CSE 4/587

Data Intensive Computing

Dr. Eric Mikida
epmikida@buffalo.edu
208 Capen Hall

Day 21
Apache Spark (continued)

Announcements and Feedback

- Project Phase 2 due tonight at 11:59pm
- Project Phase 3 due 11/28 at 11:59pm

Remaining Lecture Road-Map (TENTATIVE)

This week: Spark, spark demo, spark examples, ungraded HW

Next week: Monday will be a workshop day (no attendance), Wed no class

11/28,30: Review ungraded HW, Ethics in Big Data

12/5,7: Course wrap-up, summary, final exam review

References

- **Advanced Analytics with Spark** by S. Ryza, U. Laserson, S. Owen and J. Wills
- **Apache Spark documentation**
  - http://spark.apache.org/
  - http://spark.apache.org/docs/latest/programming-guide.html
- **Pyspark**
  - http://spark.apache.org/docs/latest/api/python/pyspark.html
- **Resilient Distributed Dataset: A Fault-tolerant Abstraction for in-Memory Cluster Computing.** M. Zaharia et al.
  - https://www.usenix.org/system/files/conference/nsdi12/nsdi12-final138.pdf

Challenges

Data cleaning: Majority of the work that goes into analyses lies in pre-processing data

- Munging, fusing, mushing and cleansing
- We need computational methods to clean data and data pipeline certainly should include an important step of “data cleaning” and “feature engineering”.
- Choosing from many features, the relevant features.
- Designing a math model from a 2D array (Ex: page rank)

Challenges

Data cleaning: Majority of the work that goes into analyses lies in pre-processing data

- Munging, fusing, mushing and cleansing
- We need computational methods to clean data and data pipeline certainly should include an important step of “data cleaning” and “feature engineering”.
- Choosing from many features, the relevant features.
- Designing a math model from a 2D array (Ex: page rank)

Need to avoid delays in repeated reading of data

Challenges

Iteration: Iteration is a fundamental part of data science.

- Modeling and analysis require typically multiple passes over the same data
- Machine learning algorithms and statistical procedures like stochastic gradient and expected maximization involve repeated scans to reach convergence
- Choosing the right features, picking the right algorithms, running the right significance tests, finding the right hyperparameters: **all require experimentation**

Challenges

Iteration: Iteration is a fundamental part of data science.

- Modeling and analysis require typically multiple passes over the same data
- Machine learning algorithms and statistical procedures like stochastic gradient and expected maximization involve repeated scans to reach convergence
- Choosing the right features, picking the right algorithms, running the right significance tests, finding the right hyperparameters: **all require experimentation**

Need to avoid delays in repeated reading of data

Challenges

Information updates: The results of data analysis presented and **the application becomes part of the production system**…

- This system must frequently or in real time update itself driven by the availability of new data; ie fraud detection system.

Challenges

Information updates: The results of data analysis presented and **the application becomes part of the production system**…

- This system must frequently or in real time update itself driven by the availability of new data; ie fraud detection system.

How about the existing approaches?

- C++, Java are not good for EDA
- R is slow for large data sets and does not integrate well with production stacks
- Read-Evaluate-Print-Loop (REPL) are good for interaction but not work production

Challenges

Information updates: The results of data analysis presented and **the application becomes part of the production system**…

- This system must frequently or in real time update itself driven by the availability of new data; ie fraud detection system.

How about the existing approaches?

- C++, Java are not good for EDA
- R is slow for large data sets and does not integrate well with production stacks
- Read-Evaluate-Print-Loop (REPL) are good for interaction but not work production

Want a framework that makes modeling easy, but also fits well in production systems

Apache Spark

Apache Spark is an open-source, distributed processing system commonly used for big data workloads.

- Utilizes in-memory caching
- Optimized execution for fast performance
- Supports general batch processing, streaming analytics, machine learning, graph databases, and ad hoc queries

Programming Productivity

Biggest bottleneck in data applications is not CPU, disk, or network but analyst productivity

If only we could collapse the entire pipeline from pre-processing of data to model evaluation into a single programming environment…

Spark transitions seamlessly between exploratory analytics and operational analytics

Word Count in Spark (Python API)

```
text_file = spark.textFile("hdfs://...")

text_file.flatMap(lambda line: line.split())
    .map(lambda word: (word, 1))
    .reduceByKey(lambda a, b: a+b)
```

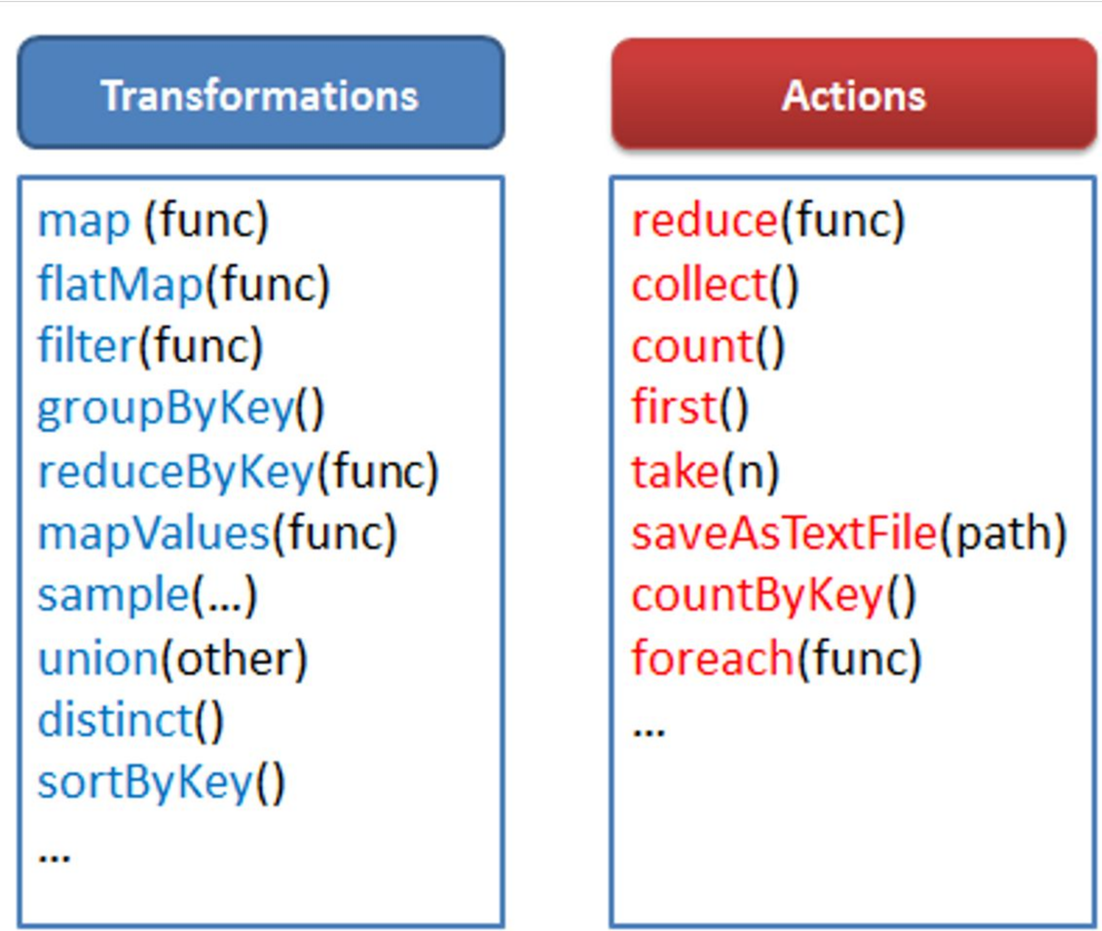
Resilient Distributed Datasets (RDDs)

The building block of the Spark API
(http://spark.apache.org/docs/latest/programming-guide.html#resilient-distributed-datasets-rdds)

In RDD API there are two types of operations:

1. *Transformations* that define a new data set based on previous ones
2. *Actions* which kick off a job to execute on a cluster

RDD Transformations and Actions



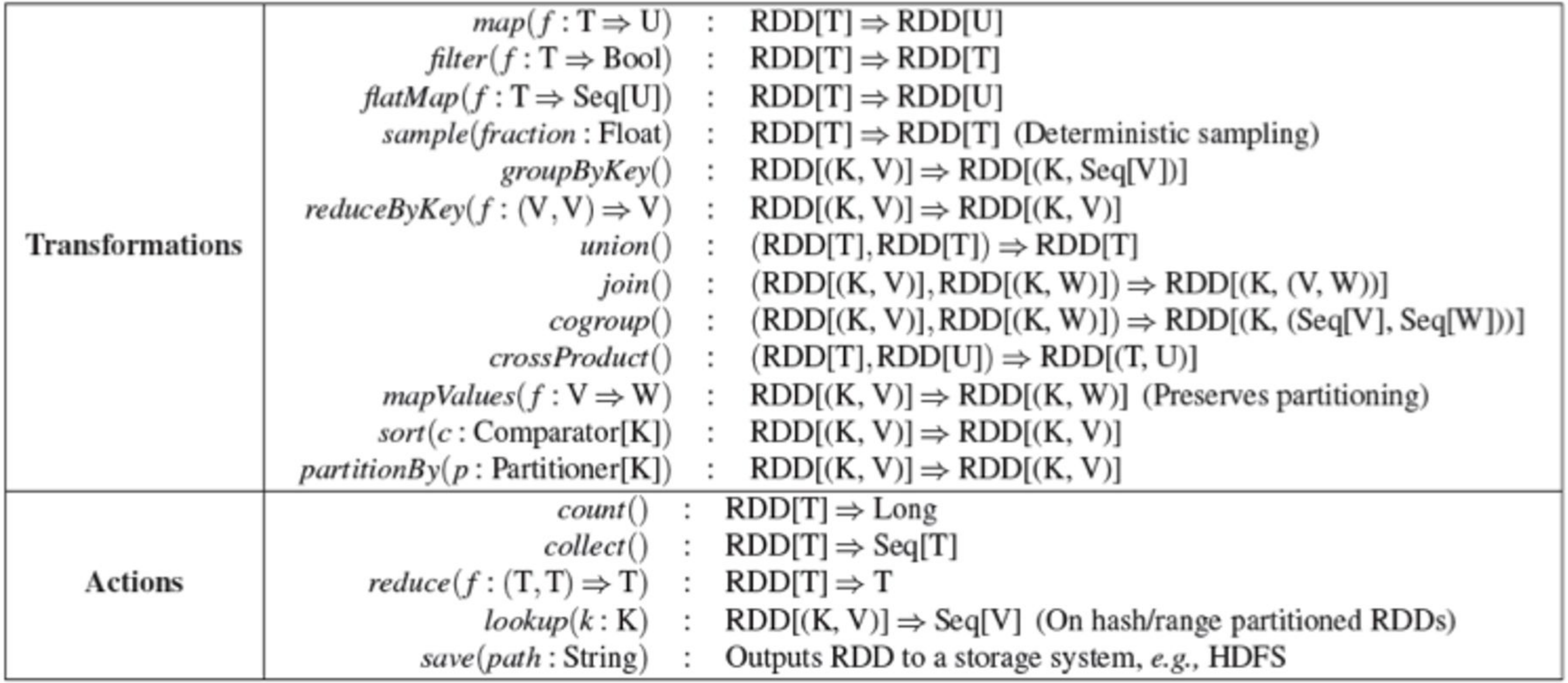
Transformations

map (func)
flatMap(func)
filter(func)
groupByKey()
reduceByKey(func)
mapValues(func)
sample(...)
union(other)
distinct()
sortByKey()
...

Actions

reduce(func)
collect()
count()
first()
take(n)
saveAsTextFile(path)
countByKey()
foreach(func)
...

RDD Transformations and Actions



Transformations	$map(f : \mathrm{T} \Rightarrow \mathrm{U})$	:	RDD[T] ⇒ RDD[U]
	$filter(f : \mathrm{T} \Rightarrow \mathrm{Bool})$	:	RDD[T] ⇒ RDD[T]
	$flatMap(f : \mathrm{T} \Rightarrow \mathrm{Seq[U]})$	:	RDD[T] ⇒ RDD[U]
	$sample(fraction : \mathrm{Float})$	:	RDD[T] ⇒ RDD[T] (Deterministic sampling)
	$groupByKey()$	:	RDD[(K, V)] ⇒ RDD[(K, Seq[V])]
	$reduceByKey(f : (\mathrm{V}, \mathrm{V}) \Rightarrow \mathrm{V})$	:	RDD[(K, V)] ⇒ RDD[(K, V)]
	$union()$	:	(RDD[T], RDD[T]) ⇒ RDD[T]
	$join()$	:	(RDD[(K, V)], RDD[(K, W)]) ⇒ RDD[(K, (V, W))]
	$cogroup()$	:	(RDD[(K, V)], RDD[(K, W)]) ⇒ RDD[(K, (Seq[V], Seq[W]))]
	$crossProduct()$	:	(RDD[T], RDD[U]) ⇒ RDD[(T, U)]
	$mapValues(f : \mathrm{V} \Rightarrow \mathrm{W})$	:	RDD[(K, V)] ⇒ RDD[(K, W)] (Preserves partitioning)
	$sort(c : \mathrm{Comparator[K]})$	:	RDD[(K, V)] ⇒ RDD[(K, V)]
	$partitionBy(p : \mathrm{Partitioner[K]})$	:	RDD[(K, V)] ⇒ RDD[(K, V)]
Actions	$count()$	:	RDD[T] ⇒ Long
	$collect()$	:	RDD[T] ⇒ Seq[T]
	$reduce(f : (\mathrm{T}, \mathrm{T}) \Rightarrow \mathrm{T})$	:	RDD[T] ⇒ T
	$lookup(k : \mathrm{K})$	:	RDD[(K, V)] ⇒ Seq[V] (On hash/range partitioned RDDs)
	$save(path : \mathrm{String})$	:	Outputs RDD to a storage system, *e.g.*, HDFS

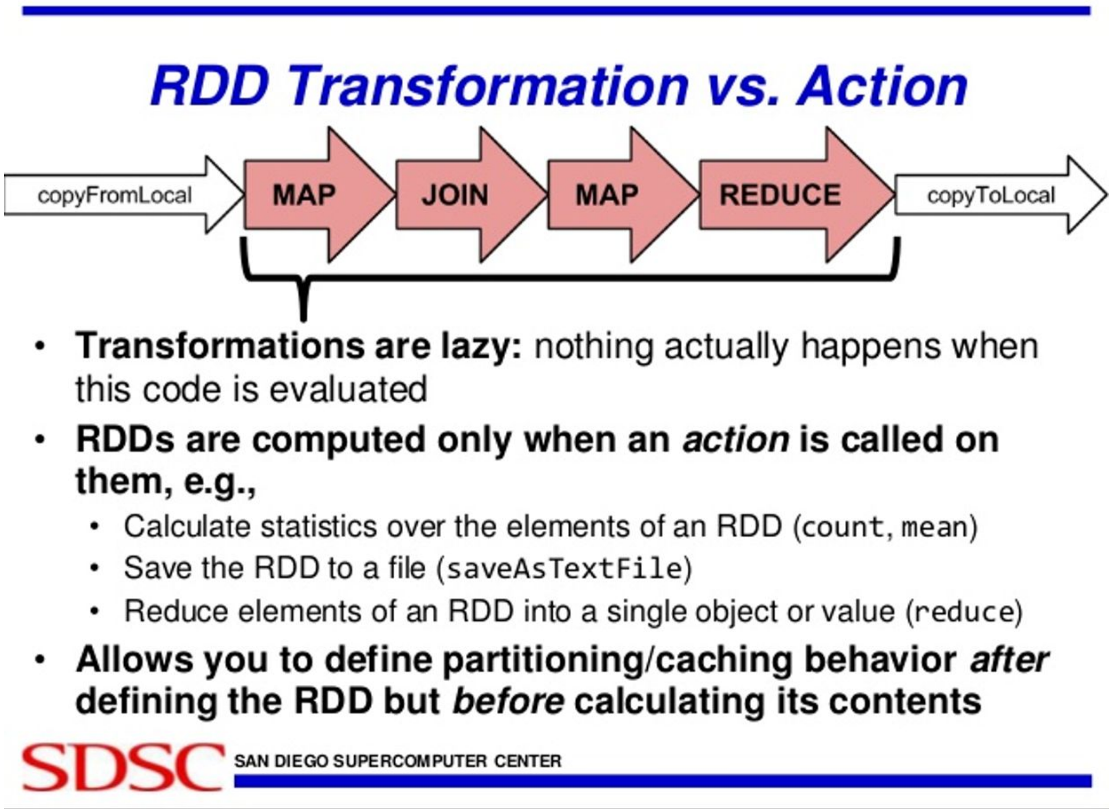
Table 2: Transformations and actions available on RDDs in Spark. Seq[T] denotes a sequence of elements of type T.

Resilient Distributed Datasets (RDDs)

A **distributed memory abstraction** that enables **in-memory computations** on large clusters in a **fault-tolerant** manner

- Motivation: iterative algorithms, interactive data mining tools
    - In both cases above keeping data in memory will help enormously for performance improvement
- RDDs are parallel data structures allowing coarse grained transformations
- It ***provides fault-tolerance by storing the lineage as opposed to the actual data*** as done in Hadoop

Transformations vs Actions


copyFromLocal
MAP
JOIN
MAP
REDUCE
copyToLocal

RDD Transformation vs. Action

- **Transformations are lazy:** nothing actually happens when this code is evaluated
- **RDDs are computed only when an *action* is called on them, e.g.,**
  - Calculate statistics over the elements of an RDD (`count`, `mean`)
  - Save the RDD to a file (`saveAsTextFile`)
  - Reduce elements of an RDD into a single object or value (`reduce`)
- **Allows you to define partitioning/caching behavior *after* defining the RDD but *before* calculating its contents**

SDSC SAN DIEGO SUPERCOMPUTER CENTER

RDD Lineage

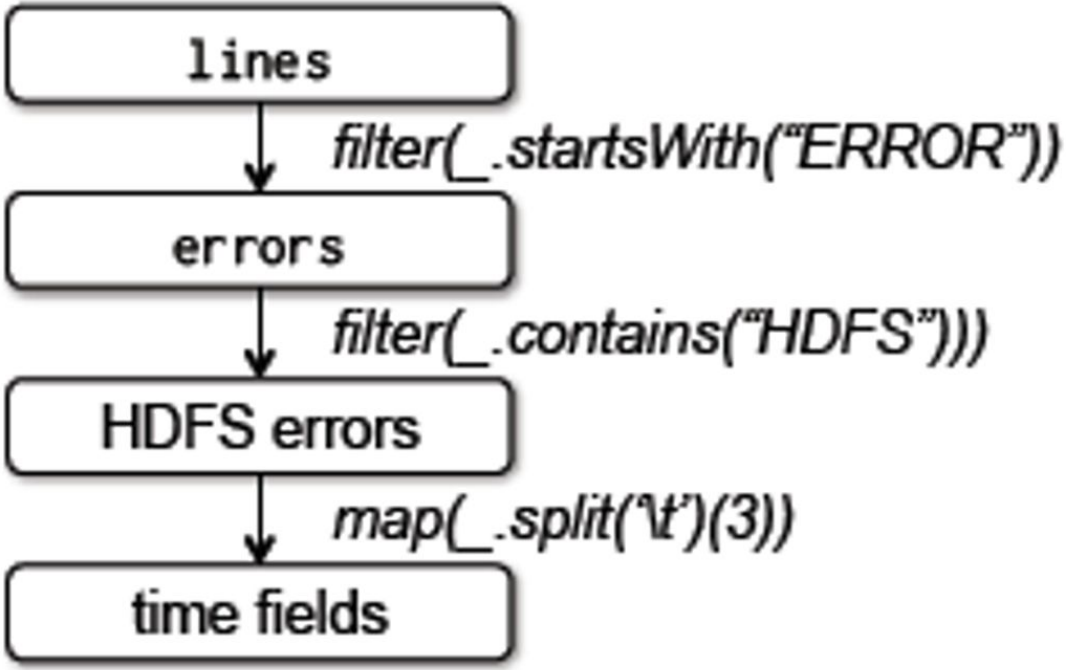
An RDD can depend on zero or more other RDDs

- ie when `x = y.map(...)`, x will depend on y
- These dependency relationships can be thought of as a graph.

You can call this graph a lineage graph, as it represents the derivation of each RDD

- It is also necessarily a ***DAG***, since a loop is impossible to be present in it.
- Narrow dependencies, where a shuffle is not required (think map and filter) can be collapsed into a single **stage**.
    - A stage is a unit of execution, generated by the scheduler from RDD dependency graph
    - Stages also depend on each other and the scheduler builds and uses this dependency graph (which is also necessarily a DAG) to schedule the stages

RDD Lineage


lines
filter(_.startsWith("ERROR"))
errors
filter(_.contains("HDFS")))
HDFS errors
map(_.split('\t')(3))
time fields

Resilience in HDFS vs Spark

HDFS

Fault-tolerance achieved by replicating blocks of data

If a node goes down, the data can be found on another node

Spark

Fault-tolerance achieved by storing chain of transformations

If data is lost, the chain of transformations can be recomputed on the original data

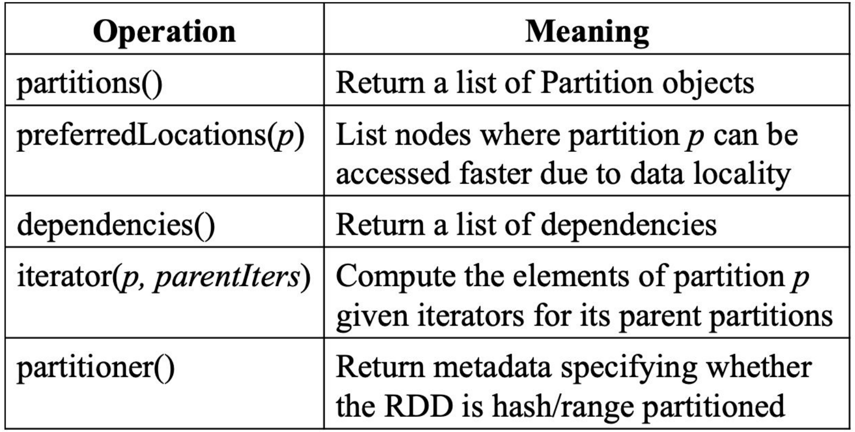
Spark will often use HDFS for stable storage of the original data

Representing RDDs

Each RDD is represented through a common interface that exposes 5 pieces of information:

1. A set of partitions, atomic pieces of datasets
2. Set of dependencies on the parent RDDs
3. Function for computing the RDD from the parents
4. Metadata about partitioning scheme
5. Data placement

See table 3 in the RDD paper →



Operation	**Meaning**
partitions()	Return a list of Partition objects
preferredLocations(*p*)	List nodes where partition *p* can be accessed faster due to data locality
dependencies()	Return a list of dependencies
iterator(*p, parentIters*)	Compute the elements of partition *p* given iterators for its parent partitions
partitioner()	Return metadata specifying whether the RDD is hash/range partitioned

Dependencies

Narrow dependencies: each parent RDD partition used by at most one child; ie map()

- allow pipelined execution: example map() and filter() in iterative fashion
- recovery after node failure is more efficient

Wide dependencies: multiple child partitions may depend on a parent RDD; ie join()

- Single failed node in a wide dependency lineage graph may cause loss of partition in many ancestral dependencies

Example Transformations

Map: Applying ***map*** to an RDD results in a new MappedRDD whose partitions and preferred locations are the same as the parent. It's iterator method applies the passed in function to the parent partitions.

Example Transformations

Map: Applying ***map*** to an RDD results in a new MappedRDD whose partitions and preferred locations are the same as the parent. It's iterator method applies the passed in function to the parent partitions.

Union: Called on 2 RDDs and returns an RDD whose partitions are the union of the parents partitions. Each child partition is computed from the corresponding parent partition.

Example Transformations

Map: Applying ***map*** to an RDD results in a new MappedRDD whose partitions and preferred locations are the same as the parent. It's iterator method applies the passed in function to the parent partitions.

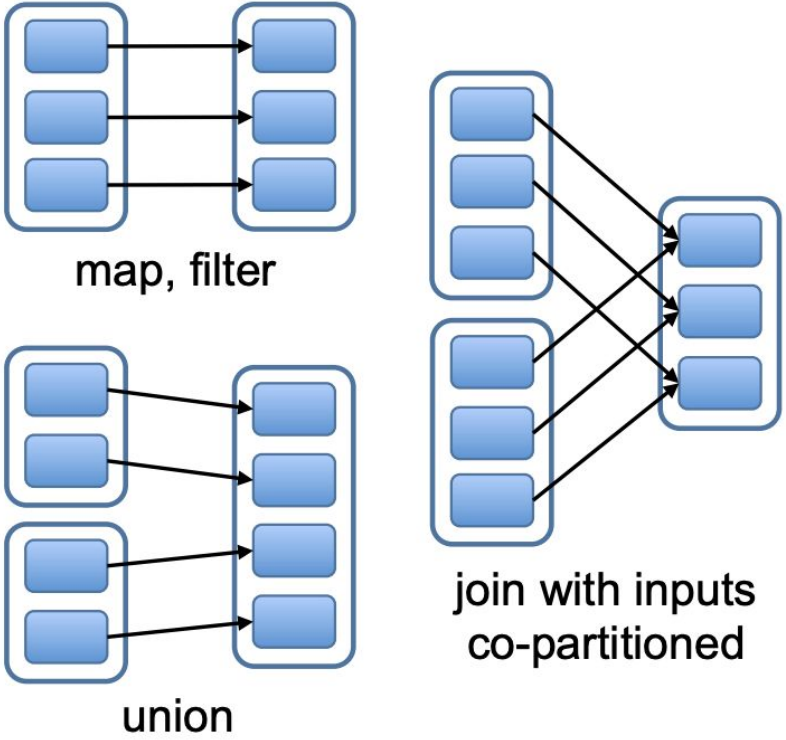
Union: Called on 2 RDDs and returns an RDD whose partitions are the union of the parents partitions. Each child partition is computed from the corresponding parent partition.

Join: Joining two RDDs leads to two narrow dependencies if both parents are partitioned with the same partitioner), two wide dependencies, or a mix

Narrow Dependencies

Narrow dependencies: each parent RDD partition used by at most one child

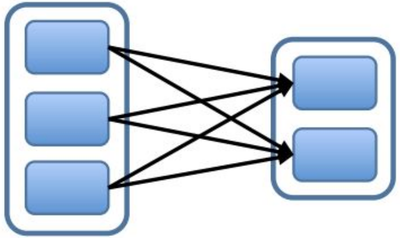
- We can pipeline computation of multiple narrow dependencies (compute map, followed by filter on a per element basis for example)



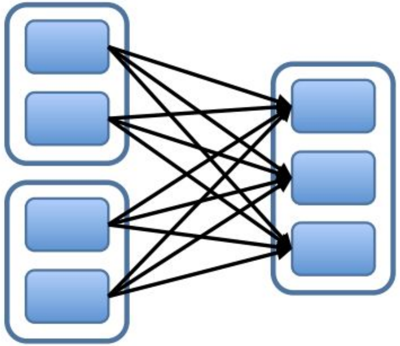
Wide Dependencies

Wide dependencies: multiple child partitions may depend on a parent RDD

- All data from all parents must be available (may require expensive data shuffling)
- **Note:** Joins may be either narrow or wide (or mixed) depending on how parents are partitioned



groupByKey



join with inputs not co-partitioned

Execution Model

Remember: *Transformations* are lazily applied; *Actions* result in actual computation

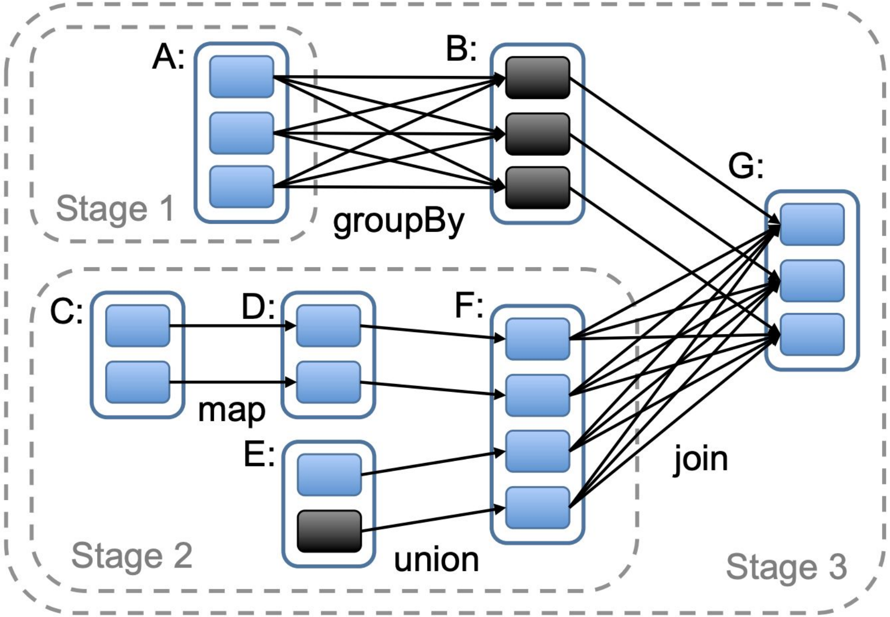
When a user runs an **action** on an RDD, the scheduler uses that RDD's lineage graph to build a DAG of **stages.**

- Each stage contains as many pipelined transformations (with narrow dependencies) as possible
- Stage boundaries determined by wide dependencies, or already computed data

Execution Model

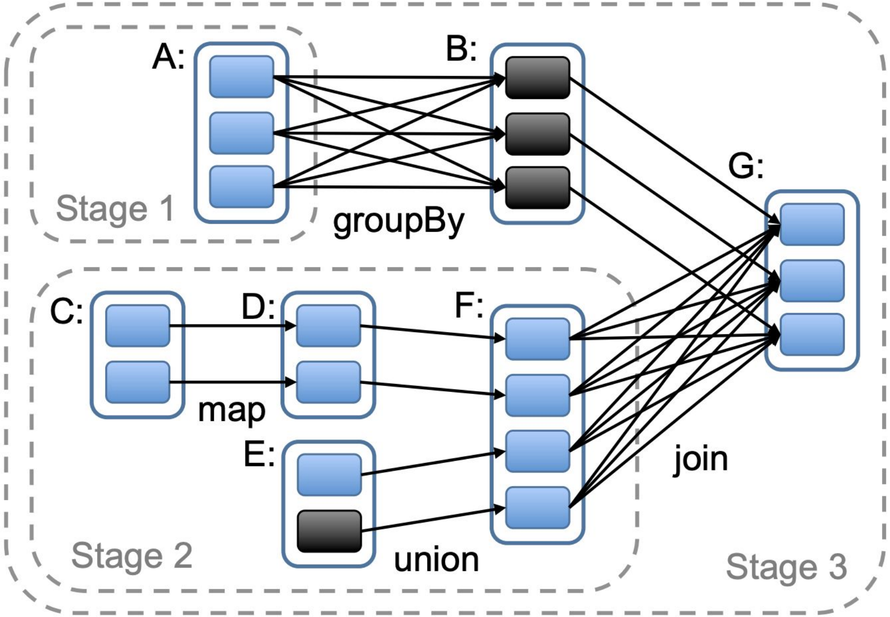
The figure to the right shows RDDs A-G, and the transformations used to derive them.

Black boxes are partitions that are already computed and stored in memory.



Execution Model

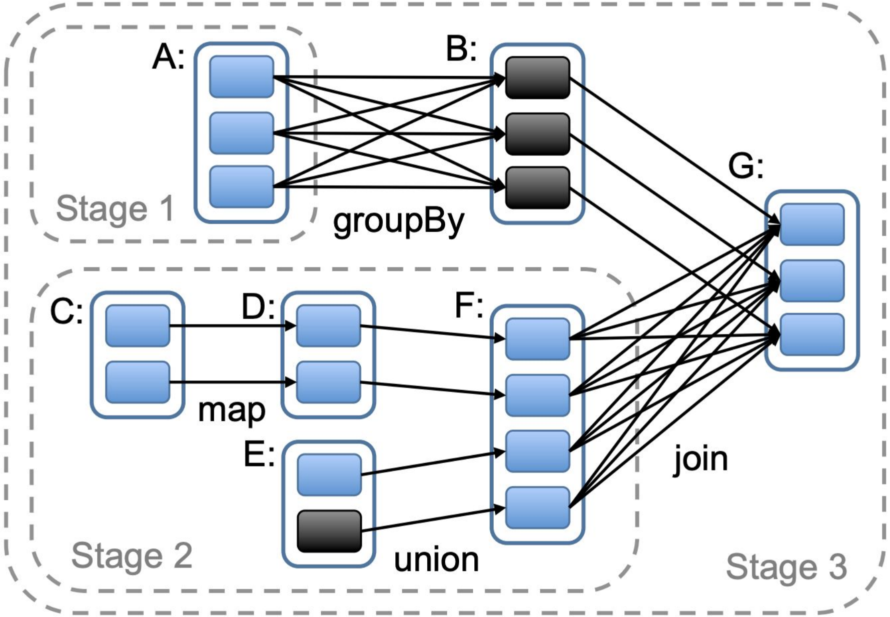
Stage 1: RDD B is derived from RDD A by a groupBy transformation.



Execution Model

Stage 1: RDD B is derived from RDD A by a groupBy transformation.

The groupBy results in wide dependencies, and therefore required data to be shuffled.

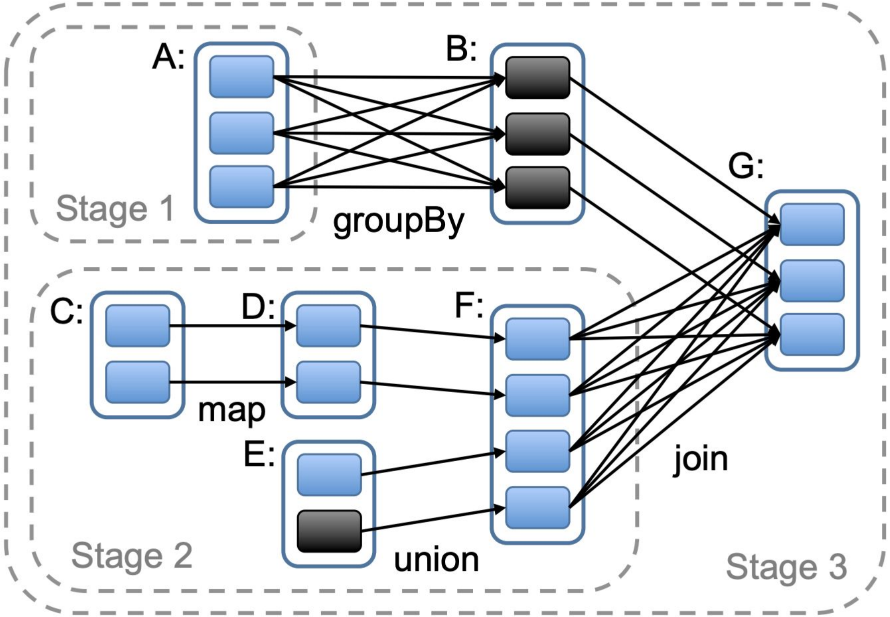


Execution Model

Stage 1: RDD B is derived from RDD A by a groupBy transformation.

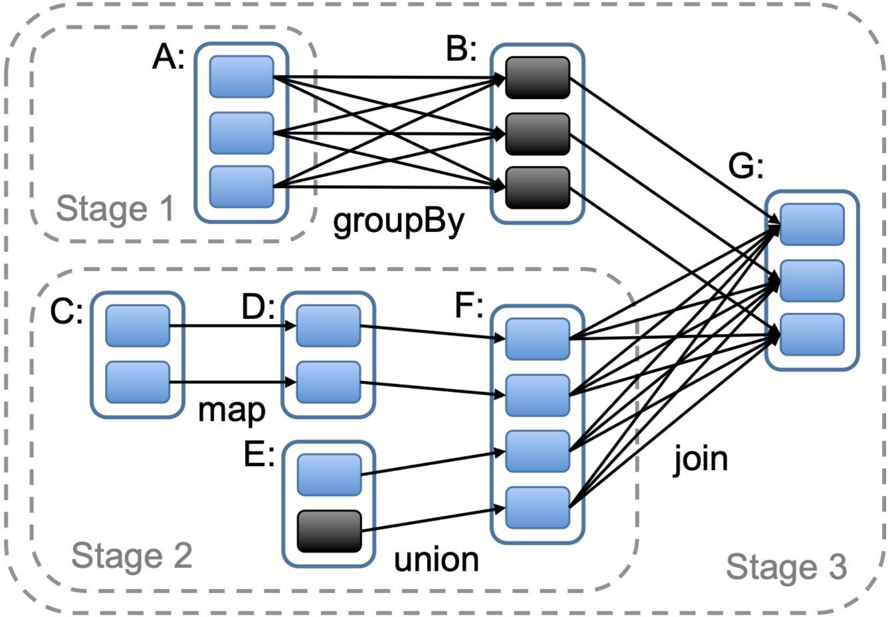
The groupBy results in wide dependencies, and therefore required data to be shuffled.

The groupBy therefore is the boundary of stage 1.



Execution Model

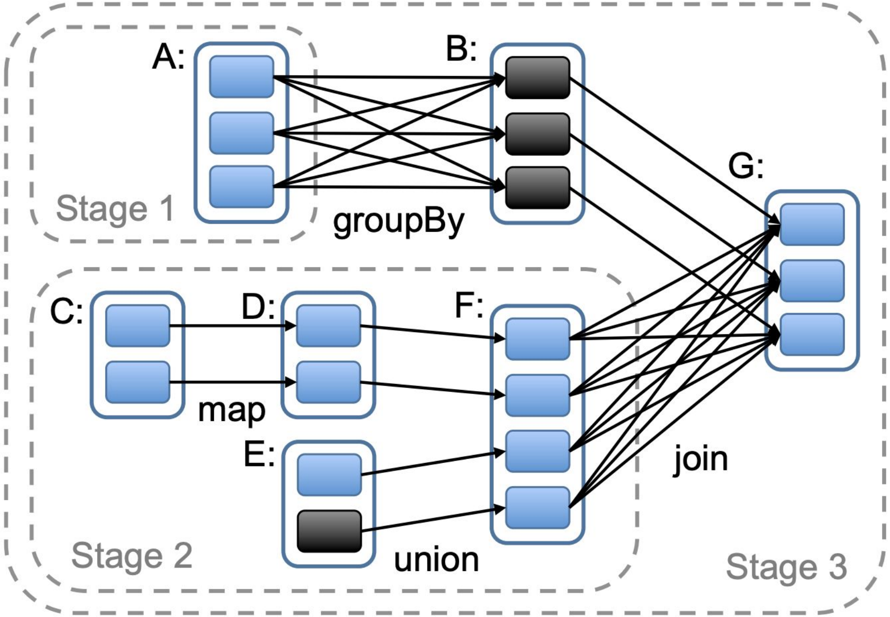
Stage 2: RDD F is derived by a union on D and E. D is derived by map on C.



Execution Model

Stage 2: RDD F is derived by a union on D and E. D is derived by map on C.

All of these operations involve narrow dependencies and can be pipelined.

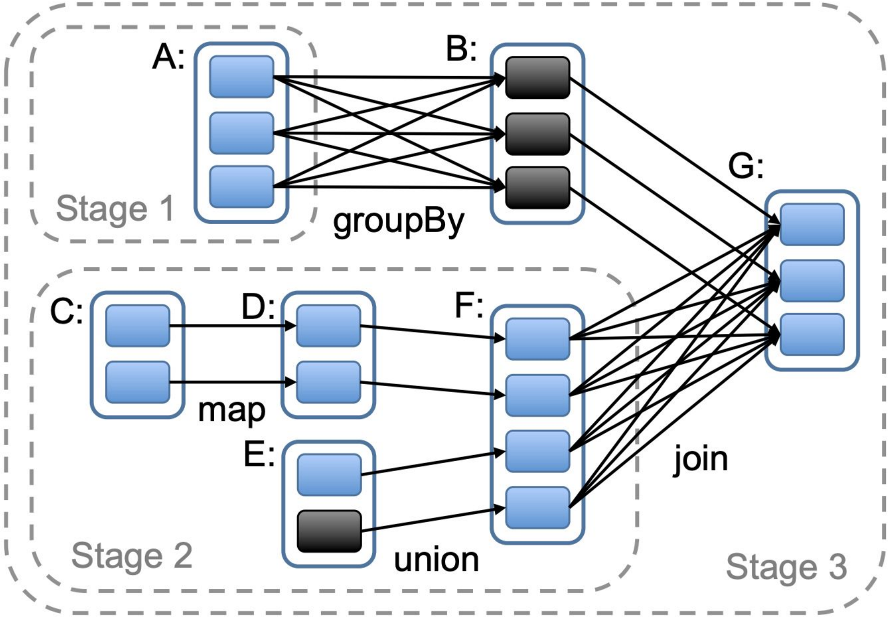


Execution Model

Stage 2: RDD F is derived by a union on D and E. D is derived by map on C.

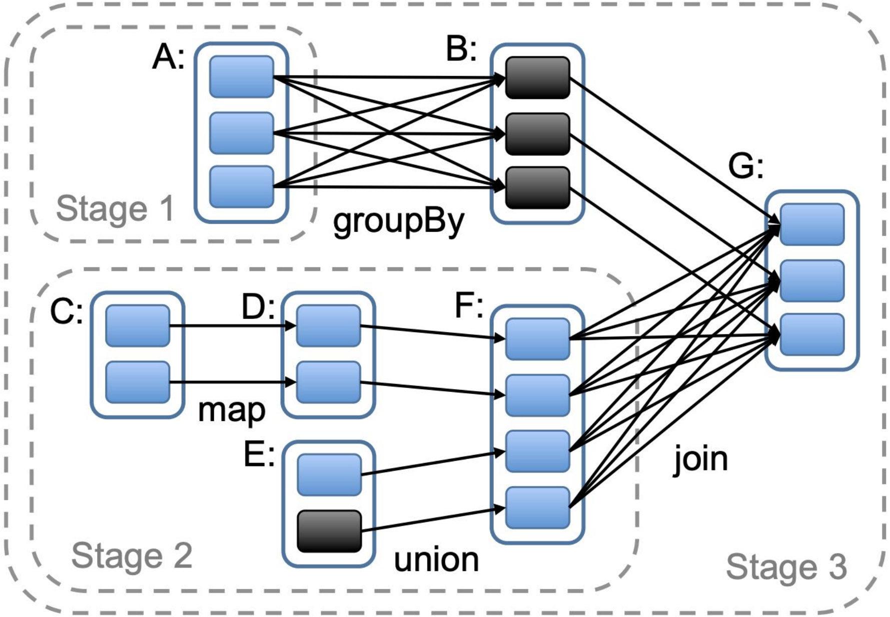
All of these operations involve narrow dependencies and can be pipelined.

RDD G is the result of join on F and B, so this is the boundary of stage 2.



Execution Model

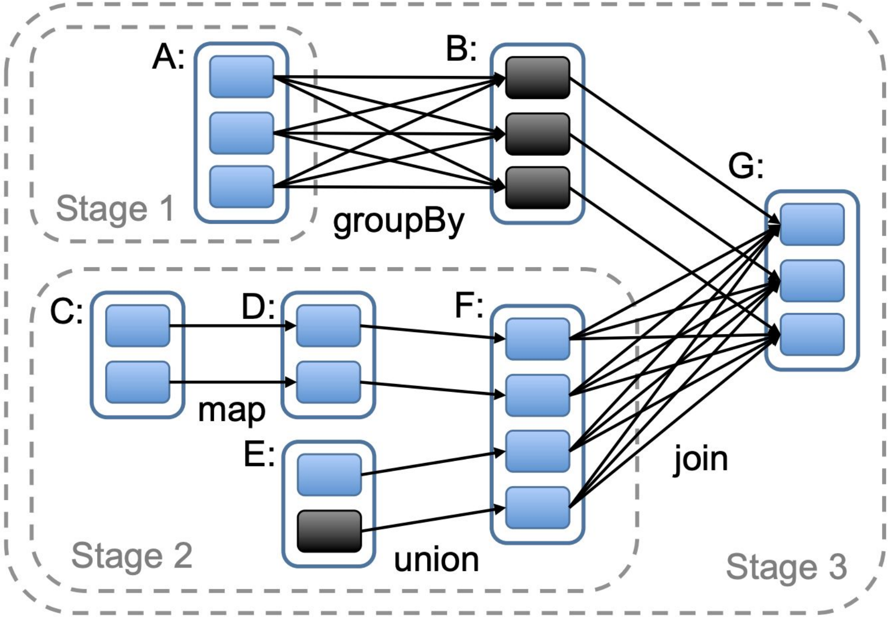
Stage 3: RDD G is derived from a join on RDD B and G.



Execution Model

Stage 3: RDD G is derived from a join on RDD B and G.

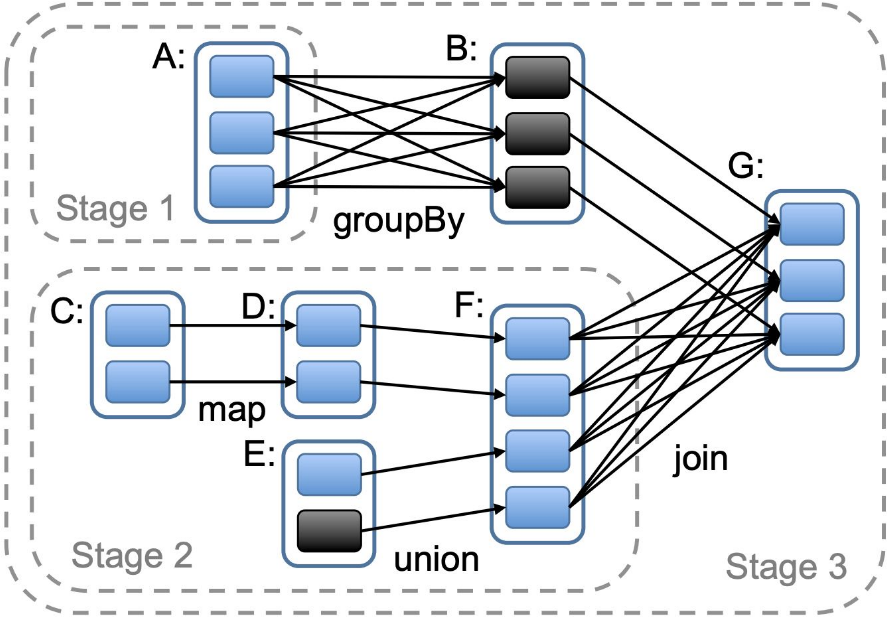
G is NOT COMPUTED until the user executes an action on G, ie saving to disk, or performing a reduction.



Execution Model

When the user calls an action on G:

Stage one does not need to be executed (it's result is already in memory)

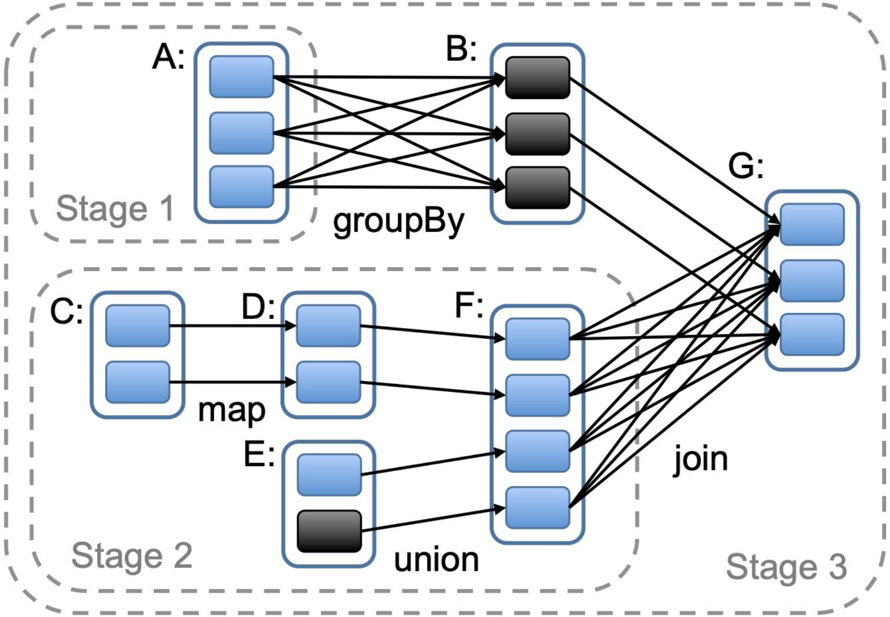


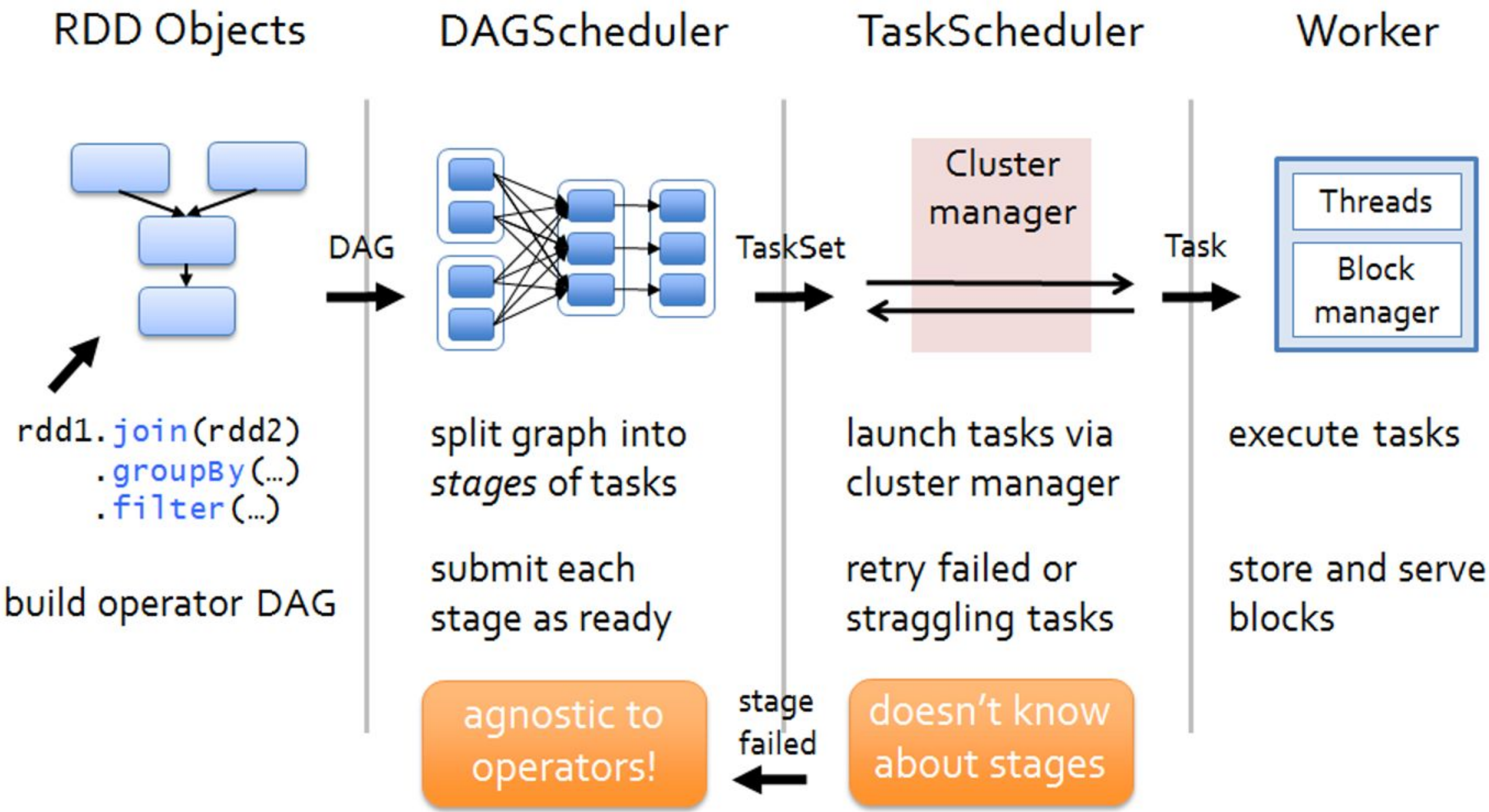
Execution Model

When the user calls an action on G:

Stage one does not need to be executed (it's result is already in memory)

Stage 2 is scheduled for execution, followed by stage 3.





RDD Objects

```
rdd1.join(rdd2)
    .groupBy(…)
    .filter(…)
```

build operator DAG

DAG

DAGScheduler

split graph into *stages* of tasks

submit each stage as ready

agnostic to operators!

TaskSet

stage failed

TaskScheduler

Cluster manager

launch tasks via cluster manager

retry failed or straggling tasks

doesn’t know about stages

Task

Worker

Threads

Block manager

execute tasks

store and serve blocks

Optimizing: Persisting and Partitioning

Spark applications separate application logic from optimization logic

This allows developers to focus on correctness and performance separately.

Optimizing: Persisting and Partitioning

Spark applications separate application logic from optimization logic
This allows developers to focus on correctness and performance separately.

We saw a similar pattern with MapReduce: Correctness was entirely determined by Map and Reduce tasks, but then components like combiners and partitioners could provide performance benefits without changing correctness.

Partitioning

Spark allows us to specify how our data is partitioned

- Careful choice of partitioning can allow for more efficient execution
- For example, if two RDDs have the same partitioning scheme, performing a join transformation on them results in narrow dependencies (can be pipelined, cheaper fault-tolerance)
- Can also avoid the need for some communication

Persisting

Spark allows us to "persist" an RDD (keep it in memory)

- Spark allows users to call **persist()** on RDDs to keep them in storage (either in memory or on disk, depending on what we ask for)
- By persisting an RDD, we will not have to re-compute it or re-read it from disk in the future
- For iterative applications, this can result in huge performance gains that are not feasible with something like MapReduce

Example: PageRank

PageRank in Spark requires 3 RDDs:

1. The RDD containing the static graph **links** we are working with. Does not change across iterations.

Example: PageRank

PageRank in Spark requires 3 RDDs:

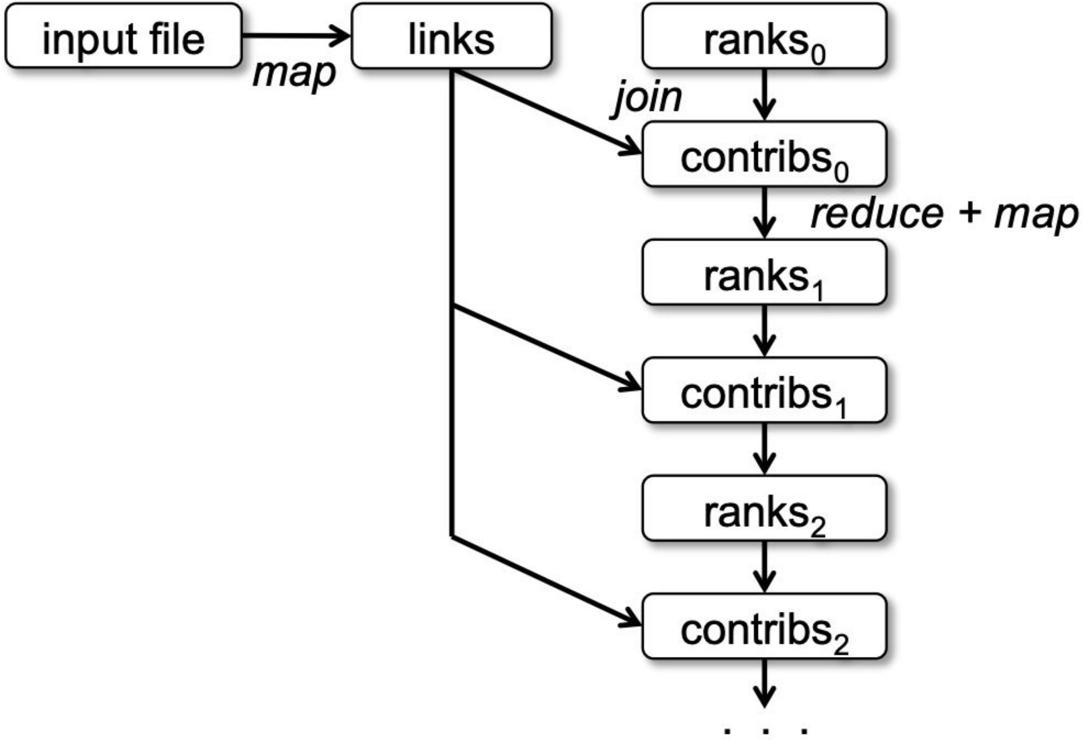
1. The RDD containing the static graph **links** we are working with. Does not change across iterations.
2. The RDD containing the **ranks** of each vertex for the current iteration. Derived from contributions from previous iterations.

Example: PageRank

PageRank in Spark requires 3 RDDs:

1. The RDD containing the static graph **links** we are working with. Does not change across iterations.
2. The RDD containing the **ranks** of each vertex for the current iteration. Derived from **contributions** from previous iterations.
3. The RDD containing the **contributions** of each page to its neighbors. This is the result of a **join** on **ranks** and **links**

Example: PageRank



Lineage graph for PageRank in Spark

Example: PageRank

Once we have the PageRank logic correct, we can optimize by effective use of persistence and partitioning:

Example: PageRank

Once we have the PageRank logic correct, we can optimize by effective use of persistence and partitioning:

Partitioning: Each iteration we must perform a **join** on **links** and **ranks**.

Example: PageRank

Once we have the PageRank logic correct, we can optimize by effective use of persistence and partitioning:

Partitioning: Each iteration we must perform a **join** on **links** and **ranks**.

If we partition the data for both RDDs the same (for example, hashing based on URL), then the data can be co-located and pipelined efficiently.

Example: PageRank

Once we have the PageRank logic correct, we can optimize by effective use of persistence and partitioning:

Partitioning: Each iteration we must perform a **join** on **links** and **ranks**.

If we partition the data for both RDDs the same (for example, hashing based on URL), then the data can be co-located and pipelined efficiently.

This will also mean our **joins** will not require data shuffling/communication.

Example: PageRank

Once we have the PageRank logic correct, we can optimize by effective use of persistence and partitioning:

Persistence: The **links** RDD is required every iteration. If we persist the **links** RDD it will be in memory when we need it.

Example: PageRank

Once we have the PageRank logic correct, we can optimize by effective use of persistence and partitioning:

Persistence: The **links** RDD is required every iteration. If we persist the **links** RDD it will be in memory when we need it.

As we perform more iterations, the lineage graph gets longer. For better resilience we can also persist some intermediate iterations.