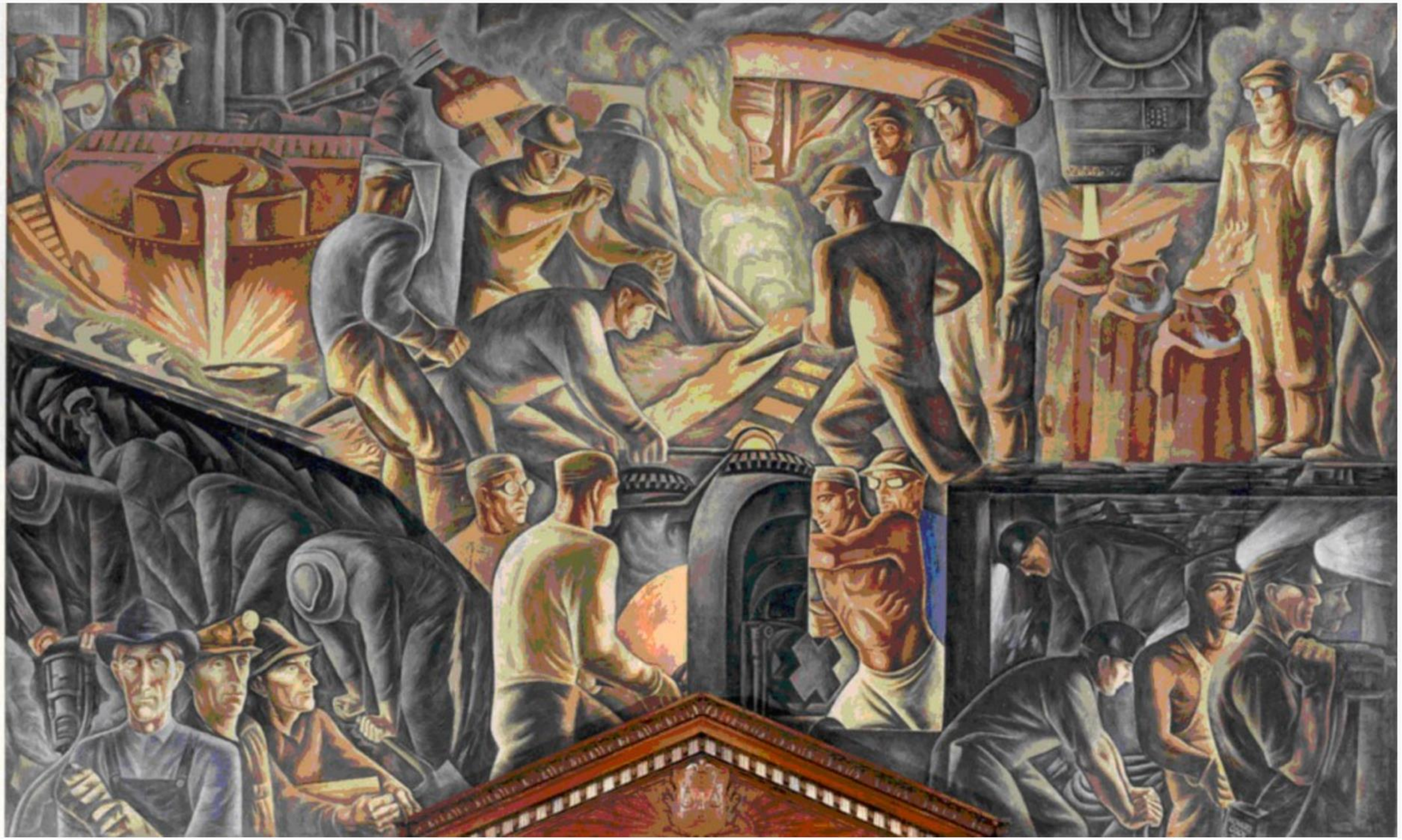
Cluster Computing: Spark

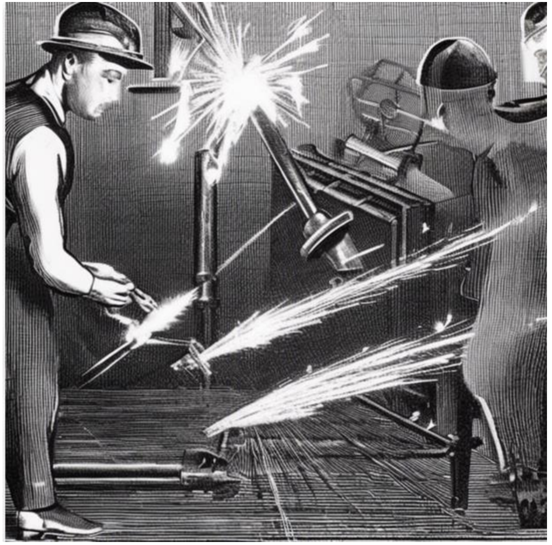
Some slides from Mosharaf Chowdhury, Sam Madden



”Steel Industry”, fresco in Pittsburgh US Courthouse and Post Office, Howard Norton Cook, 1936

“metal sparks industry in heroic early 20th century style”, Stable Diffusion, November 26, 2022

”Steel Industry”, fresco in Pittsburgh US Courthouse and Post Office, Howard Norton Cook, 1936

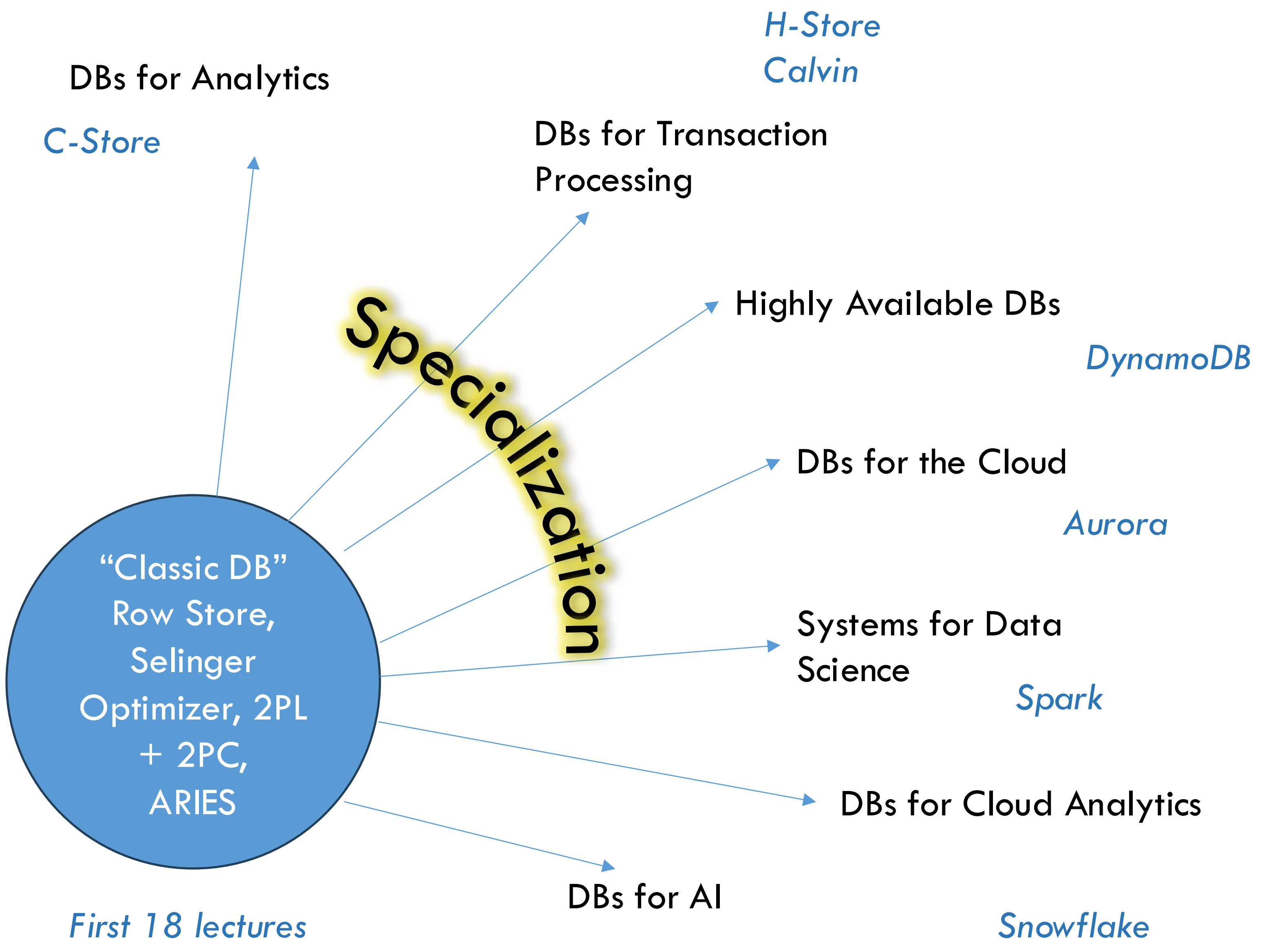


“metal sparks industry in heroic early 20th century style”, Stable Diffusion, November 26, 2022

same prompt, DALL-E 3, 2023

GPT-4, November 17, 2024

Where Are We???



H-Store
Calvin
DBs for Analytics
C-Store
DBs for Transaction Processing
Highly Available DBs
DynamoDB
Specialization
DBs for the Cloud
Aurora
"Classic DB" Row Store, Selinger Optimizer, 2PL + 2PC, ARIES
Systems for Data Science
Spark
DBs for Cloud Analytics
DBs for AI
First 18 lectures
Snowflake

Today

- Data Systems for "Data Science"
  - Efficient Parallel Execution for "One Off" Data Processing Tasks
    - E.g., featurization for ML, indexing data, extracting information from data, etc
  - Often involving unstructured → structured data conversion
    - E.g., processing a set of text document into an inverted index of words and their locations in the documents
  - Not really SQL, but a set of parallel operations that are reminiscent of SQL filters and joins

- MapReduce/Hadoop, briefly, and then Spark

MapReduce: programming model for processing large data sets across a distributed cluster.

- Programmer specifies:

  Map Function:

  - Processes input key/value pairs to generate intermediate key/value pairs.

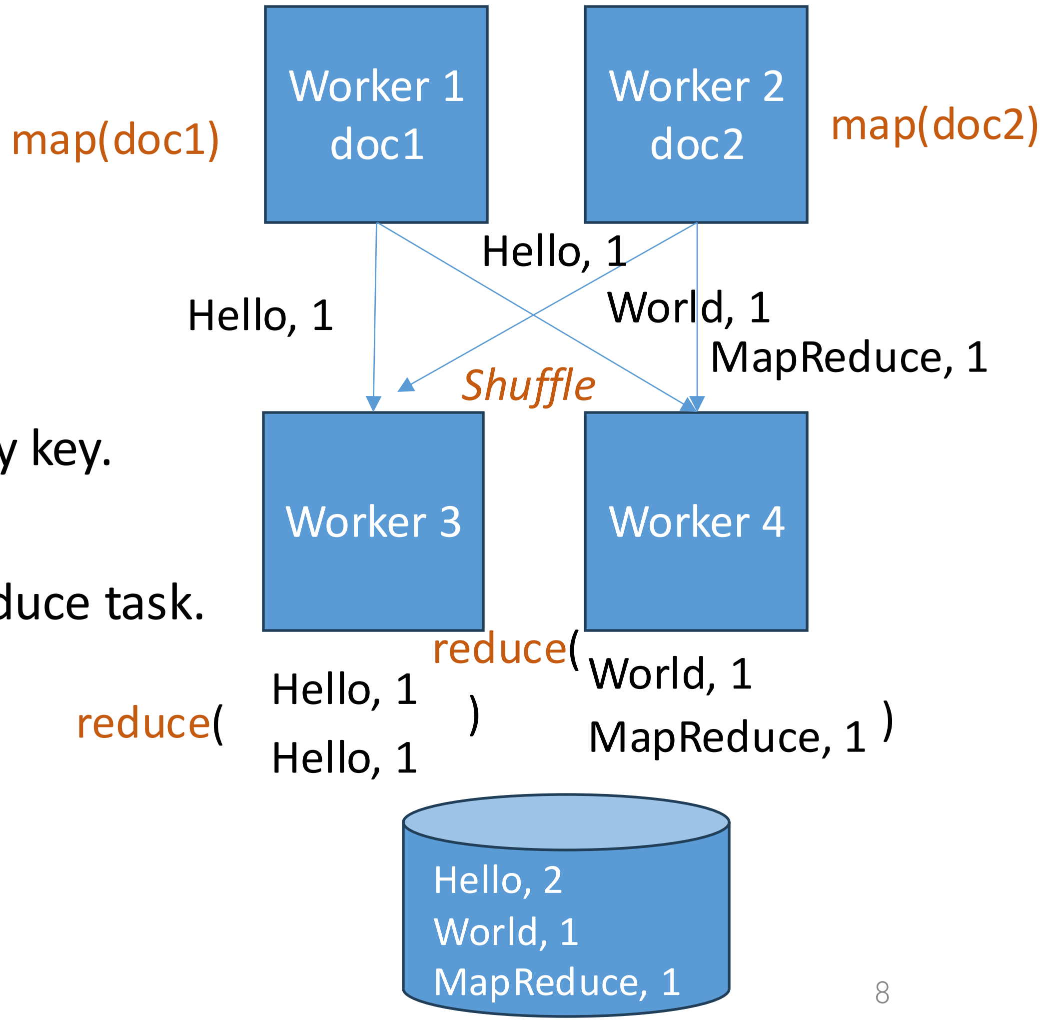
  Reduce Function:

  - Merges all intermediate values associated with the same intermediate key.

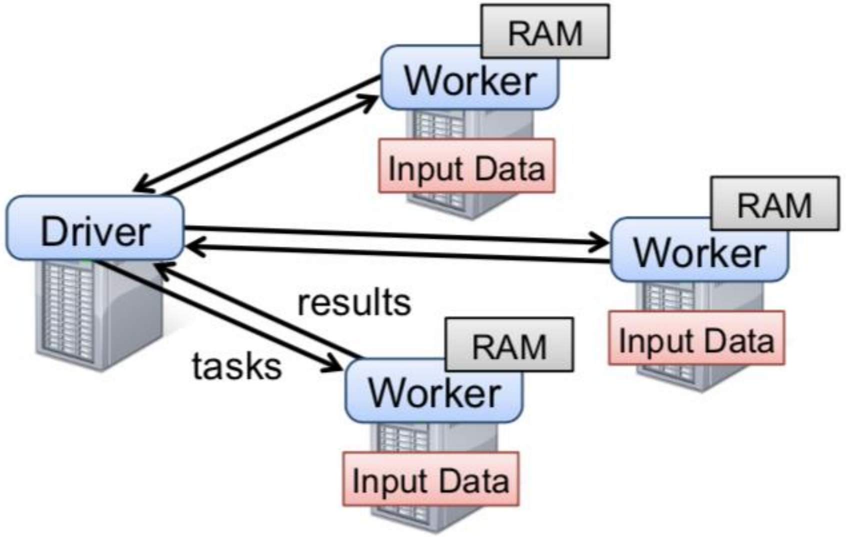
```
# Map Function
def map(key, value):
       for word in value.split():
              emit(word, 1)
# Reduce Function
def reduce(key, values):
       total_count = sum(values)
       emit(key, total_count)
# Example Input: [('doc1', 'hello world'), ('doc2', 'hello mapreduce')]
# MapReduce Process Execution
# Example Output: [('hello', 2), ('world', 1), ('mapreduce', 1)]
```

MapReduce Execution

- **Input Splitting:**
  - Data is divided into chunks for the map tasks.
- **Mapping:**
  - Each chunk is processed by a map task independently.
- **Shuffling:**
  - Intermediate key/value pairs are sorted and grouped by key.
- **Reducing:**
  - Each group of intermediate values is processed by a reduce task.
- **Output:**
  - Final output is generated from the reduce tasks.



Motivation & Background



Frameworks back in 2012:
MapReduce, a bit of Microsoft's Dryad

- Pros:
  - Allowed parallel computation without worrying low level details (e.g., work distribution, fault tolerance)
  - Provided a set of high-level operations (map, reduce)
  - You didn't have to think about schemas
- Cons:
  - Little to no support for leveraging cluster memory
  - Large overhead for reusing data in iterative or interactive tasks (I/O, replication, serialization)
  - You didn't have to think about schemas
  - Implementations had bad latency

Spark: Resilient Distributed Datasets (RDDs)

- Utilize Distributed Memory while providing efficient fault tolerance
  - Avoid storing data updates explicitly
  - Instead, obtain fault tolerance by logging transformations (*lineage*)
- Limit operations to coarse-grained transformations (e.g., map, filter)
- Allow user control of data persistence, partitioning, and caching

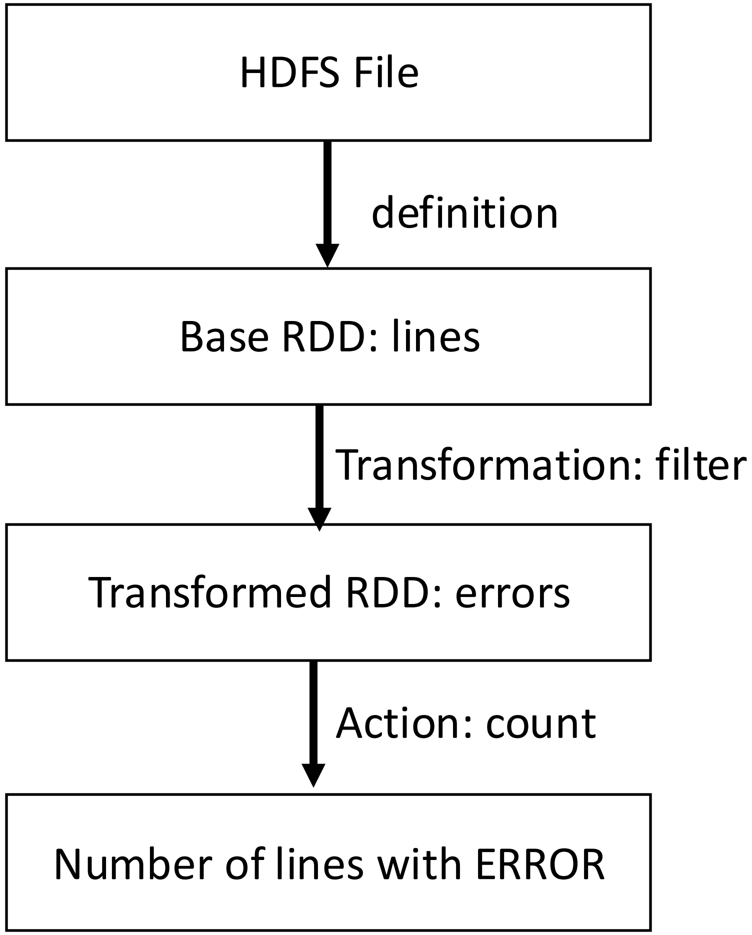
- How did MapReduce obtain fault tolerance?

RDDs

- **Read-only**, partitioned collection of records
- Created from either data in stable storage or other RDDs
- A sequence of **transformations** defines an RDD:
  - Map, filter, flatmap, sample, groupbykey, reducebykey, join, union
- **Actions** return value or export data to storage system
  - count, collect, save, reduce, lookup
- No need to actually run code until there's an **action**
- **Read-only** means we can exploit speculative (re)execution
  - MapReduce also does this

Example: Console Log Mining

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

errors = lines.filter(_.startsWith("ERROR"))

errors.count()
```



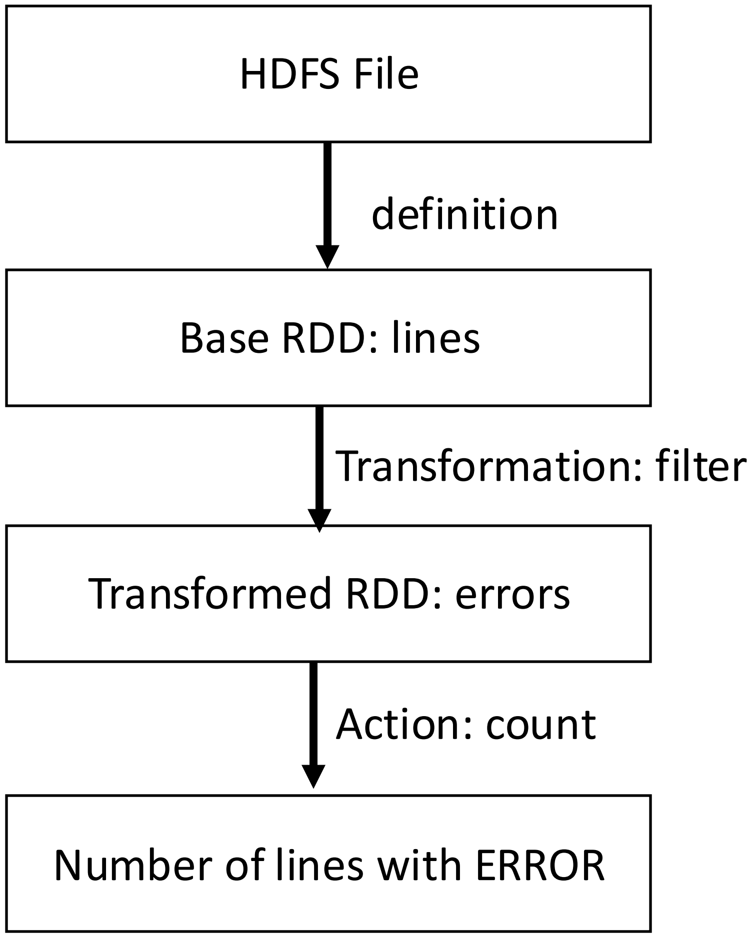
RDDs: Fault Tolerance

Limit operations to coarse-grained transformations and **only log the transformations** instead of replicating data for recovering

- **Lineage:** transformations used to build a dataset
- Recover lost partition by applying lineage from corresponding data partition in stable storage
- Because data is read-only, this is always possible

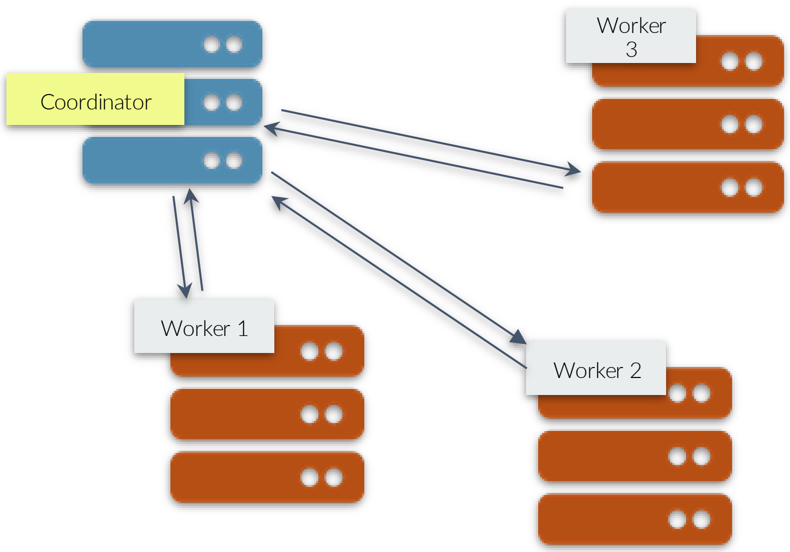
Example: Console Log Mining

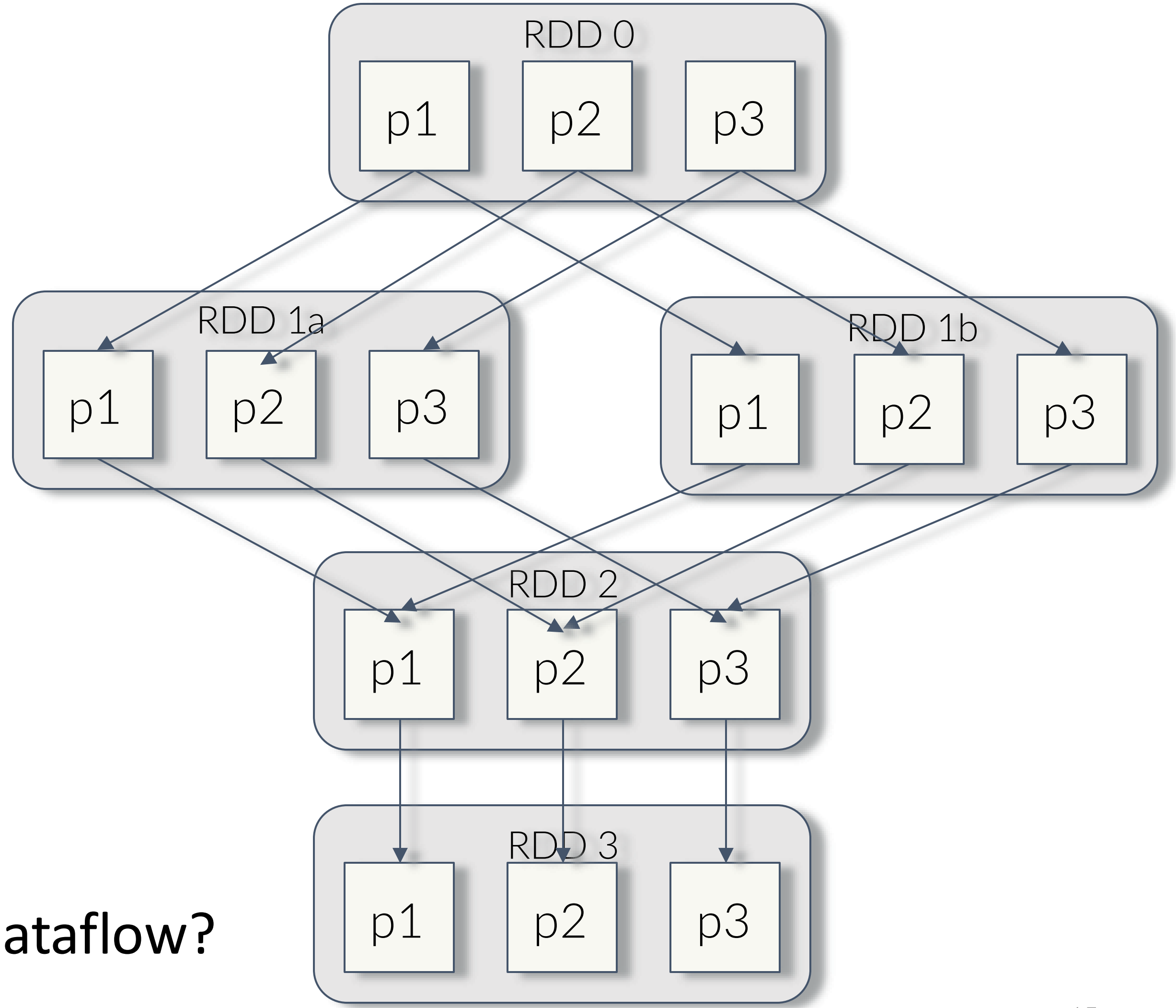
```
lines = spark.textFile("hdfs://...")

errors = lines.filter(_.startsWith("ERROR"))

errors.count()
```



“Base RDD” and “Transformed RDD” may never be actually stored on disk

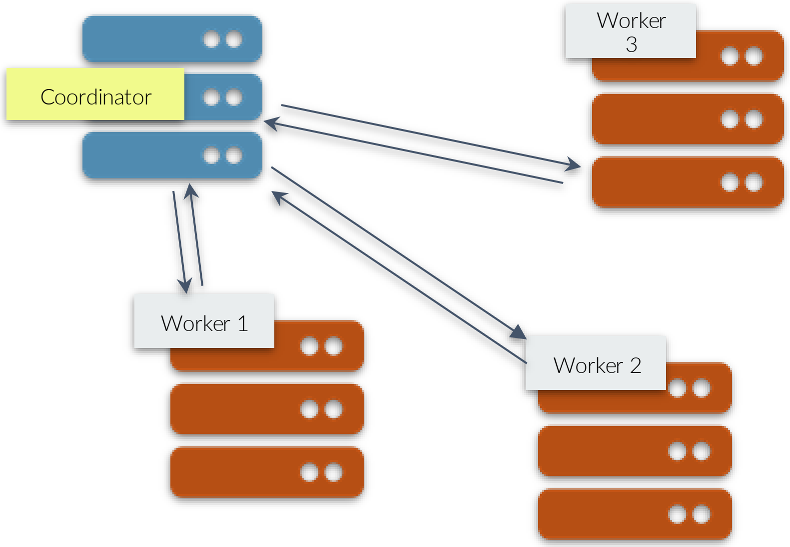
Example

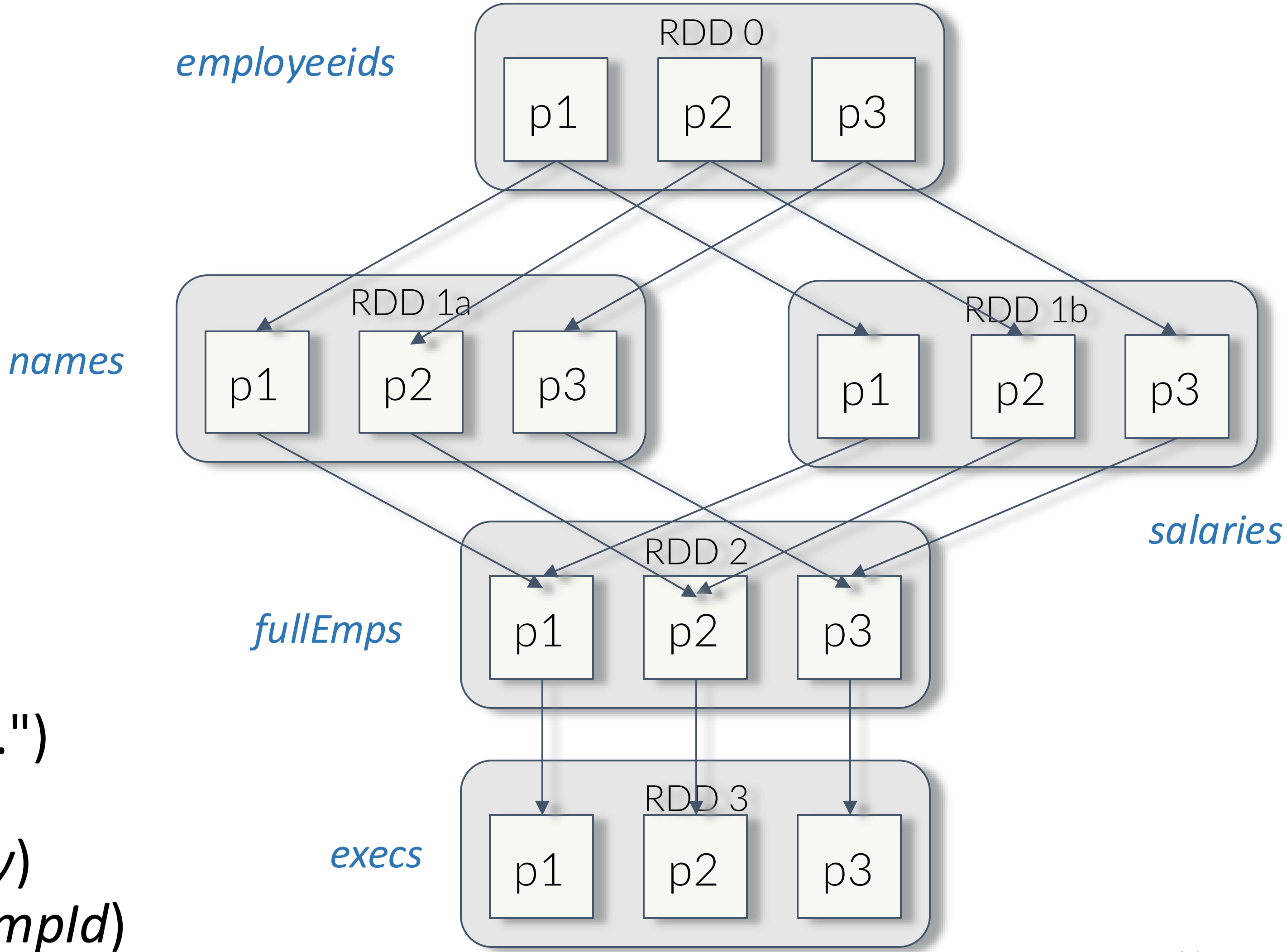




4 RDDs
3 partitions. (e.g., all p1s are on worker 1)
What's a possible program that leads to this dataflow?
Take a minute.

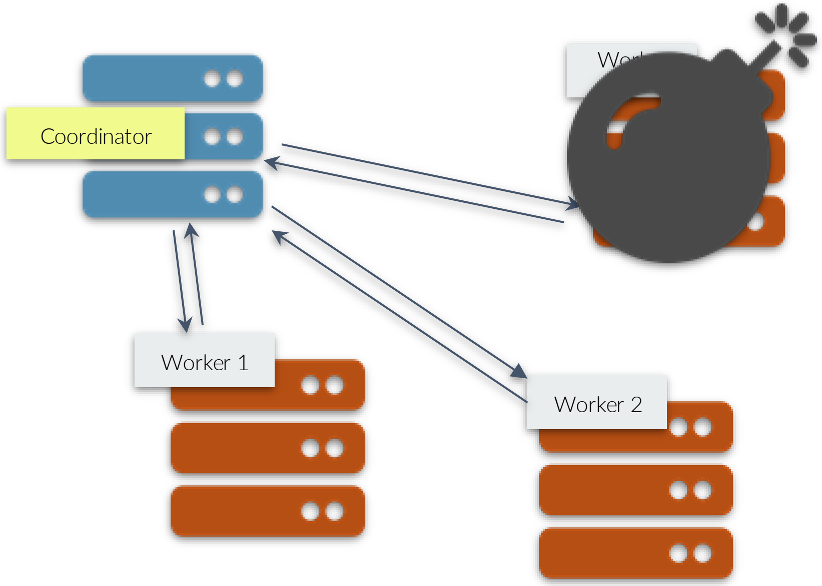
Example

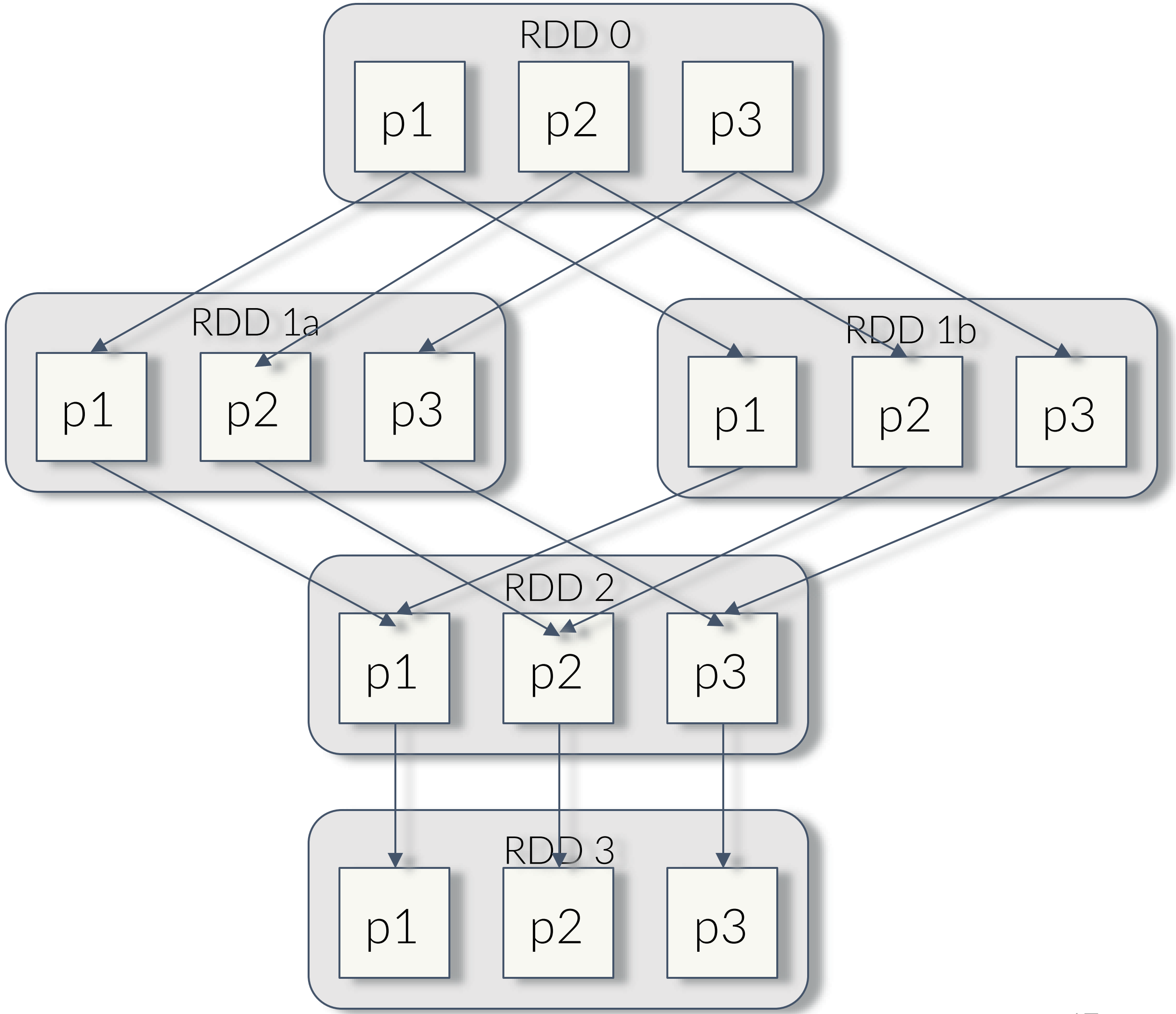




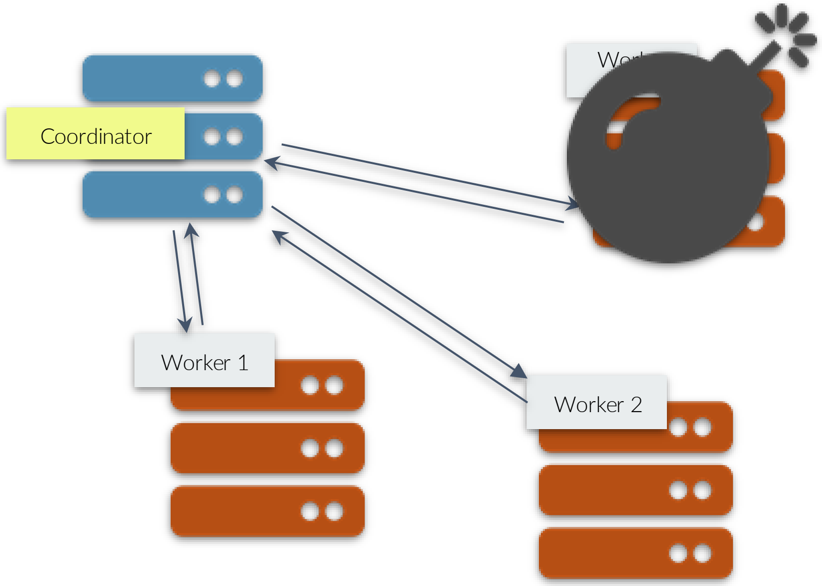
employeeIds = spark.textFile("hdfs://...")
names = map(*# map from ID to name*)
salaries = map(*# map from ID to salary*)
fullEmps = join(*# names, salaries on empId*)
execs = fullEmps.filter(*# filter on salary*)

Example

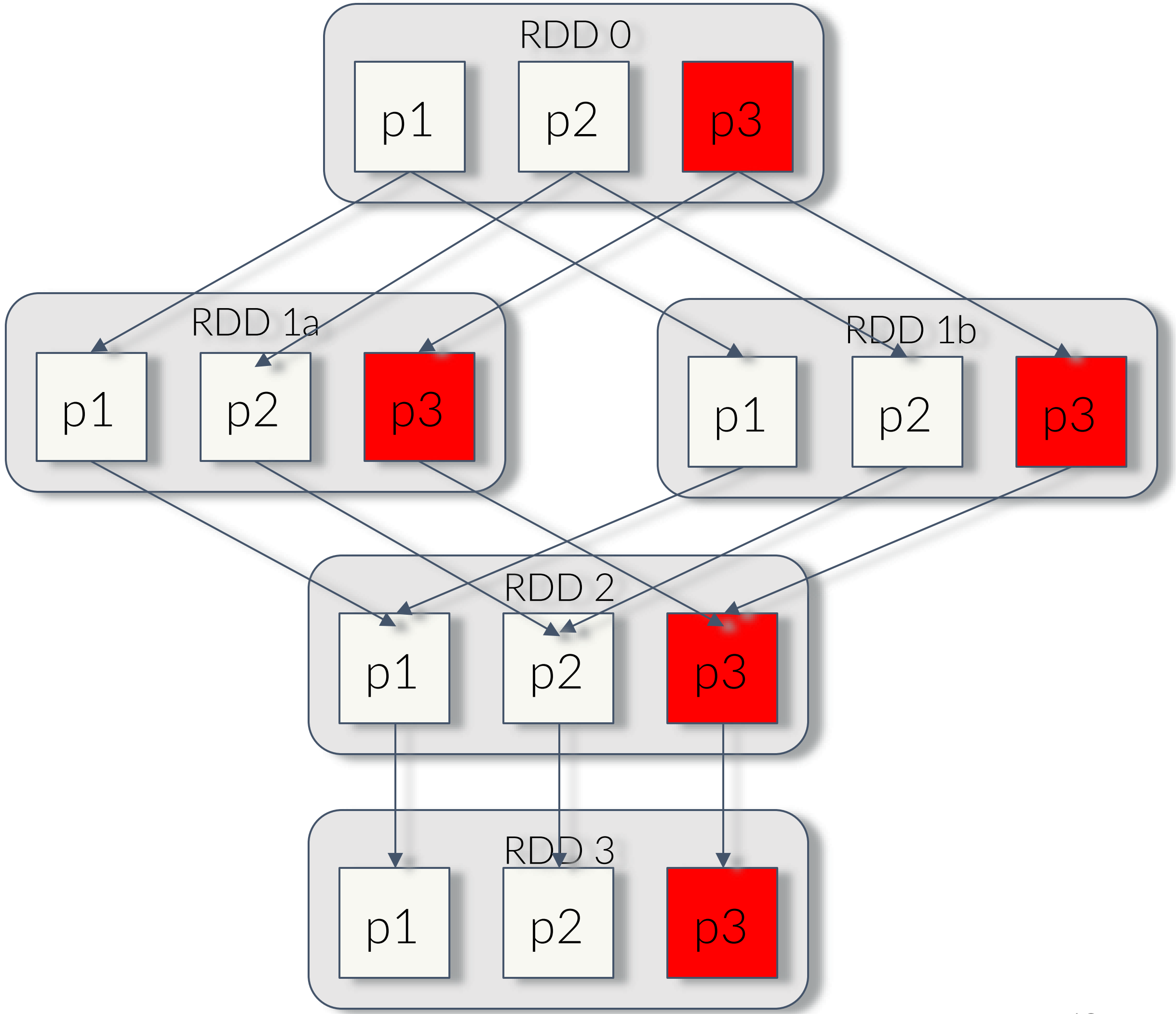

Coordinator
Worker 1
Worker 2


RDD 0
p1
p2
p3
RDD 1a
p1
p2
p3
RDD 1b
p1
p2
p3
RDD 2
p1
p2
p3
RDD 3
p1
p2
p3

Example

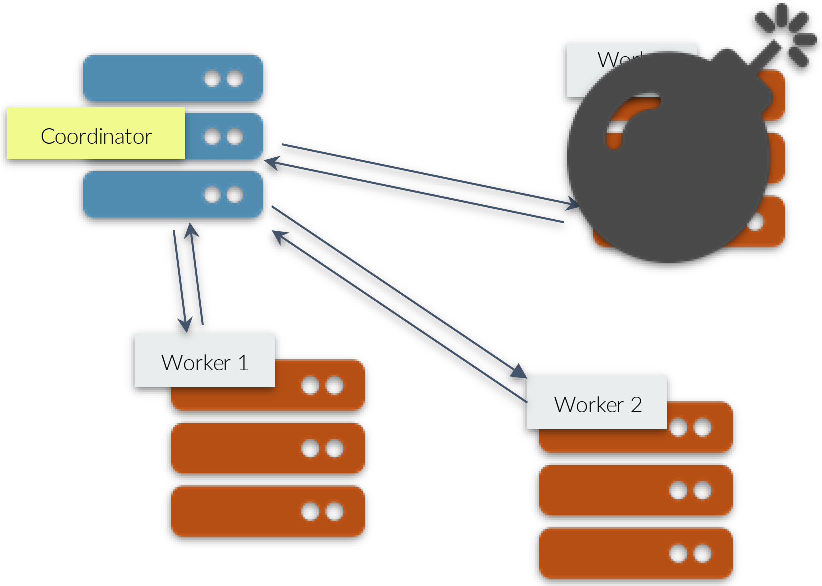


Coordinator
Worker 1
Worker 2

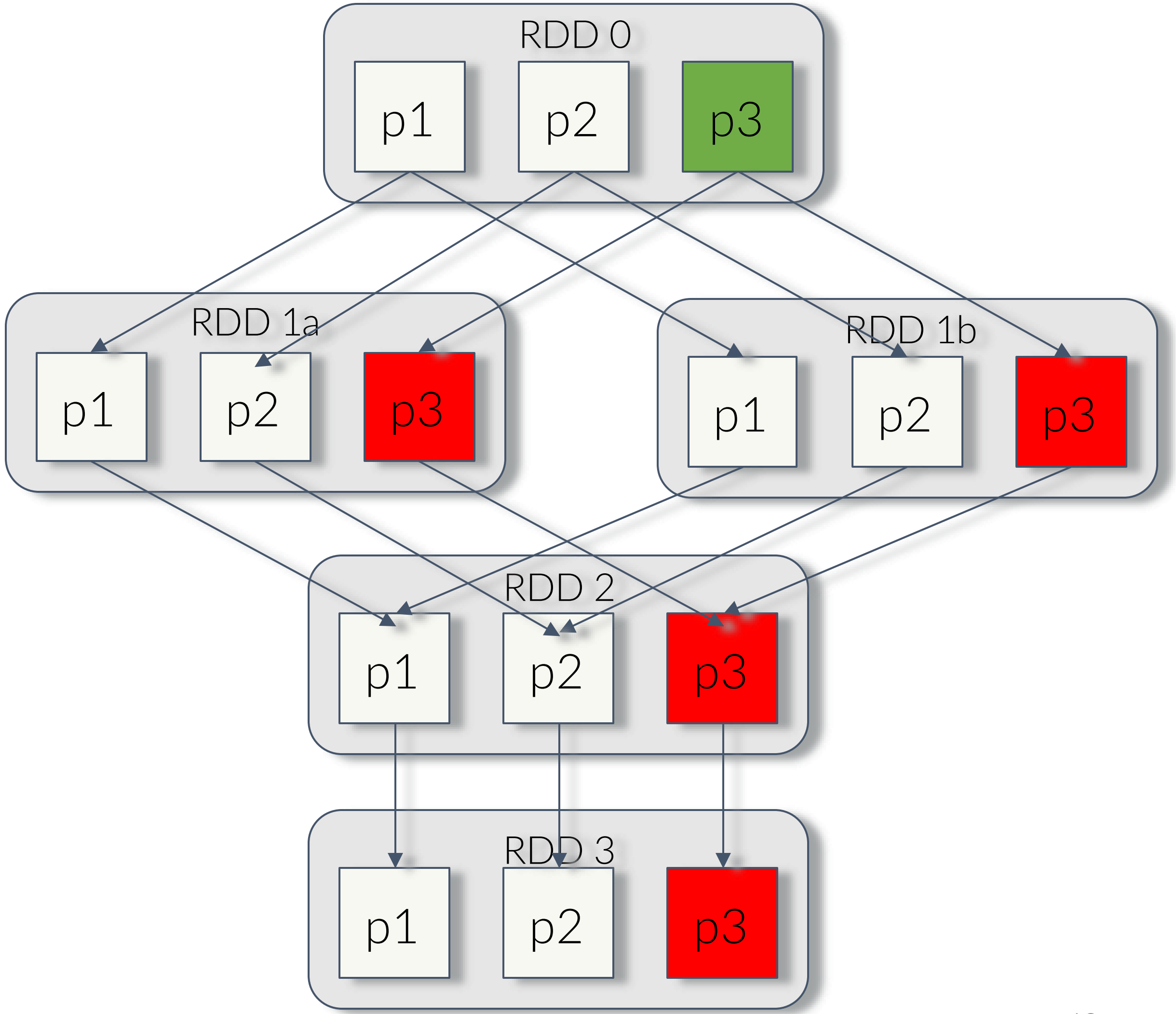


RDD 0
p1
p2
p3
RDD 1a
p1
p2
p3
RDD 1b
p1
p2
p3
RDD 2
p1
p2
p3
RDD 3
p1
p2
p3

Example

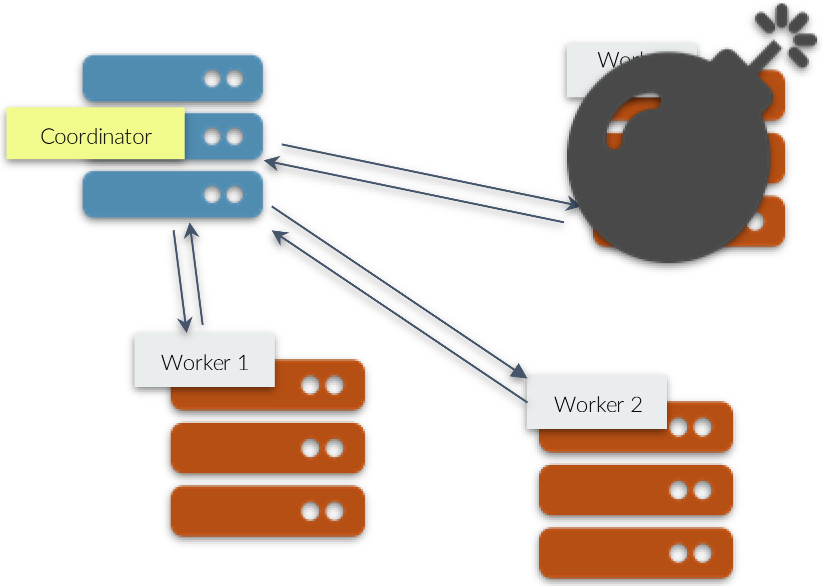


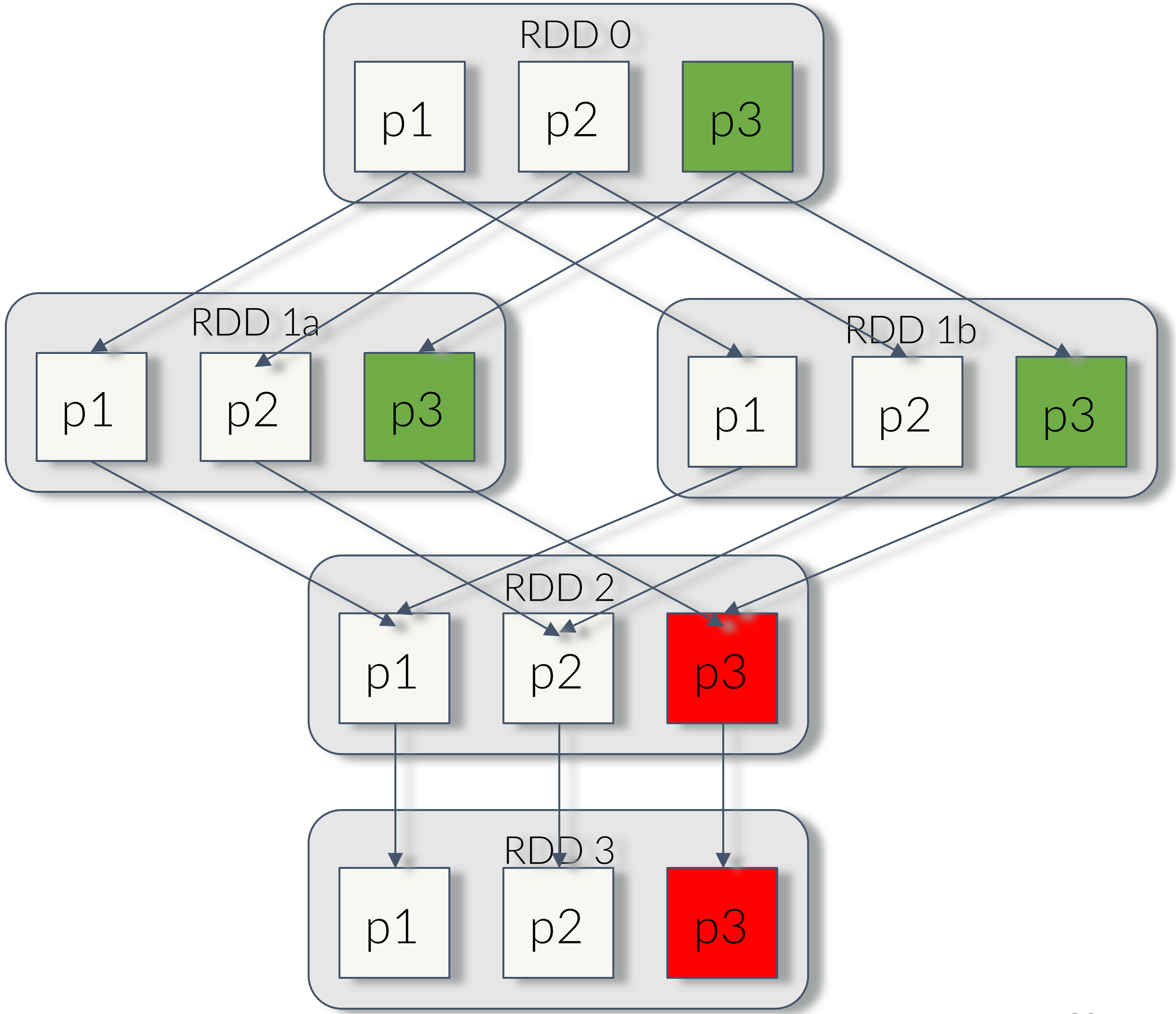
Coordinator
Worker 1
Worker 2



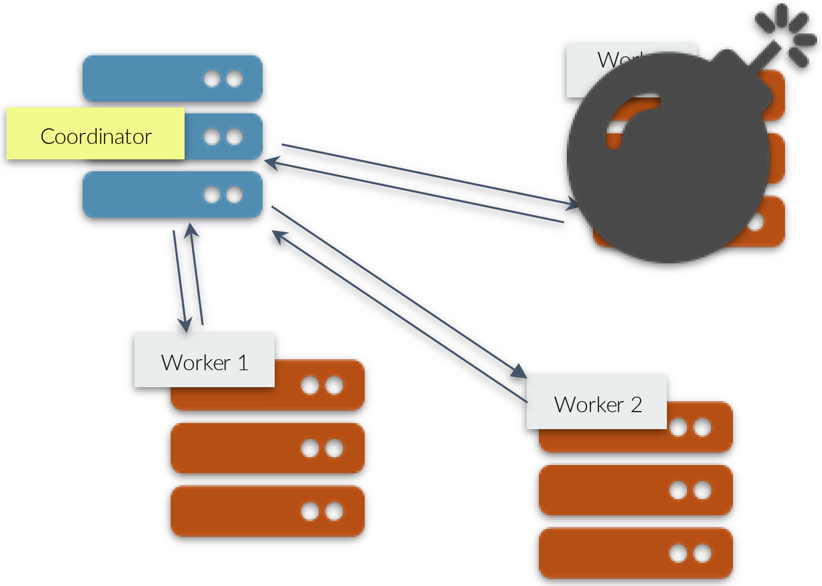
RDD 0
p1
p2
p3
RDD 1a
p1
p2
p3
RDD 1b
p1
p2
p3
RDD 2
p1
p2
p3
RDD 3
p1
p2
p3

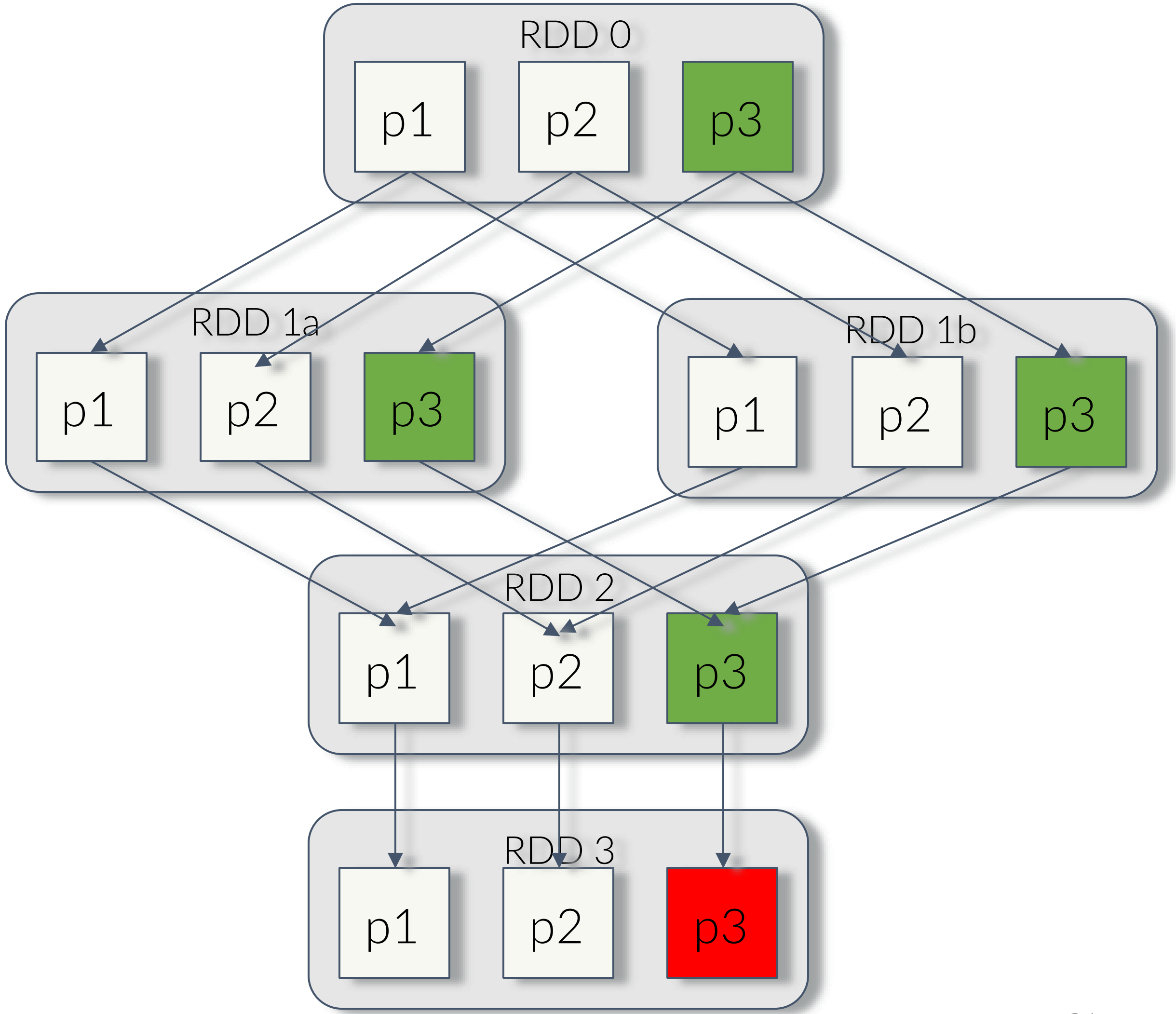
Example


Coordinator
Worker 1
Worker 2

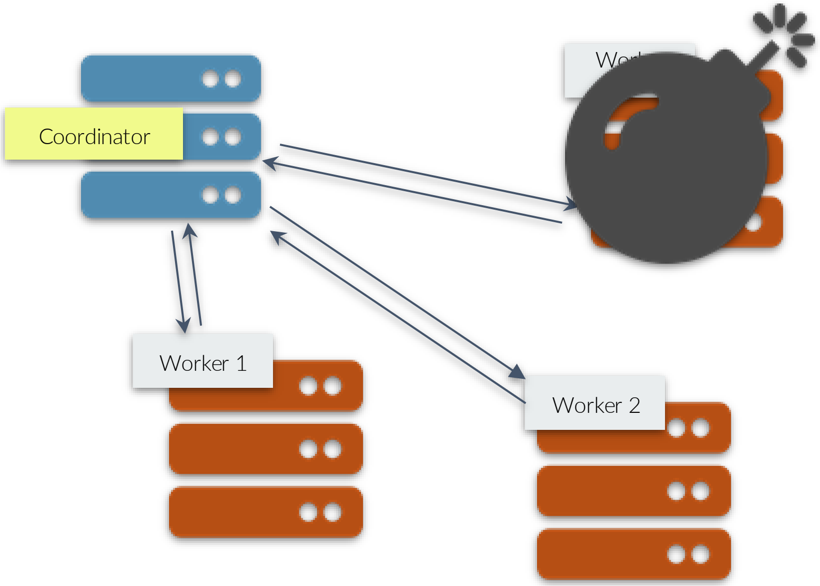

RDD 0
p1
p2
p3
RDD 1a
p1
p2
p3
RDD 1b
p1
p2
p3
RDD 2
p1
p2
p3
RDD 3
p1
p2
p3

Example

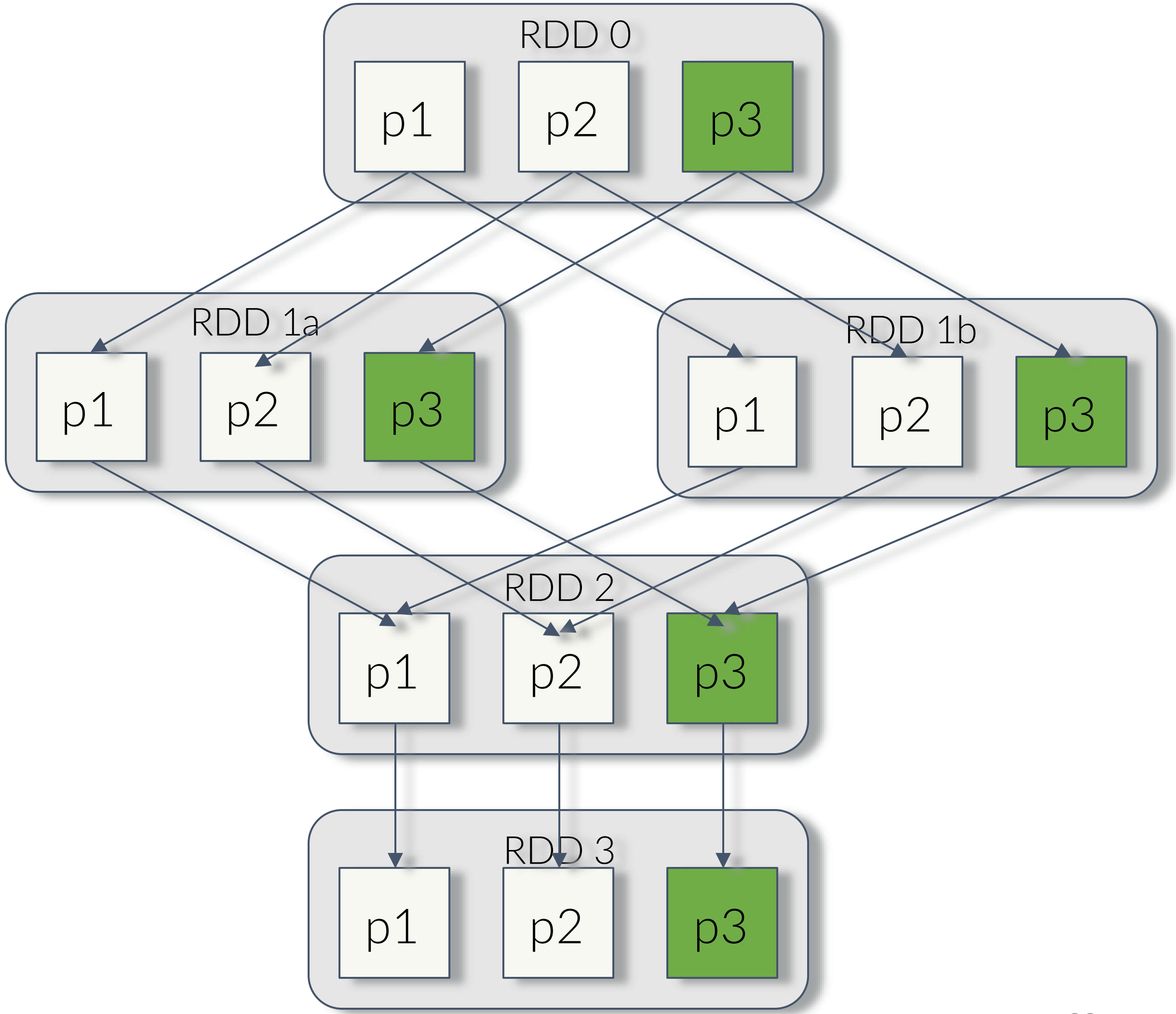

Coordinator
Worker 1
Worker 2


RDD 0
p1
p2
p3
RDD 1a
p1
p2
p3
RDD 1b
p1
p2
p3
RDD 2
p1
p2
p3
RDD 3
p1
p2
p3

Example



Coordinator
Worker 1
Worker 2



RDD 0
p1
p2
p3
RDD 1a
p1
p2
p3
RDD 1b
p1
p2
p3
RDD 2
p1
p2
p3
RDD 3
p1
p2
p3

RDDs

- User can control
  - Persistence: indicate storage strategy (e.g. in-memory)
  - Partitioning: placement optimization (e.g. hash partitioning)

Example : PageRank

TRY IT! Don't look at the next slide!
Try to write down pseudocode for implementing PageRank
In terms of join, map, and reduce

```
val links = spark.textFile(...).map
var ranks = // RDD of (URL, rank) pairs, initialized to 1
```

- PageRank is an iterative algorithm for computing rank (centrality) of web graph nodes

Damping factor

Page rank of p_i

$$PR(p_i) = \frac{1-d}{N} + d \sum_{p_j \in M(p_i)} \frac{PR(p_j)}{L(p_j)}$$

Number of links from p_j

Number of documents

Pages that link to p_i

- The PageRank paper shows that if you keep recomputing this value then the quantities will converge
- The size of the lineage graph depends on how many iterations you perform

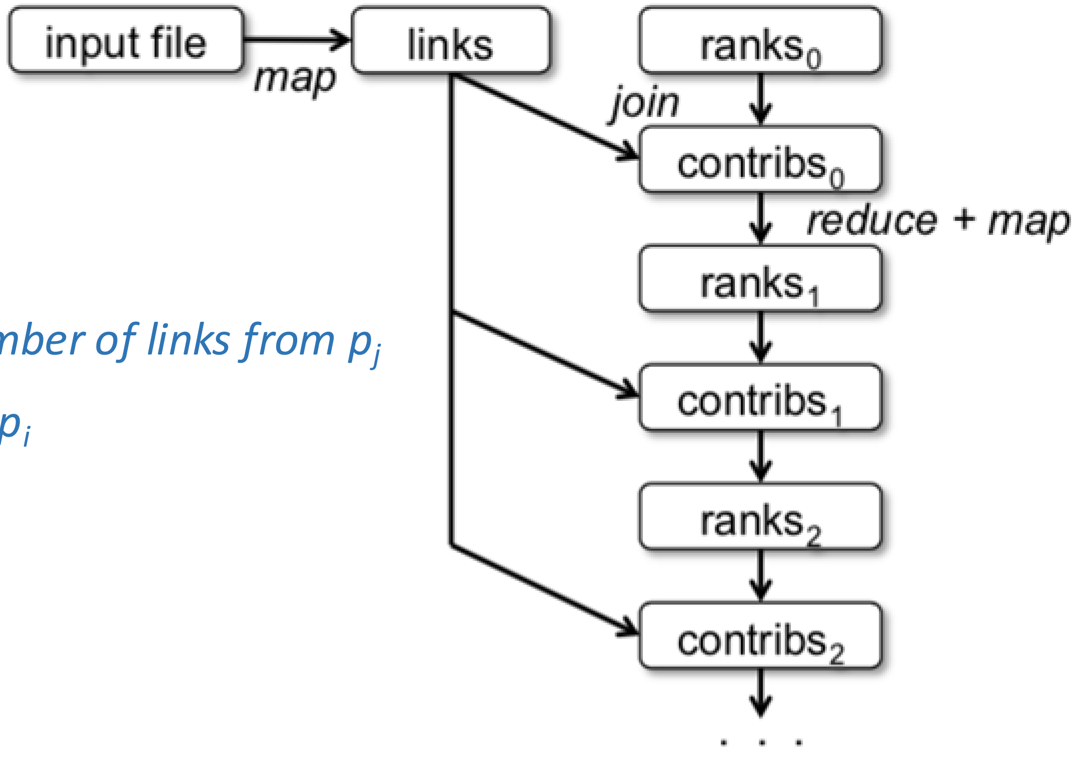
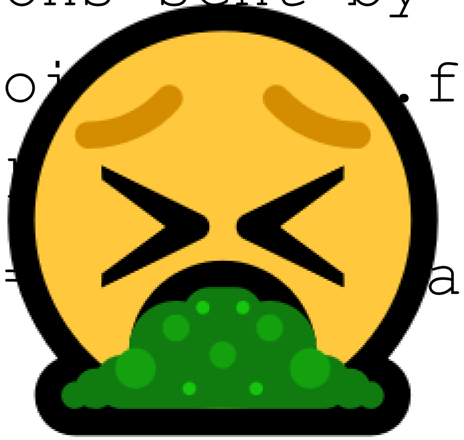


Figure 3: Lineage graph for datasets in PageRank.

```
val links = spark.textFile(...).map(...).persist()
var ranks = // RDD of (URL, rank) pairs, initialized to 1

for (i <- 1 to ITERATIONS) {
     // Build an RDD of (targetURL, float) pairs
     // with the contributions sent by each page
     val contribs = links.jo                .flatMap {
          (url, (links, ran
            links.map(dest =                ank/links.size))
     }
     // Sum contributions by URL and get new ranks
     ranks = contribs.reduceByKey((x,y) => x+y)
       .mapValues(sum => 1-d/N + (d)*sum)
}
```



$$PR(p_i) = \frac{1-d}{N} + d \sum_{p_j \in M(p_i)} \frac{PR(p_j)}{L(p_j)}$$

URL	Link
http://a	http://b
http://b	http://c
http://c	http://a
http://a	http://c

Ranks
(http://a,1)
(http://1,1)
(http://c,1)

(a, b) => (a + b)
Defines a function that takes two parameters a and b and sums them

```
val contribs = links.join(ranks).flatMap {
             (url, (links, rank)) =>
               links.map(dest => (dest, rank/links.size))
      }
```

URL	Links	Rank
http://a	{http://b, http://c}	1
http://b	{http://c}	1
http://c	{http://a}	1

(a, b) => (a + b)
Defines a function that takes two parameters a and b and sums them

```
val contribs = links.join(ranks).flatMap {
                 (url, (links, rank)) =>
                   links.map(dest => (dest, rank/links.size))
        }
```

URL	Links	Rank
http://a	{http://b, http://c}	1
http://b	{http://c}	1
http://c	{http://a}	1

$$PR(p_i) = \frac{1-d}{N} + d \sum_{p_j \in M(p_i)} \frac{PR(p_j)}{L(p_j)}$$

For each row in join

```
val contribs = links.join(ranks).flatMap {
               (url, (links, rank)) =>
                 links.map(dest => (dest, rank/links.size))
        }
```

Apply function to it

The function maps over link in row's links set

Example

http://a	**{http://b, http://c}**	**1**

Apply map() to both of these

{(http://b, 1/2), (http://c, 1/2)}

(a, b) => (a + b)
Defines a function that takes two parameters a and b and sums them

Inner map on each row

{(http://b, 1/2), (http://c, 1/2)}
{(http://c, 1)}
{(http://a, 1)}

Flattened

(http://b, 1/2)
(http://c, 1/2)
(http://c, 1)
(http://a, 1)

(http://b, .5)
(http://c, .5)
(http://c, 1)
(http://a, 1)

(a, b) => (a + b)
Defines a function that takes two parameters a and b and sums them

```
// Sum contributions by URL and get new ranks
     ranks = contribs.reduceByKey((x,y) => x+y)
       .mapValues(sum => 1-d/N + (d)*sum)
```

Apply x+y to combine rows that have the same key

Compute the weighted rank

(http://a, 1)
(http://b, .5)
(http://c, 1.5)

d = .7; 1-d/N = .1

(http://a, .1 + .7 = .8)
(http://b, .1 + .35 = .45)
(http://c, .1 + 1.05 = 1.15)

$$PR(p_i) = \frac{1-d}{N} + d \sum_{p_j \in M(p_i)} \frac{PR(p_j)}{L(p_j)}$$

New **ranks** table; b is weighted less because only a links to it, and a links to 2 pages. c is weighted more because both a and b link to it.

PageRank Challenges

What problems might we face, failure-wise, that we wouldn’t face if we wrote similar code with MapReduce?

Take a minute

What problems might we face, runtime-wise, if we implement this naïvely?

Take a minute

PageRank Challenges

What problems might we face, failure-wise, that we wouldn't face if we wrote similar code with MapReduce?

Very long lineage chain for ranks; slow. *Soln: Use explicit persistence to avoid having to regenerate* ***`ranks`*** *from lineage (not necessary for* ***`links`****)*

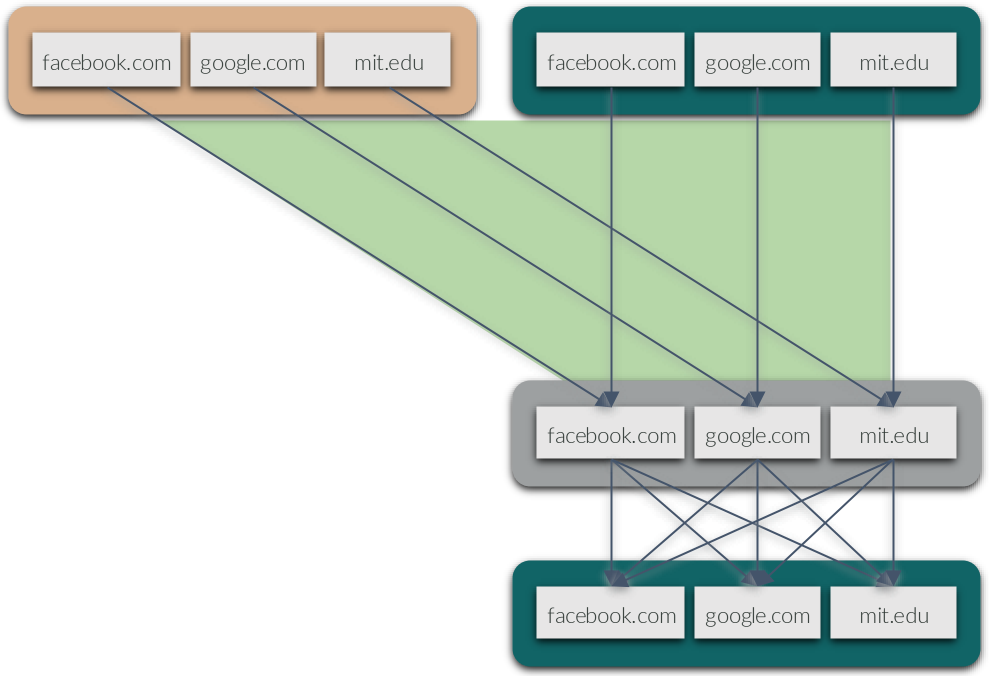
What problems might we face, runtime-wise, if we implement this naïvely?

Very slow joins*. Soln: Partition both links and ranks in the same way, so joins always happen on a single machine.*

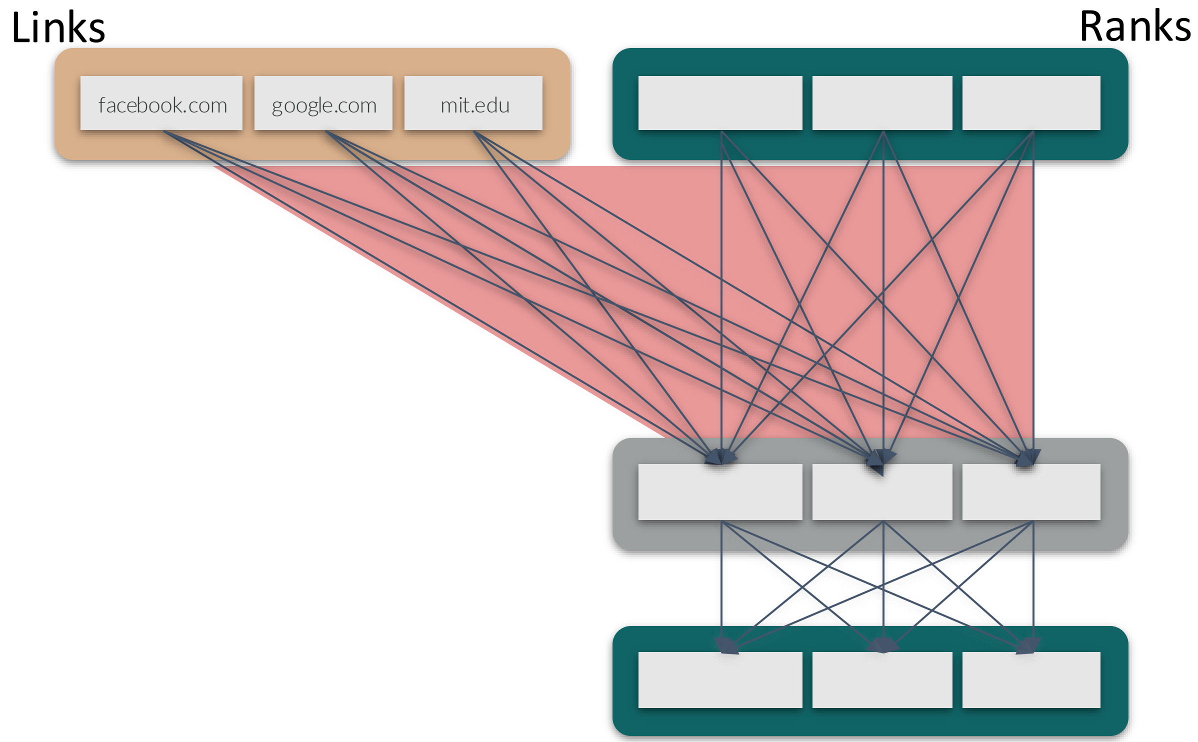
Use Spark support for controlling partitioning!

Example : PageRank

Partition rank and corresponding links on the same machine to eliminate cross-machine communication



PageRank with hash partitioning

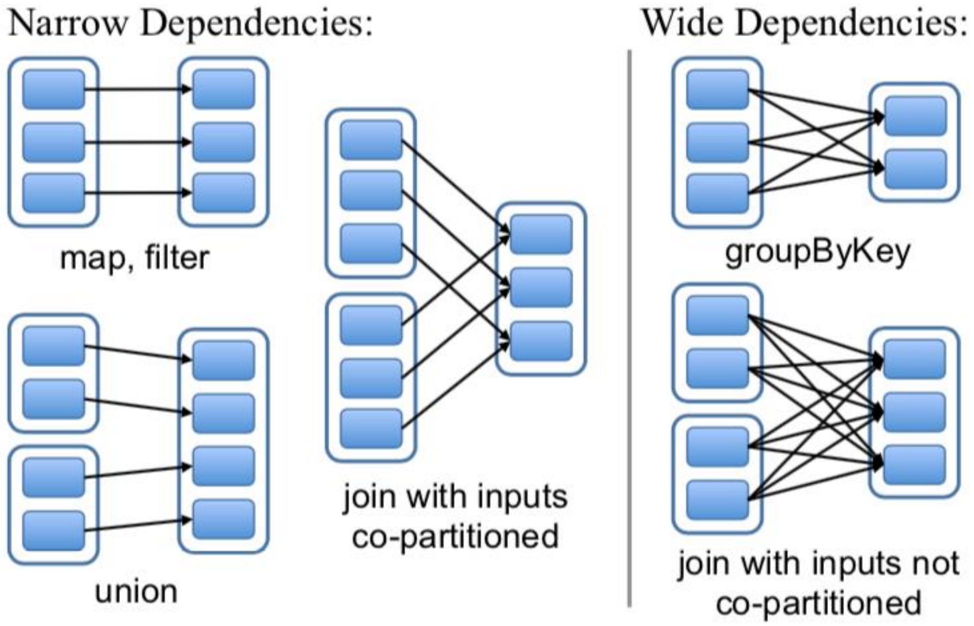


PageRank without partitioning

```
links = spark.textFile(...).map(...)
                .partitionBy(myPartFunc).persist()
```

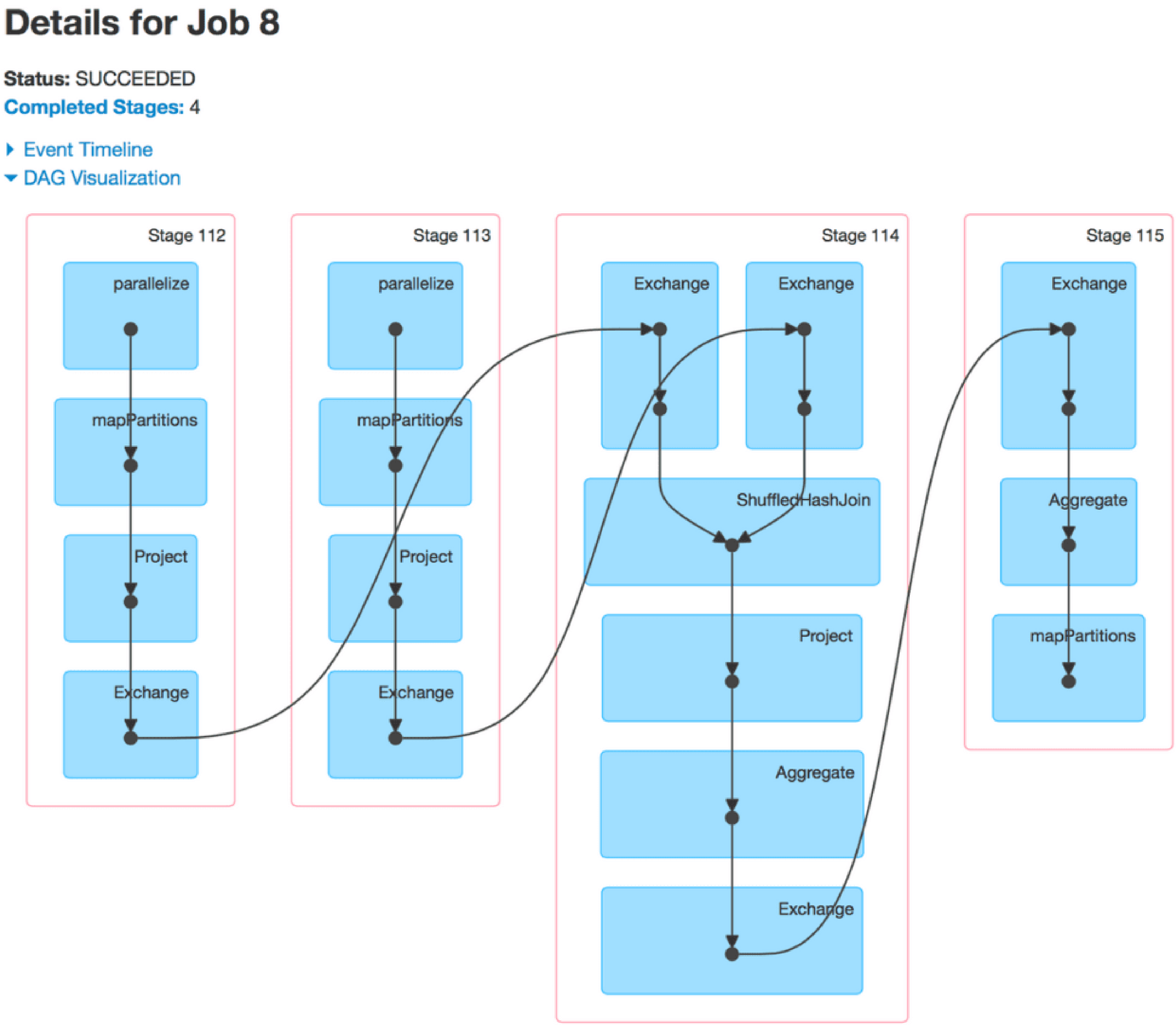
RDD Representation

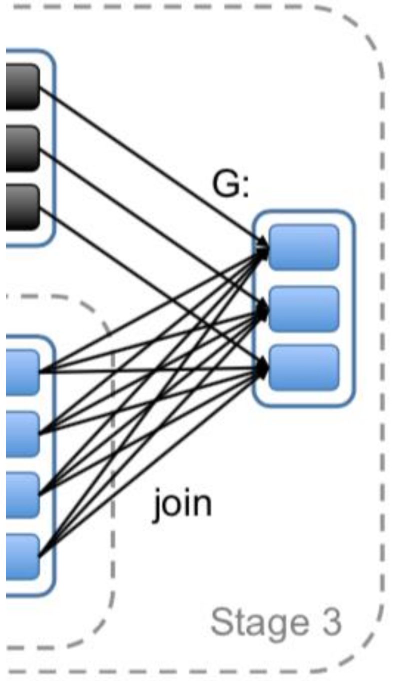
- Partitions: atomic pieces of the RDD
- Dependencies: relations with parent RDDs
  - Narrow Dependencies: A parent RDD partition is used by at most one child partition (e.g. map, filter). Can be pipelined
  - Wide Dependencies: A parent RDD partition is used by multiple child partitions (e.g. join, groupByKey). Need internode communication



Job

- Buil
  - E
    P
    C
  - S
- Assi
- Exe





Fault Tolerance

- Task failures
  - Stage's parents available: rerun on another node
  - Some stages unavailable: resubmit tasks to compute missing partitions in parallel
- Does not tolerate scheduler failures
  - Solution: Lineage graph replication

Memory Management

- Three storage strategies:
  - In-memory storage as deserialized Java objects,
    (fastest performance, since JVM can access each RDD element natively)

  - In-memory storage as serialized data,
    (more memory-efficient than Java object graphs, useful when space is limited)

  - On-disk storage
    (useful when RDDs are larger than RAM, but expensive to recompute from lineage)

- LRU policy for eviction at RDD level when there is not enough RAM
- Or, use user-specified "persistence priority" for eviction

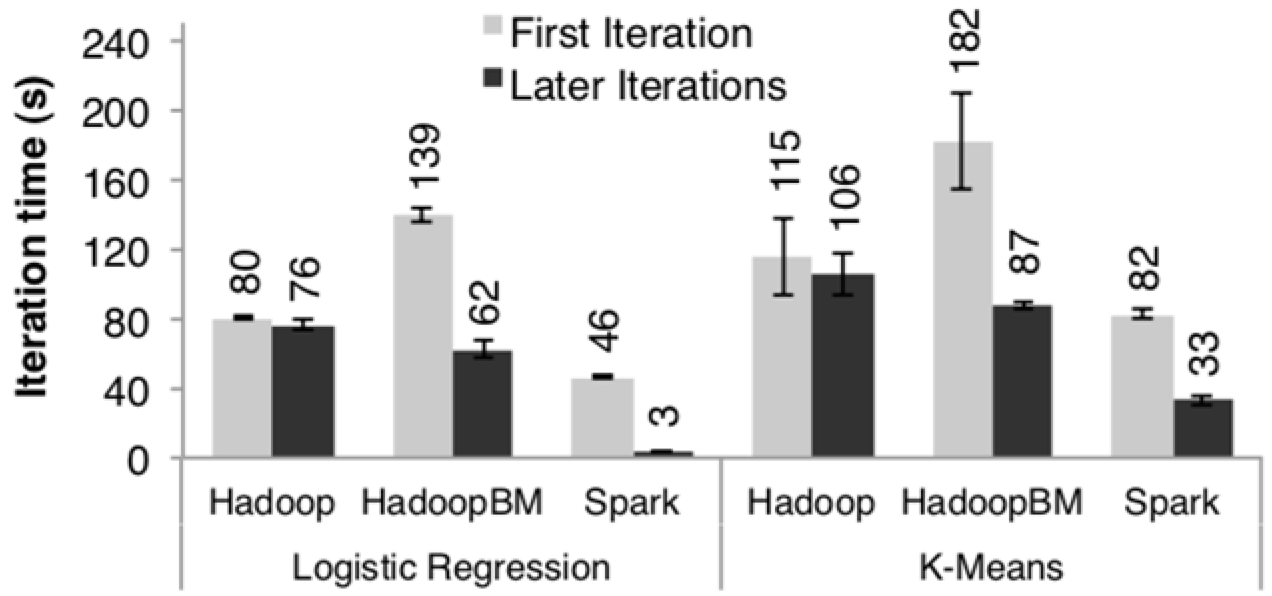
Checkpointing and Failures with Spark

- Short lineage chain?
  - Just recompute from lineage
- Long lineage chain with narrow dependencies?
  - Fast to recompute from lineage using pipelined execution
- Long lineage chain with wide dependencies?
  - This can be time-consuming. A node failure might require recomputing everything!
  - Use persistence as a checkpoint to prevent long recoveries

- A lot more is left to the user than with MapReduce or RDBMS

Performance: Iterative Machine Learning

- K-Means and Logistic Regression
- Experiment Setup:
  - 10 iterations
  - 10GB datasets
  - 25-100 machines



Note: HadoopBM in its first iteration converts text input data to a more efficient binary format

Performance: Iterative Machine Learning

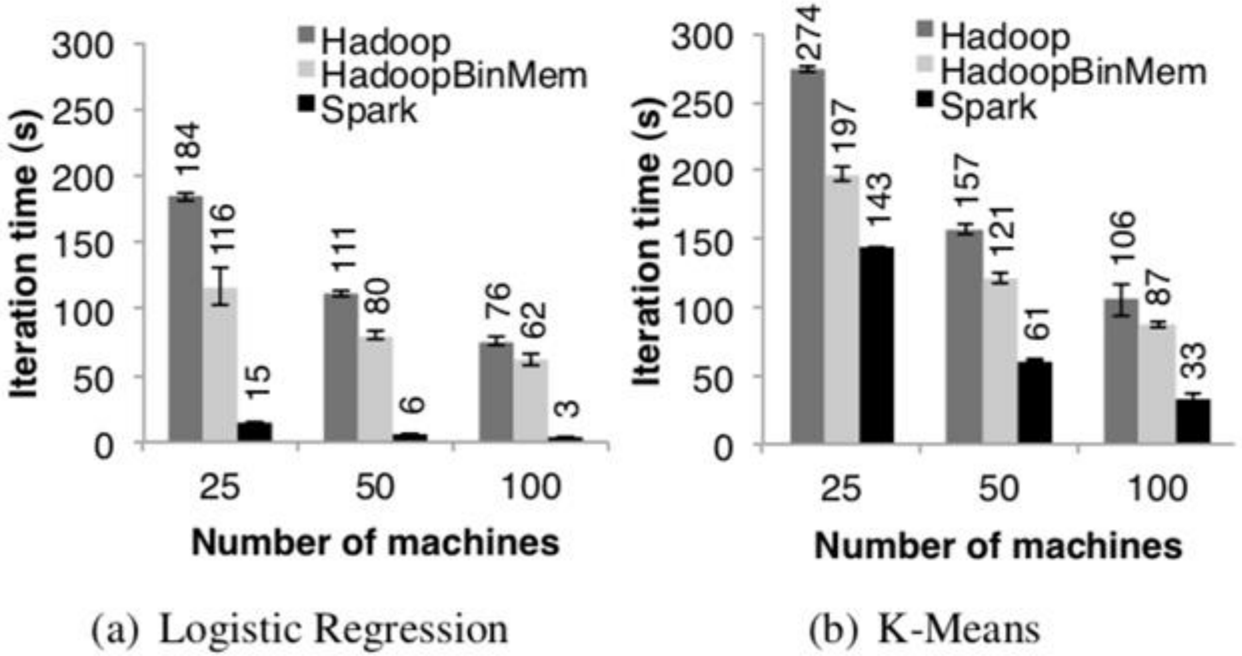


Figure 8: Running times for iterations after the first in Hadoop, HadoopBinMem, and Spark. The jobs all processed 100 GB.

Failure and Recovery

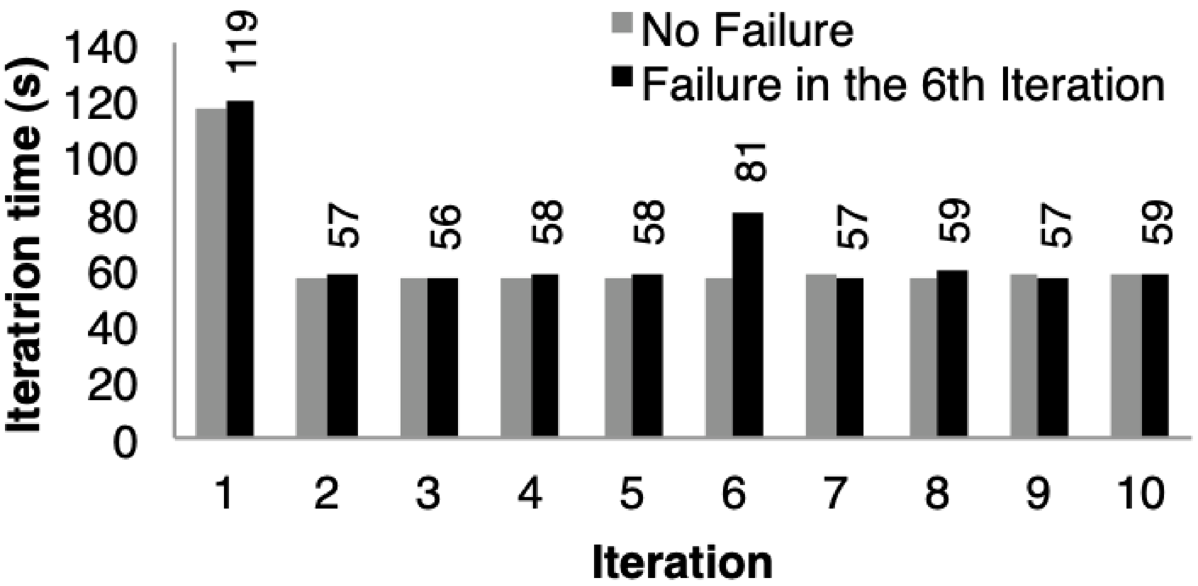


Figure 11: Iteration times for k-means in presence of a failure. One machine was killed at the start of the 6th iteration, resulting in partial reconstruction of an RDD using lineage.

Spark and MapReduce

- Spark has pretty much taken over “large-scale arbitrary compute jobs” from MapReduce
- Are there any advantages to MapReduce? Not really; you can express a MapReduce program almost exactly using Spark

Spark and RDBMS

- Spark doesn't have anything to say about transactions
- Spark has more optimization opportunities than MapReduce, but they're still mostly manual. Nothing like RDBMS optimizer (is it even possible with Spark?)
- Some room for exploiting RDBMS techniques, like joins
  - (Certainly, more room than with MapReduce)
- Scala programs or SQL queries?
- Spark SQL exists as SQL layer, much like Hive for MapReduce
- Likely prefer a RDBMS for updates or data that is re-accessed frequently.

Abstract representation of a RDBMS fighting the Spark system high fantasy photorealistic render