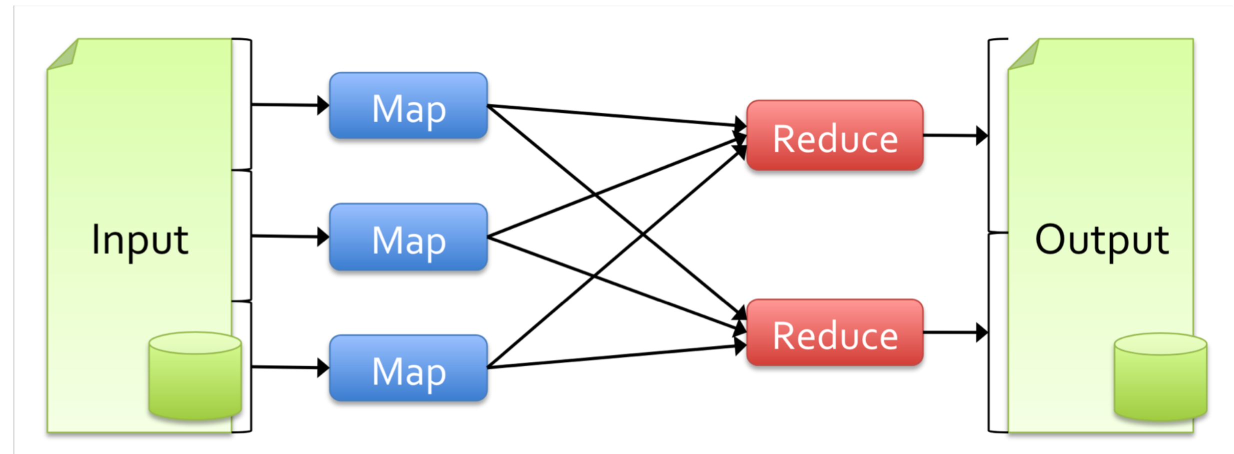
RESILIENT DISTRIBUTED DATASETS: A FAULT-TOLERANT ABSTRACTION FOR IN-MEMORY CLUSTER COMPUTING

MATEI ZAHARIA, MOSHARAF CHOWDHURY, TATHAGATA DAS, ANKUR DAVE, JUSTIN MA, MURPHY MCCAULEY, MICHAEL J. FRANKLIN, SCOTT SHENKER, ION STOICA

UNIVERSITY OF CALIFORNIA, BERKELEY

MOTIVATION

- At the time: MapReduce [3] was dominant
  - A restricted, two phase programming model
  - Poor support for in-memory computation

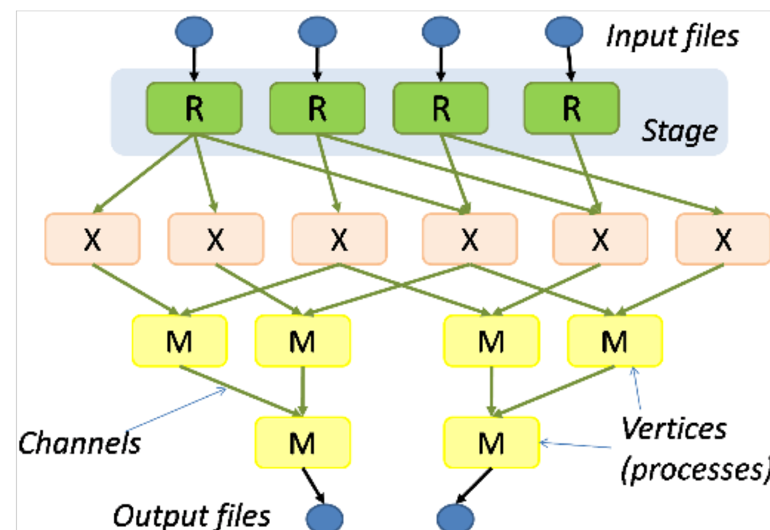


Source: https://spark.apache.org/talks/overview.pdf

- **Bad for *interactive analysis* and *iterative algorithms***

OTHER IDEAS: DRYAD AND CIEL

- Dryad [1, 2]: use arbitrary DAGs



Source: https://www.microsoft.com/en-us/research/project/dryad/

- Ciel [4]: better support for iterative and recursive algorithms

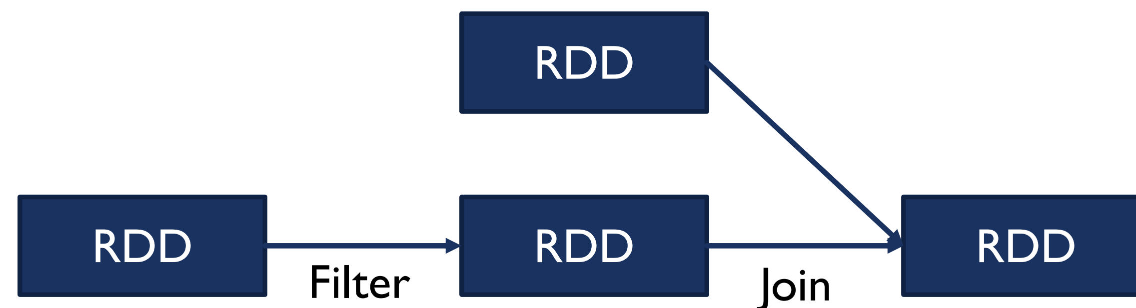
WHAT IS AN RDD?

- “a read only, partitioned collection of records”
- Create one from:
  1. Data in stable storage (e.g. HDFS)
  2. Applying transformations such as filter, map or join to other RDDs

THE KEY IDEA FOR FAULT-TOLERANCE

- Record the ***lineage*** of an RDD
- i.e. keep the DAG of transformations applied to your base RDDs

- RDDs can be re-computed by retracing the steps in the DAG



WHY IS THIS BETTER?

- Previous shared-memory systems relied on replication to achieve fault tolerance
- Replication is expensive

IN-MEMORY COMPUTATION

- RDDs can be kept in-memory
- Trade-offs against distributed shared memory (DSM)
  - No arbitrary updates (**immutability**)

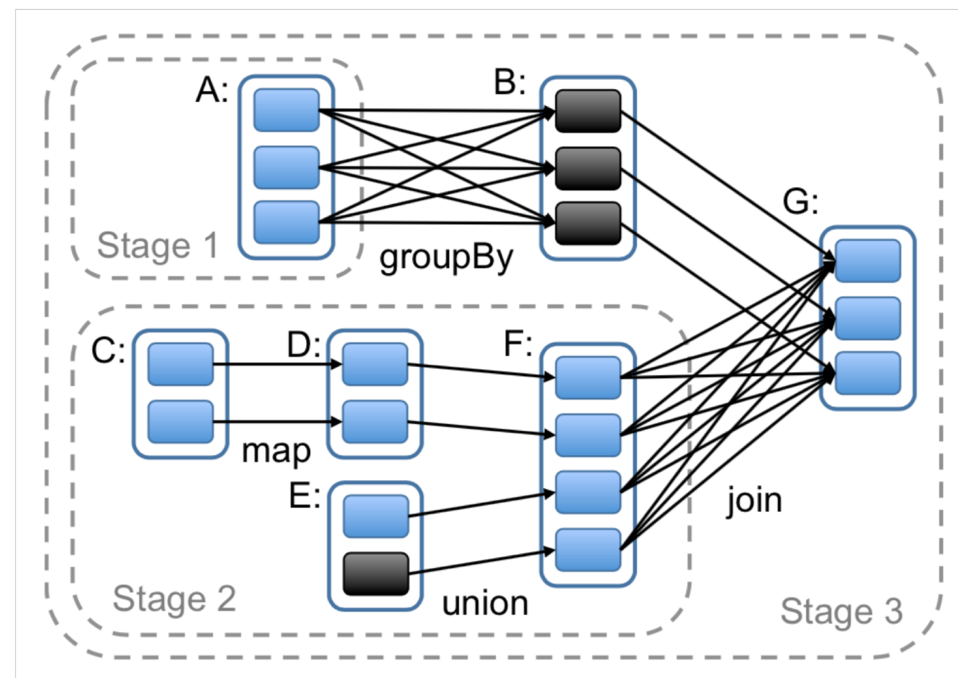
- Advantages:
  1. Allows lineage to work
  2. Can run backup copies of jobs
  3. Can schedule based on data-locality

THE PROGRAMMING MODEL

- **Transformations** are *lazy* operations used to build the DAG
  - e.g. map, filter, reduce, sample, join, groupBy, sort, etc
- **Actions** launch the computation and return a result to the programmer
  - e.g. count, collect, save
- General – can express MapReduce in Spark

NARROW AND WIDE DEPENDENCIES

- Narrow – each partition of the parent RDD is used by at most one partition of the child RDD
- Wide – can't exploit pipelining / data-locality
  - Implement a *shuffle* stage like MapReduce



EXAMPLE – PAGERANK

```
// Load graph as an RDD of (URL, outlinks) pairs
val links = spark.textFile(...)
              .map(...) // parse
              .persist() // keep in memory


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

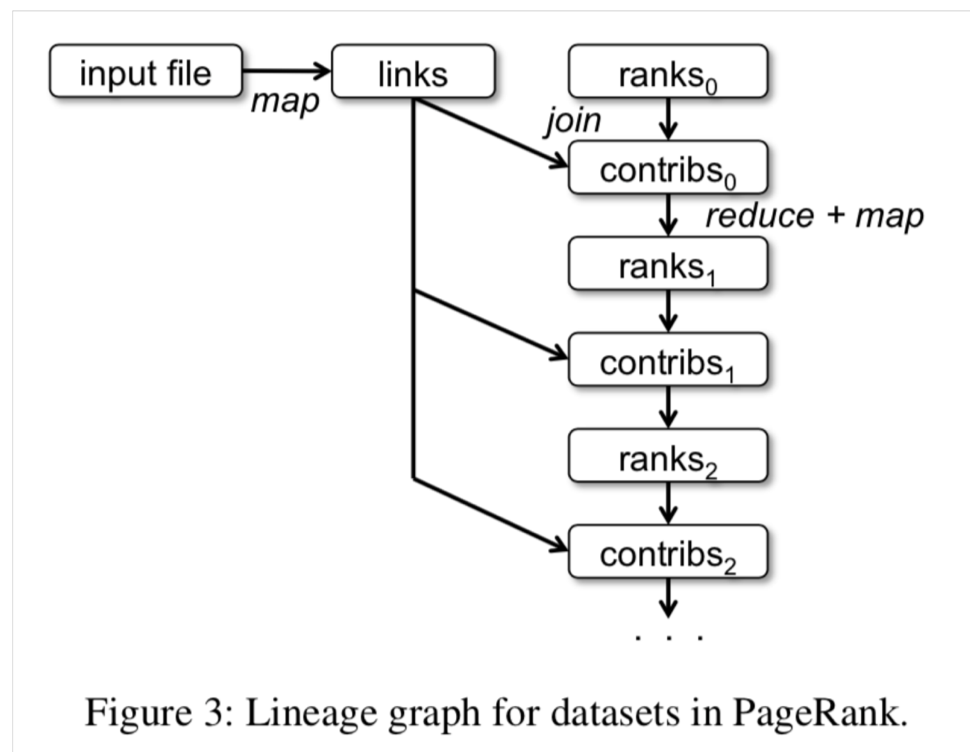


Figure 3: Lineage graph for datasets in PageRank.

PERFORMANCE – NODE FAILURE

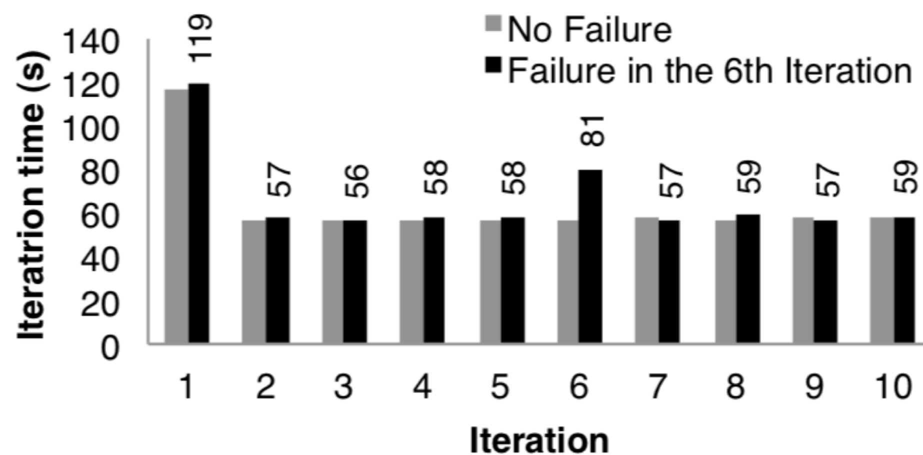


Figure 11: Iteration times for k-means in presence of a failure. One machine was killed at the start of the 6th iteration, resulting in partial reconstruction of an RDD using lineage.

- Loss of tasks and partitions on a node
- Run in parallel on other nodes to recover lost partitions

PERFORMANCE – ITERATIVE ALGORITHMS

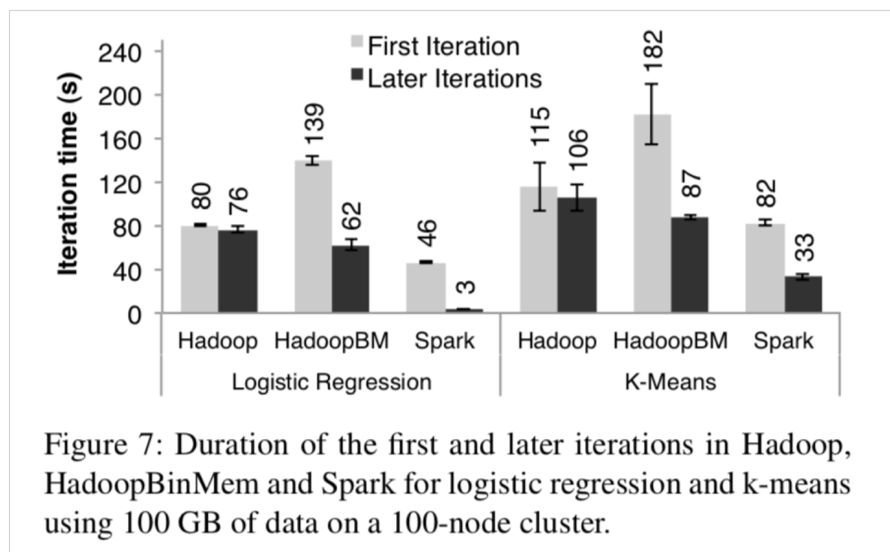


Figure 7: Duration of the first and later iterations in Hadoop, HadoopBinMem and Spark for logistic regression and k-means using 100 GB of data on a 100-node cluster.

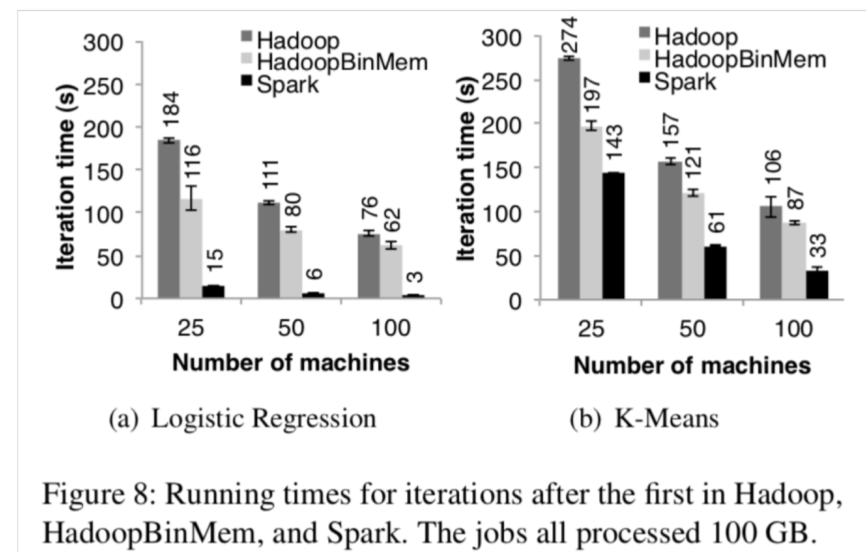


Figure 8: Running times for iterations after the first in Hadoop, HadoopBinMem, and Spark. The jobs all processed 100 GB.

- HadoopBM – store data in lower overhead format with in-memory HDFS

- First iteration – lower protocol overhead vs Hadoop
- Subsequent iterations – deserialization is expensive for HadoopBM!

- K-Means more compute-limited

PERFORMANCE – BIG DATASETS AND INTERACTIVITY

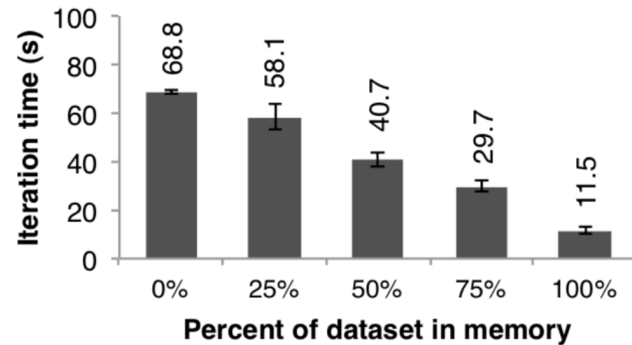


Figure 12: Performance of logistic regression using 100 GB data on 25 machines with varying amounts of data in memory.

- Sensible degradation of performance as dataset exceeds available memory

- Interactivity – can get query results within seconds (vs minutes for Hadoop)
  - Hadoop needed 25s to do a no-op in the paper!

TAKEAWAYS

- Replication is expensive – serialization, IO
- A broader programming model than MapReduce is practical
- In-memory caching is effective
- Making memory immutable allows lineage fault-tolerance

CRITICISMS

1. Lots of tuning – manually control partitioning and memory-persistence
2. Only one contrived experiment on fault recovery time
3. Batching as the default assumption
4. Low level programming model – can't have automatic optimisation

REFERENCES

[1]
Y. Yu *et al.*, 'DryadLINQ: A System for General-purpose Distributed Data-parallel Computing Using a High-level Language', in *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, Berkeley, CA, USA, 2008, pp. 1–14.

[2]
M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, 'Dryad: Distributed Data-Parallel Programs from Sequential Building Blocks', p. 14.

[3]
J. Dean and S. Ghemawat, 'MapReduce: simplified data processing on large clusters', *Communications of the ACM*, vol. 51, no. 1, p. 107, Jan. 2008.

[4]
D. G. Murray, M. Schwarzkopf, C. Smowton, S. Smith, A. Madhavapeddy, and S. Hand, 'CIEL: a universal execution engine for distributed data-flow computing', p. 14.

[5]
M. Zaharia *et al.*, 'Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing', p. 14.