EECS 262a
Advanced Topics in Computer Systems
Lecture 15

PDBMS / Spark
March 14th, 2016

John Kubiatowicz
Electrical Engineering and Computer Sciences
University of California, Berkeley

http://www.eecs.berkeley.edu/~kubitron/cs262

Today's Papers

- Parallel Database Systems: The Future of High Performance Database Systems
  Dave DeWitt and Jim Gray. Appears in *Communications of the ACM,* Vol. 32, No. 6, June 1992
- Spark: Cluster Computing with Working Sets
  M. Zaharia, M. Chowdhury, M.J. Franklin, S. Shenker and I. Stoica. Appears in Proceedings of *HotCloud 2010*, June 2010.
  - M. Zaharia, et al, Resilient Distributed Datasets: A fault-tolerant abstraction for in-memory cluster computing, NSDI 2012.

- Monday: Column Store DBs
- Wednesday: Comparison of PDBMS, CS, MR

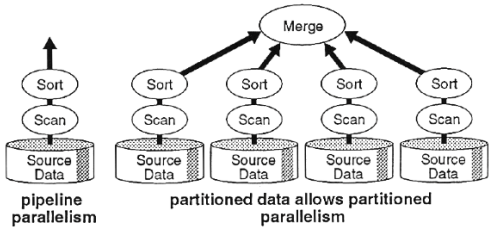
- Thoughts?

DBMS Historical Context

- Mainframes traditionally used for large DB/OLTP apps
  - Very expensive: $25,000/MIPS, $1,000/MB RAM
- Parallel DBs: "An idea whose time has passed" 1983 paper by DeWitt
- Lots of dead end research into specialized storage tech
  - CCD and bubble memories, head-per-track disks, optical disks
- Disk throughput doubling, but processor speeds growing faster
  - Increasing processing – I/O gap
- 1992: Rise of parallel DBs – why??

Parallel Databases

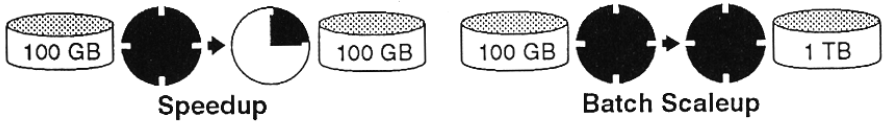
- Side benefit of Relational Data Model: parallelism
  - Relational queries are trivially parallelizable
  - Uniform operations applied to uniform streams of data
- Tv



  - Pipeline: can sort in parallel with scan's output
  - Isolated partitions are ideal for parallelism

Speedup and Scaleup

- Broad issues in many types of parallel systems



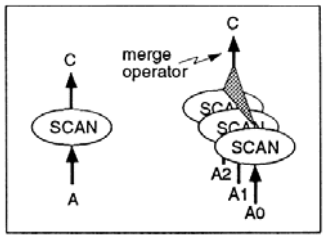
- Speedup (fixed problem size) vs. Scaleup (problem &HW grow)
  - Linear Speedup – N-times larger system yields a speedup of N
  - Linear Scaleup – N-times larger system runs N-times larger job in same time as original system
  - Near-Linear Scaleup – "ignore" overhead of non-linear operators (e.g., *nlog*(*n*) sort yields deviation from linear scaleup)
- Barriers to parallelism: Startup, Interference and Skew
  - Startup: scheduling delays
  - Interference: overhead of shared resource contention
  - Skew: Variance in partitioned operator service times

Hardware Architectures

- Insight: *Build parallel DBs using cheap microprocessors*
  - Many microprocessors >> one mainframe
  - $250/MIPS, $100/MB RAM << $25,000/MIP, $1,000/MB RAM
- Three HW architectures (Figs 4 and 5):
  - Shared-memory: common global mem and disks (IBM/370, DEC VAX)
    - » Limited to 32 CPUs, have to code/schedule cache affinity
  - Shared-disks: private memory, common disks (DEC VAXcluster)
    - » Limited to small clusters, have to code/schedule system affinity
  - Shared-nothing: private memory and disks, high-speed network interconnect (Teradata, Tandem, nCUBE)
    - » 200+ node clusters shipped, 2,000 node Intel hypercube cluster
    - » Hard to program completely isolated applications (except DBs!)
- Real issue is interference (overhead)
  - 1% overhead limits speedup to 37x: 1,000 node cluster has 4% effective power of single processor system!

DB Dataflow Model

- The rise of SQL!
  - Relational data model operators take relations as input and produce relations as output
  - Iterators and pipelines enable operators to be composed into dataflow graphs
  - Dataflow graph operations can be executed in parallel



- Porting applications is trivial
  - No work required to yield near-linear speedups and scaleups
  - This is why DBs adopted large-scale parallel processing much earlier than systems

Hard Challenges

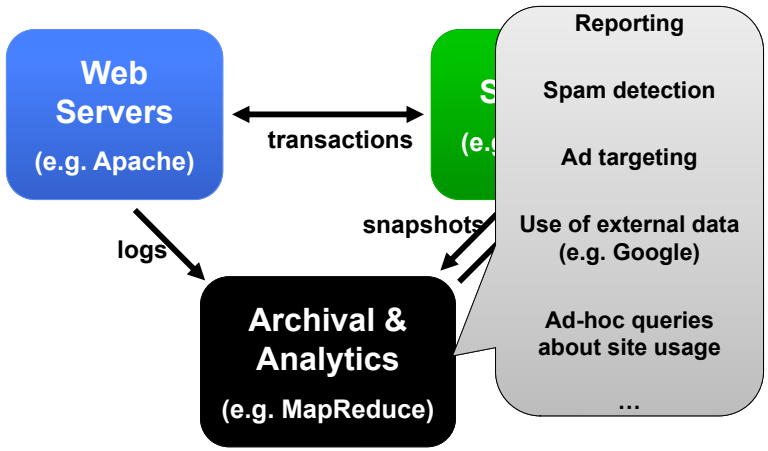
- DB layout – partitioning data across machines
  - Round-Robin partitioning: good for reading entire relation, bad for associative and range queries by an operator
  - Hash-partitioning: good for assoc. queries on partition attribute and spreads load, bad for assoc. on non-partition attribute and range queries
  - Range-partitioning: good for assoc. accesses on partition attribute and range queries but can have hot spots (data skew or execution skew) if uniform partitioning criteria
- Choosing parallelism within a relational operator
  - Balance amount of parallelism versus interference
- Specialized parallel relational operators
  - Sort-merge and Hash-join operators
- Other "hard stuff": query optimization, mixed workloads, UTILITIES!

Is this a good paper?

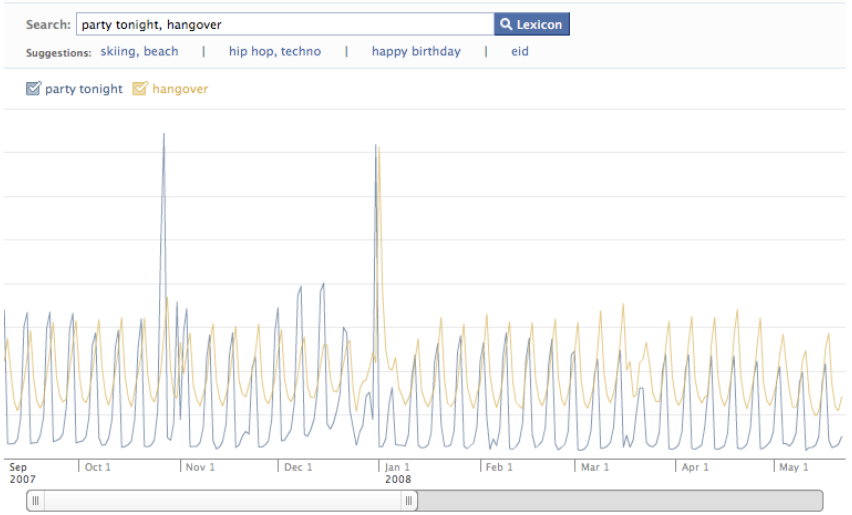
- What were the authors' goals?
- What about the evaluation/metrics?
- Did they convince you that this was a good system/approach?
- Were there any red-flags?
- What mistakes did they make?
- Does the system/approach meet the "Test of Time" challenge?
- How would you review this paper today?

BREAK

Typical Web Application

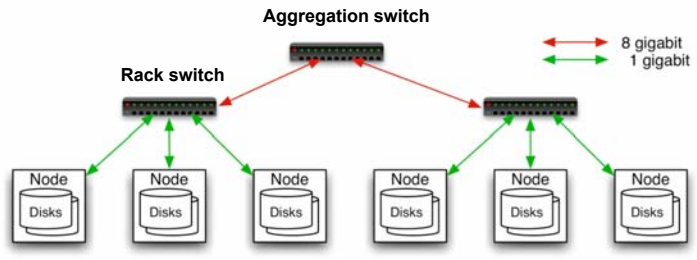


Example: Facebook Lexicon



www.facebook.com/lexicon (now defunct)

Typical Hadoop (MapReduce) Cluster



- 40 nodes/rack, 1000-4000 nodes in cluster
- 1 Gbps bandwidth in rack, 8 Gbps out of rack
- Node specs (Facebook):
  8-16 cores, 32 GB RAM, 8×1.5 TB disks, no RAID

Challenges

- *Cheap nodes fail, especially when you have many*
  - Mean time between failures for 1 node = 3 years
  - MTBF for 1000 nodes = 1 day
  - Implication: Applications must tolerate faults
- *Commodity network = low bandwidth*
  - Implication: Push computation to the data
- *Nodes can also "fail" by going slowly (execution skew)*
  - Implication: Application must tolerate & avoid stragglers

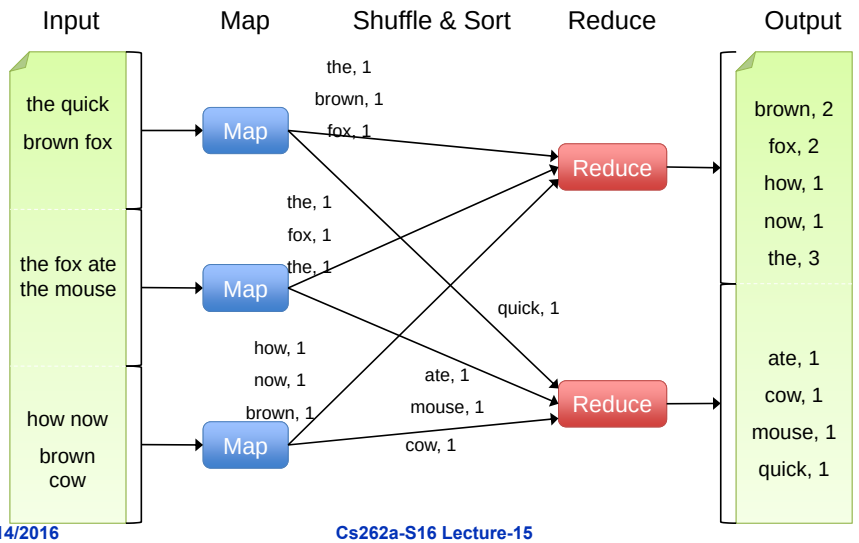
MapReduce

- First widely popular programming model for data-intensive apps on commodity clusters
- Published by Google in 2004
  - Processes 20 PB of data / day
- Popularized by open-source Hadoop project
  - 40,000 nodes at Yahoo!, 70 PB at Facebook

MapReduce Programming Model

- Data type: key-value *records*
- Map function:
  $(K_{in}, V_{in}) \rightarrow list(K_{inter}, V_{inter})$
- Reduce function:
  $(K_{inter}, list(V_{inter})) \rightarrow list(K_{out}, V_{out})$

Word Count Execution



MapReduce Execution Details

- Mappers preferentially scheduled on same node or same rack as their input block
  - Minimize network use to improve performance
- Mappers save outputs to local disk before serving to reducers
  - Allows recovery if a reducer crashes

Fault Tolerance in MapReduce

1. If a task crashes:
   - Retry on another node
     - » OK for a map because it had no dependencies
     - » OK for reduce because map outputs are on disk
   - If the same task repeatedly fails, fail the job

➢ Note: For fault tolerance to work, tasks must be *deterministic* and *side-effect-free*

Fault Tolerance in MapReduce

2. If a node crashes:
   - Relaunch its current tasks on other nodes
   - Relaunch any maps the node previously ran
     - » Necessary because their output files are lost

Fault Tolerance in MapReduce

3. If a task is going slowly – straggler (execution skew):
   - Launch second copy of task on another node
   - Take output of whichever copy finishes first

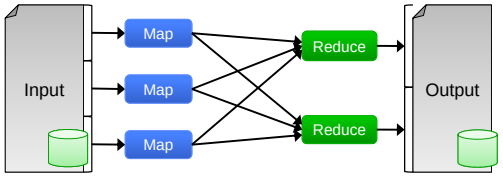
- Critical for performance in large clusters

Takeaways

- By providing a data-parallel programming model, MapReduce can control job execution in useful ways:
  - Automatic division of job into tasks
  - Placement of computation near data
  - Load balancing
  - Recovery from failures & stragglers

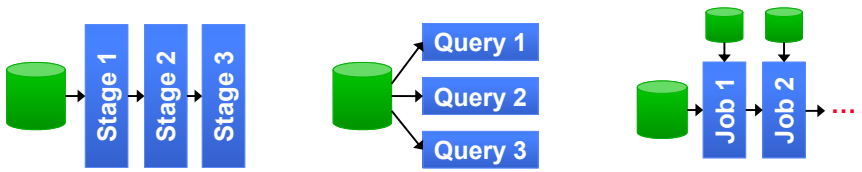
Issues with MapReduce

- Hard to express more complex programs
  - E.g. word count + a sort to find the top words
  - Need to write many different map and reduce functions that are split up all over the program
  - Must write complex operators (e.g. join) by hand



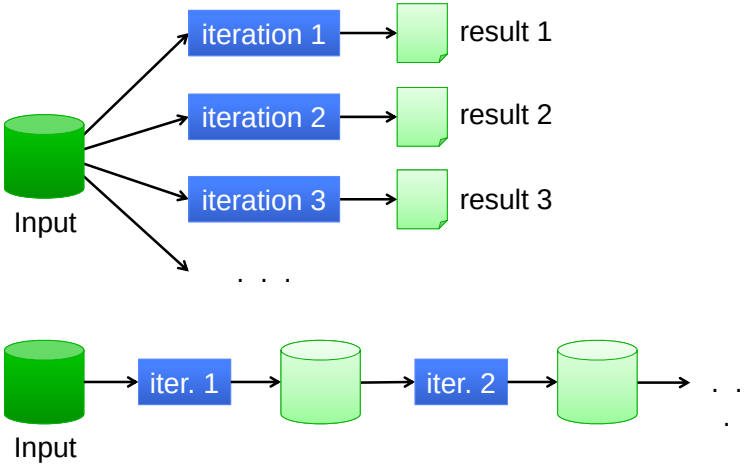
Issues with MapReduce

- *Acyclic data flow* from stable storage to stable storage ➔ poor support for applications that need to *reuse* pieces of data (I/O bottleneck and compute overhead)
  - Iterative algorithms (e.g. machine learning, graphs)
  - Interactive data mining (e.g. Matlab, Python, SQL)
  - Stream processing (e.g., continuous data mining)

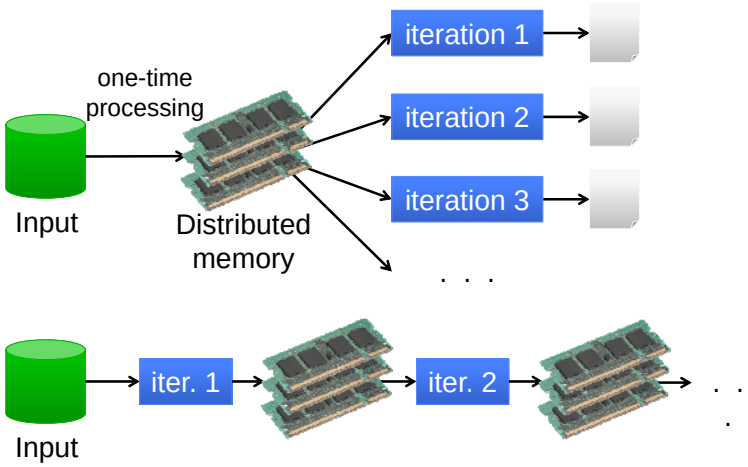


Iterative job **Interactive mining** **Stream processing**

Example: Iterative Apps



Goal: Keep Working Set in RAM



Spark Goals

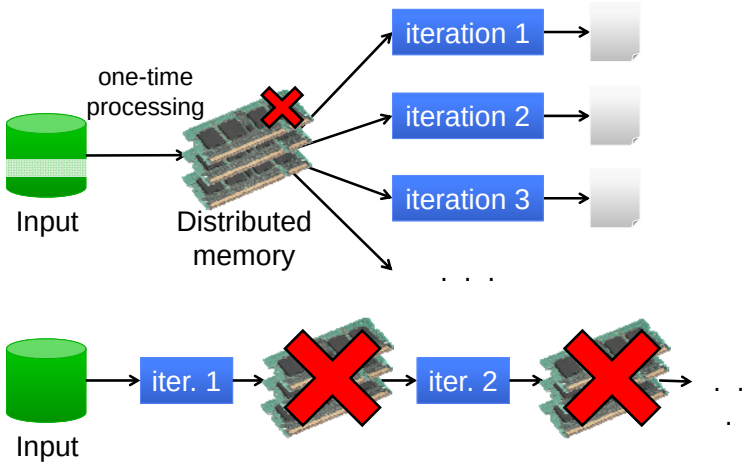
- Support iterative and stream jobs (apps with data reuse) efficiently:
  - Let them keep data in memory
- Experiment with programmability
  - Leverage Scala to integrate cleanly into programs
  - Support interactive use from Scala interpreter
- Retain MapReduce's fine-grained fault-tolerance and automatic scheduling benefits of MapReduce

Spark
Lightning-Fast Cluster Computing

Key Idea: Resilient Distributed Datasets (RDDs)

- *Restricted* form of distributed shared memory
  - Read-only (immutable), partitioned collections of records
  - Caching hint tells system to retain RDD in memory
  - Can only be created through deterministic transformations (map, group-by, join, …)
- Allows efficient implementation & recovery
  - Key idea: rebuild lost data using *lineage*
  - Enables hint model that allows memory to be reused if necessary
  - No cost if nothing fails
- Rich enough to capture many models:
  - **Data flow models**: MapReduce, Dryad, SQL, …
  - **Specialized models** for iterative apps: Pregel, Hama, …

RDD Recovery



Programming Model

- Driver program
  - Implements high-level control flow of an application
  - Launches various operations in parallel
- Resilient distributed datasets (RDDs)
  - Immutable, partitioned collections of objects
  - Created through parallel *transformations* (map, filter, groupBy, join, …) on data in stable storage
  - Can be *cached* for efficient reuse
- Parallel actions on RDDs
  - Foreach, reduce, collect
- Shared variables
  - Accumulators (add-only), Broadcast variables (read-only)

Parallel Operations

- *reduce* – Combines dataset elements using an associative function to produce a result at the driver program
- *collect* – Sends all elements of the dataset to the driver program (e.g., update an array in parallel with *parallelize*, *map*, and *collect*)
- *foreach* – Passes each element through a user provided function
- No grouped *reduce* operation

Shared Variables

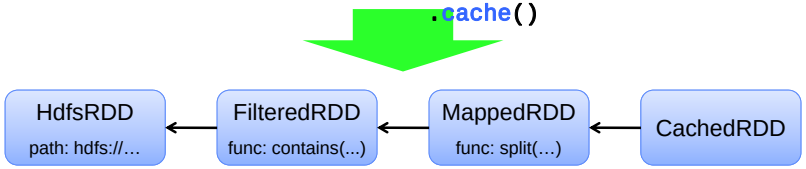
- Broadcast variables
  - Used for large read-only data (e.g., lookup table) in multiple parallel operations – distributed once instead of packaging with every closure
- Accumulators
  - Variables that works can only “add” to using an associative operation, and only the driver program can read

RDD Fault Tolerance

RDDs maintain *lineage* information that can be used to reconstruct lost partitions

Ex:

```
cachedMsgs = textFile(...).filter(_.contains("error"))
                          .map(_.split('\t')(2))
                          .cache()
```



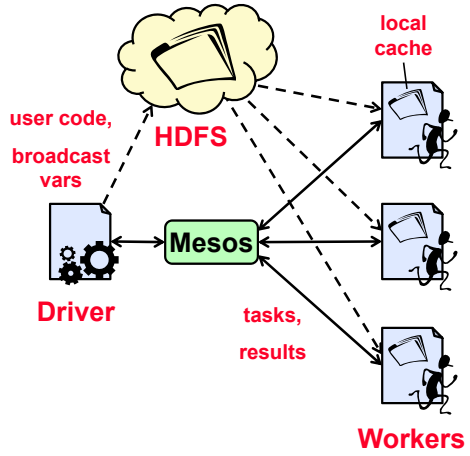
Architecture

Driver program connects to Mesos and schedules tasks

Workers run tasks, report results and variable updates

Data shared with HDFS/NFS

No communication between workers for now



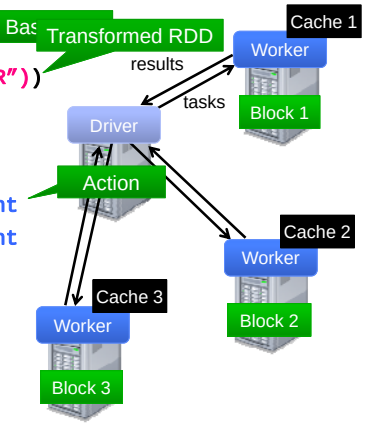
Spark Version of Word Count

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

file.flatMap(line => line.split(" "))
    .map(word => (word, 1))
    .reduceByKey(_ + _)
```


Spark Version of Log Mining

Load error messages from a log into memory, then interactively search for various patterns

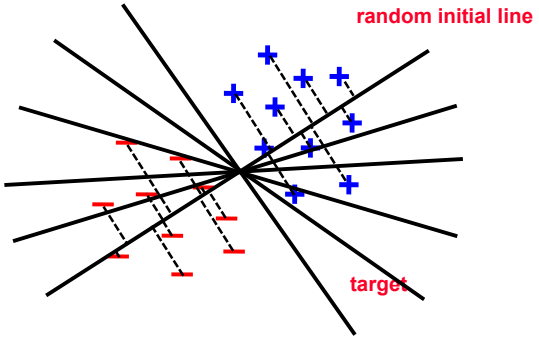
```
lines = spark.textFile("hdfs://...")
errors = lines.filter(_.startsWith("ERROR"))
messages = errors.map(_.split('\t')(2))
cachedMsgs = messages.cache()

cachedMsgs.filter(_.contains("foo")).count
cachedMsgs.filter(_.contains("bar")).count
. . .
```



Result: scaled to 1 TB data in 5-7 sec (vs 170 sec for on-disk data)

Logistic Regression

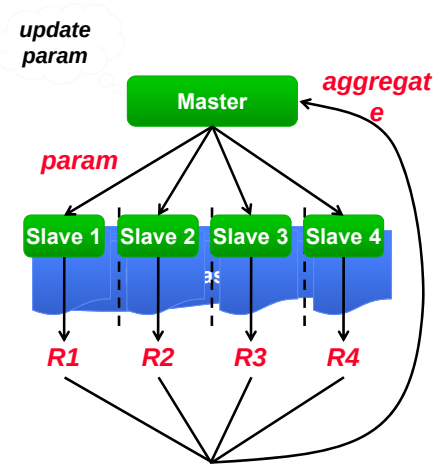
Goal: find best line separating two sets of points



Example: Logistic Regression

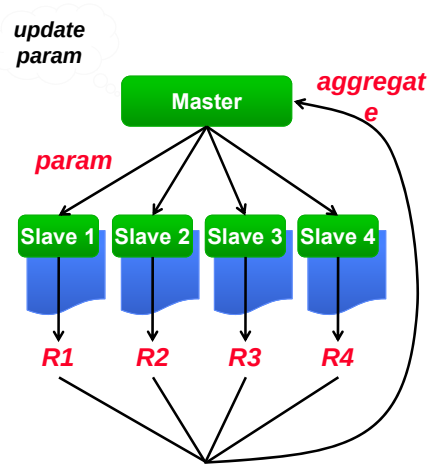
```
val data = spark.textFile(...).map(readPoint).cache()

var w = Vector.random(D)

for (i <- 1 to ITERATIONS) {
  val gradient = data.map(p =>
    (1 / (1 + exp(-p.y*(w dot p.x))) - 1) * p.y * p.x
  ).reduce(_ + _)
  w -= gradient
}

println("Final w: " + w)
```


Job Execution

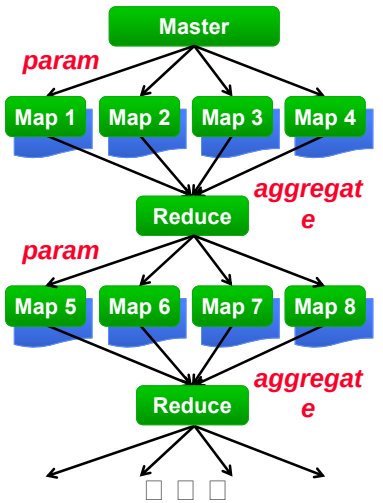


Spark

Job Execution

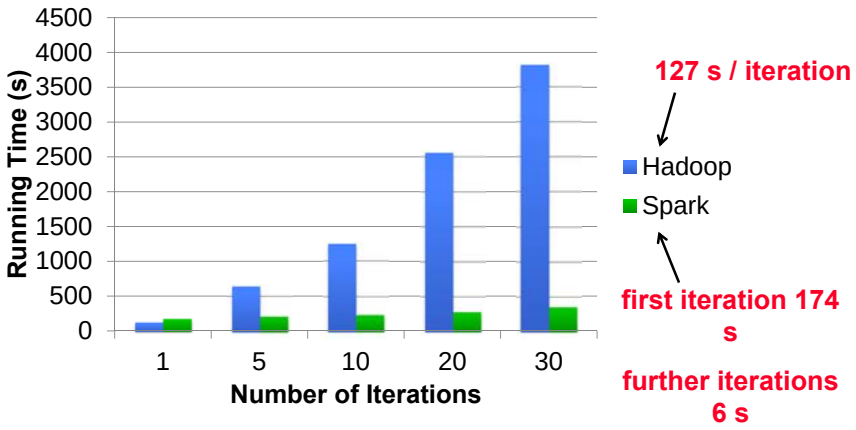


Spark



Hadoop / Dryad

Performance



Interactive Spark

Modified Scala interpreter to allow Spark to be used interactively from the command line

Required two changes:

- Modified wrapper code generation so that each "line" typed has references to objects for its dependencies
- Place generated classes in distributed filesystem

Enables in-memory exploration of big data

What RDDs are Not Good For

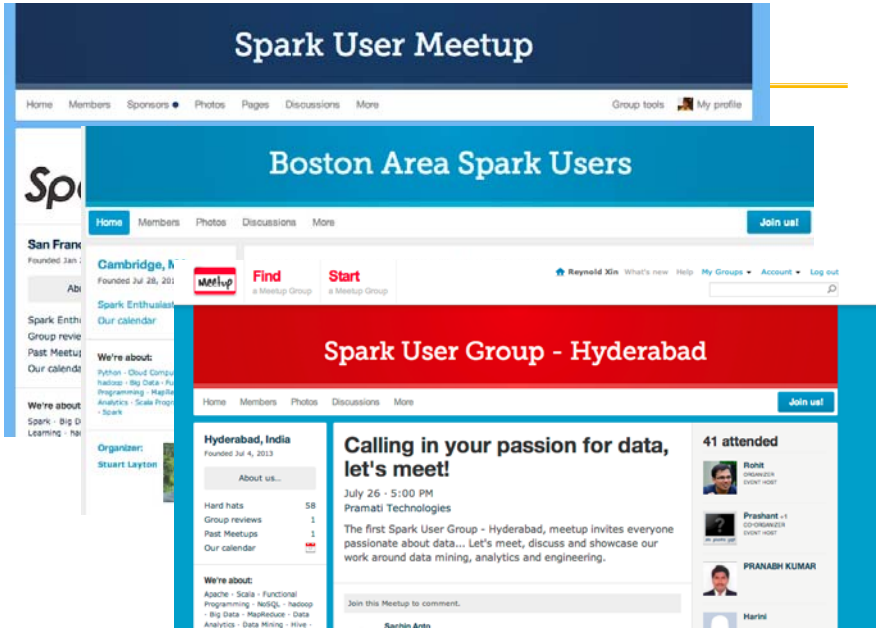
- RDDs work best when an application applies the same operation to many data records
  - Our approach is to just log the operation, not the data
- Will not work well for apps where processes asynchronously update shared state
  - Storage system for a web application
  - Parallel web crawler
  - Incremental web indexer (e.g. Google's Percolator)

Milestones

- 2010: Spark open sourced
- Feb 2013: Spark Streaming alpha open sourced
- Jun 2013: Spark entered Apache Incubator
- Aug 2013: Machine Learning library for Spark

Frameworks Built on Spark

- MapReduce
- HaLoop
  - Iterative MapReduce from UC Irvine / U Washington
- Pregel on Spark (Bagel)
  - Graph processing framework from Google based on BSP message-passing model
- Hive on Spark (Shark)
  - In progress



Summary

- Spark makes distributed datasets a first-class primitive to support a wide range of apps
- RDDs enable efficient recovery through lineage, caching, controlled partitioning, and debugging

Meta Summary

- Three approaches to parallel and distributed systems
  - Parallel DBMS
  - Map Reduce variants (Spark, …)
  - Column-store DBMS (Monday 3/28)
- Lots of on-going “discussion” about best approach
  - We’ll have ours on Wednesday

Is this a good paper?

- What were the authors' goals?
- What about the evaluation/metrics?
- Did they convince you that this was a good system/approach?
- Were there any red-flags?
- What mistakes did they make?
- Does the system/approach meet the "Test of Time" challenge?
- How would you review this paper today?