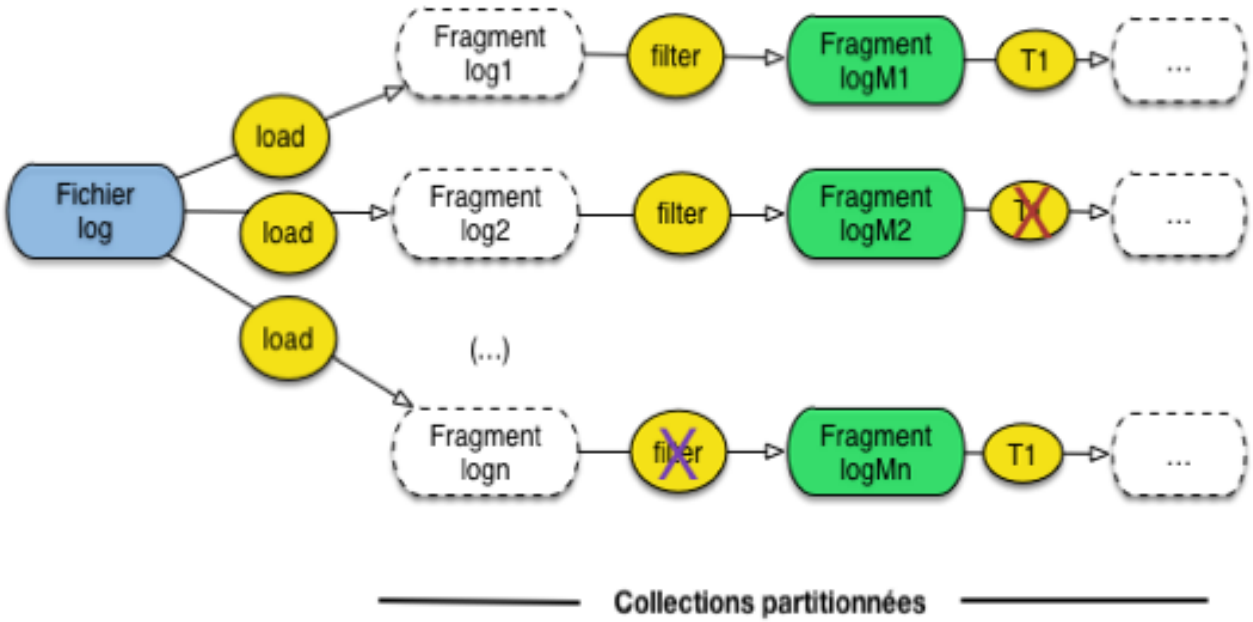
1. Spark acquires executors (processes) on nodes in the cluster
2. Spark sends application code (defined by JAR or Python files passed to SparkContext) to the executors
3. SparkContext sends tasks to the executors to run



A Spark workflow



A Spark workflow



Spark API

- Transformations: operations that enable to create RDDs
- Actions: operations that return a value to the application or export data to a storage system
  - Example : count, collect, save
  - Spark computes RDDs lazily the first time they are used in an action, so that it can pipeline transformations
- A persist method to indicate which RDDs can be reused in future operations
  - Users can define the persistence strategies
  - Users can set a persistence priority on each RDD to specify which in-memory data should spill to disk first

Example of transformations operations

Transformation	**Meaning**
map(func)	Return a new distributed dataset formed by passing each element of the source through a function func
filter(func)	Return a new dataset formed by selecting those elements of the source on which func returns true
flatMap(func)	Similar to map, but each input item can be mapped to 0 or more output items
mapPartitions(func)	Similar to map, but runs separately on each partition (block) of the RDD
union(Dataset)	Return a new dataset that contains the union of the elements in the source dataset and the argument
intersection(Dataset)	Return a new RDD that contains the intersection of elements in the source dataset and the argument

Example of transformations operations (cont.)

Transformation	Meaning
distinct([numPartitions]))	Return a new dataset that contains the distinct elements of the source dataset
groupByKey([numPartitions])	When called on a dataset of (K, V) pairs, returns a dataset of (K, Iterable<V>) pairs
reduceByKey(func, [numPartitions])	When called on a dataset of (K, V) pairs, returns a dataset of (K, V) pairs where the values for each key are aggregated using the given reduce function func, which must be of type (V,V) => V.
aggregateByKey(zeroValue) (seqOp, combOp, [numPartitions])	When called on a dataset of (K, V) pairs, returns a dataset of (K, U) pairs where the values for each key are aggregated using the given combine functions and a neutral "zero" value.
sortByKey([ascending], [numPartitions])	When called on a dataset of (K, V) pairs where K implements Ordered, returns a dataset of (K, V) pairs sorted by keys in ascending or descending order, as specified in the boolean ascending argument
join(otherDataset, [numPartitions])	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.
cogroup(otherDataset, [numPartitions])	When called on datasets of type (K, V) and (K, W), returns a dataset of (K, (Iterable<V>, Iterable<W>)) tuples. This operation is also called groupWith.

Example of actions operations

Action	Meaning
reduce(func)	Aggregate the elements of the dataset using a function func (which takes two arguments and returns one). The function should be commutative and associative so that it can be computed correctly in parallel
collect()	Return all the elements of the dataset as an array at the driver program. This is usually useful after a filter or other operation that returns a sufficiently small subset of the data
count()	Return the number of elements in the dataset
first()	Return the first element of the dataset (similar to take(1))
take(n)	Return an array with the first n elements of the dataset
takeSample(withReplacement, num, [seed])	Return an array with a random sample of num elements of the dataset, with or without replacement, optionally pre-specifying a random number generator seed
takeOrdered(n, [ordering])	Return the first n elements of the RDD using either their natural order or a custom comparator

Action	Meaning
saveAsTextFile(path)	Write the elements of the dataset as a text file (or set of text files) in a given directory in the local filesystem, HDFS or any other Hadoop-supported file system. Spark will call toString on each element to convert it to a line of text in the file
saveAsSequenceFile(path)	Write the elements of the dataset as a Hadoop SequenceFile in a given path in the local filesystem, HDFS or any other Hadoop-supported file system. This is available on RDDs of key-value pairs that implement Hadoop's Writable interface. In Scala, it is also available on types that are implicitly convertible to Writable (Spark includes conversions for basic types like Int, Double, String, etc)
saveAsObjectFile(path)	Write the elements of the dataset in a simple format using Java serialization, which can then be loaded using SparkContext.objectFile()
countByKey()	Only available on RDDs of type (K, V). Returns a hashmap of (K, Int) pairs with the count of each key
foreach(func)	Run a function func on each element of the dataset. This is usually done for side effects such as updating an Accumulator or interacting with external storage systems

Shuffle operation

- **Re-distributing** data across partitions
- Complex and costly operation
- Operations which can cause a shuffle
  - Repartition operations: repartition and coalesce
  - ByKey operations (except for counting): groupByKey and reduceByKey
  - Join operations: cogroup and join.

```
val mytexte = sc.textFile("Mondocument.txt")
val counts = mytexte.flatMap(line => line.split(" "))
          .map(w => (w, 1))
          .reduceByKey(_ + _)
```

RDD persistence

Storage Level	Meaning
MEMORY_ONLY	Store RDD as deserialized Java objects in the JVM. If the RDD does not fit in memory, some partitions will not be cached and will be recomputed on the fly each time they're needed. This is the default level
MEMORY_AND_DISK	Store RDD as deserialized Java objects in the JVM. If the RDD does not fit in memory, store the partitions that don't fit on disk, and read them from there when they're needed
MEMORY_ONLY_SER	Store RDD as serialized Java objects (one byte array per partition). This is generally more space-efficient than deserialized objects, especially when using a fast serializer, but more CPU-intensive to read
MEMORY_AND_DISK_SER	Similar to MEMORY_ONLY_SER, but spill partitions that don't fit in memory to disk instead of recomputing them on the fly each time they're needed
DISK_ONLY	Store the RDD partitions only on disk
MEMORY_ONLY_2, MEMORY_AND_DISK_2, etc.	Same as the levels above, but replicate each partition on two cluster nodes

Examples of Spark programs

```
 val lines = sc.textFile("data.txt")
val lineLengths = lines.map(s => s.length)
val totalLength = lineLengths.reduce((a, b) => a + b)
val wordsDataset = sc.parallelize(Seq("Spark I am your father", "May the
spark be with you", "Spark I am your father")).toDS()
val groupedDataset = wordsDataset.flatMap(_.toLowerCase.split(" "))
                  .filter(_ != "")
                  .groupBy("value")
val countsDataset = groupedDataset.count()
countsDataset.show()
 val txt = sc.textFile("data.txt")
val wordCounts = textFile.flatMap(line ⇒ line.split(" "))
                         .groupByKey(identity).count()
wordCounts.collect()
```

Parallel operations

- Reduce
- Collect
- Foreach
- ...

Shared variables

- Function work on separate copies of variables
- Variables copied to each machine
- Updates to the variables on the remote machine are not propagated back to the driver program
- Two kinds of variables
  - Broadcast Variables
    - Keep a read-only variable **cached on each machine** rather than shipping a copy with tasks
    - Spark automatically broadcasts the common data needed by tasks within each stage
      val MyBroadcVar = sc.broadcast(Array(1, 2, 3))
  - Accumulators
    - Variables that are only added to through an associative and commutative operation
      val accum = sc.longAccumulator("My Accumulator")
      sc.parallelize(Array(1, 2, 3, 4)).foreach(x => accum.add(x))

Where are we now?

RDD $\rightarrow$ DataFrames $\rightarrow$ Datasets

- RDD
  - (+) compile-time safe, lazy, based on scala collection API
  - (-) low level, cannot be optimized by Spark, slow on non JVM language such as Python
- A Dataset is a distributed collection of data (uses a specialized Encoder to serialize objects)
- A DataFrame
  - provides higher level abstraction (a DSL)
  - Enables the use of query language (e.g., SQL)
  - Have schema

  is a Dataset organized into named columns (conceptually equivalent to a table).

Datasets: examples

Convert a sequence to a dataset

```
val dataset = Seq(1, 2, 3).toDS()
dataset.show()
case class Person(name: String, age: Int)
val personDS = Seq(Person("Max", 33), Person("Adam", 32),
Person("Muller", 62)).toDS()
personDS.show()
```

Comparison of RDDs with distributed shared memory

Aspect	**RDDs**	**Distr. Shared Mem.**
Reads	Coarse- or fine-grained	Fine-grained
Writes	Coarse-grained	Fine-grained
Consistency	Trivial (immutable)	Up to app/runtime
Fault recovery	Fine-grained and low- overhead using lineage	Requires checkpoints and program rollback
Straggler mitigation	Possible using backup tasks	Difficult
Work placement	Automatic based on data locality	Up to app (runtimes aim for transparency)
Behavior if not enough RAM	Similar to existing data flow systems	Poor performance (swapping?)

Apache Spark API

Spark shell: ./bin/spark-shell