EECS 262a
Advanced Topics in Computer Systems
Lecture 21

Comparison of Parallel DB, CS, MR and Spark
November 6th, 2023

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

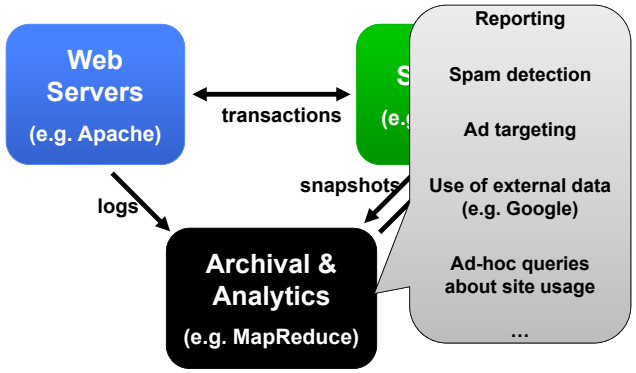
http://www.eecs.berkeley.edu/~kubitron/cs262

Today's Papers

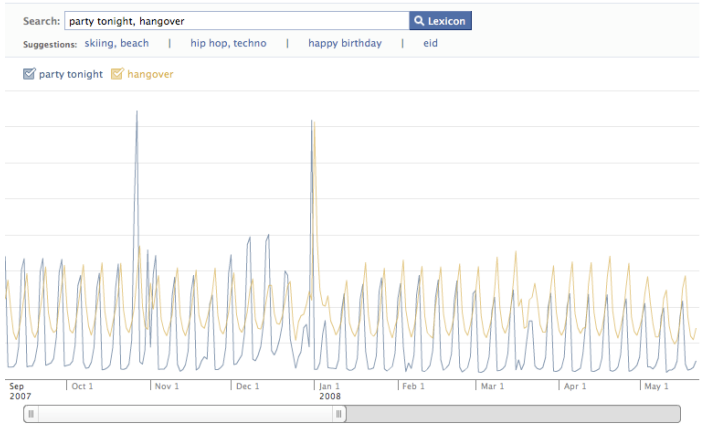
- A Comparison of Approaches to Large-Scale Data Analysis
  Andrew Pavlo, Erik Paulson, Alexander Rasin, Daniel J. Abadi, David J. DeWitt, Samuel Madden, Michael Stonebraker. Appears in *Proceedings of the ACM SIGMOD International Conference on Management of Data,* 2009
- Spark: Cluster Computing with 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.

- Thoughts?

Typical Web Application

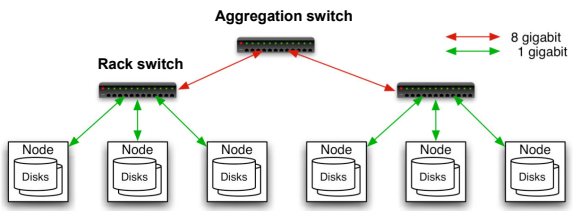


Example: Facebook Lexicon



www.facebook.com/lexicon (now defunct)

Sample 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

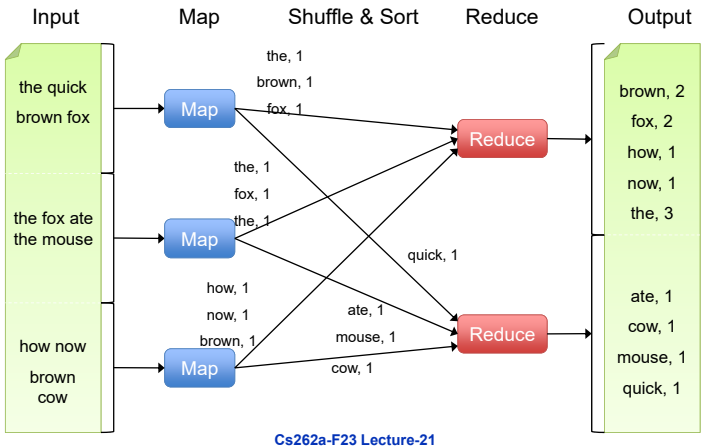
- Data type: key-value *records*
- Map function:

$$(K_{in}, V_{in}) \rightarrow list(K_{inter}, V_{inter})$$

- 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
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
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

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

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

Comparisons between
Parallel DBMS and MapReduce

Two Approaches to Large-Scale Data Analysis

- Both approaches are "Shared nothing"
- MapReduce
  - Distributed file system
  - Map, Split, Copy, Reduce
  - MR scheduler
- Parallel DBMS
  - Standard relational tables, (physical location transparent)
  - Data are partitioned over cluster nodes
  - SQL
  - Join processing: T1 joins T2
    - » If T2 is small, copy T2 to all the machines
    - » If T2 is large, hash partition T1 and T2 and send partitions to different machines (this is similar to the split-copy in MapReduce)
  - Query Optimization
  - Intermediate tables not materialized by default

Architectural Differences

	Parallel DBMS	MapReduce
Schema Support	O	X
Indexing	O	X
Programming Model	Stating what you want (SQL)	Presenting an algorithm (C/C++, Java, …)
Optimization	O	X
Flexibility	Spotty UDF Support	Good
Fault Tolerance	Not as Good	Good
Node Scalability	<100	>10,000

Schema Support

- MapReduce
  - Flexible, programmers write code to interpret input data
  - Good for single application scenario
  - Bad if data are shared by multiple applications. Must address data syntax, consistency, etc.
- Parallel DBMS
  - Relational schema required
  - Good if data are shared by multiple applications

Programming Model & Flexibility

- MapReduce
  - Low level: "We argue that MR programming is somewhat analogous to Codasyl programming…"
  - "Anecdotal evidence from the MR community suggests that there is widespread sharing of MR code fragments to do common tasks, such as joining data sets."
  - very flexible
- Parallel DBMS
  - SQL
  - user-defined functions, stored procedures, user-defined aggregates

Indexing

- MapReduce
  - No native index support
  - Programmers can implement their own index support in Map/Reduce code
  - But hard to share the customized indexes in multiple applications
- Parallel DBMS
  - Hash/b-tree indexes well supported

Execution Strategy & Fault Tolerance

- MapReduce
  - Intermediate results are saved to local files
  - If a node fails, run the node-task again on another node
  - At a mapper machine, when multiple reducers are reading multiple local files, there could be large numbers of disk seeks, leading to poor performance.
- Parallel DBMS
  - Intermediate results are pushed across network
  - If a node fails, must re-run the entire query

Avoiding Data Transfers

- MapReduce
  - Schedule Map close to data
  - But other than this, programmers must avoid data transfers themselves
- Parallel DBMS
  - A lot of optimizations
  - Such as determine where to perform filtering

Node Scalability

- MapReduce
  - 10,000's of commodity nodes
  - 10's of Petabytes of data
- Parallel DBMS
  - <100 expensive nodes
  - Petabytes of data

Performance Benchmarks

- Benchmark Environment
- Original MR task (Grep)
- Analytical Tasks
  - Selection
  - Aggregation
  - Join
  - User-defined-function (UDF) aggregation

Node Configuration

- 100-node cluster
  - Each node: 2.40GHz Intel Core 2 Duo, 64-bit red hat enterprise Linux 5 (kernel 2.6.18) w/ 4Gb RAM and two 250GB SATA HDDs.
- Nodes interconnected with Cisco Catalyst 3750E 1Gb/s switches
  - Internal switching fabric has 128Gbps
  - 50 nodes per switch
- Multiple switches interconnected via 64Gbps Cisco StackWise ring
  - The ring is only used for cross-switch communications.

Tested Systems

- Hadoop (0.19.0 on Java 1.6.0)
  - HDFS data block size: 256MB
  - JVMs use 3.5GB heap size per node
  - "Rack awareness" enabled for data locality
  - Three replicas w/o compression: Compression or fewer replicas in HDFS does not improve performance
- DBMS-X (a parallel SQL DBMS from a major vendor)
  - Row store
  - 4GB shared memory for buffer pool and temp space per node
  - Compressed table (compression often reduces time by 50%)
- Vertica
  - Column store
  - 256MB buffer size per node
  - Compressed columns by default

Benchmark Execution

- Data loading time:
  - Actual loading of the data
  - Additional operations after the loading, such as compressing or building indexes
- Execution time
  - DBMS-X and vertica:
    - » Final results are piped from a shell command into a file
  - Hadoop:
    - » Final results are stored in HDFS
    - » An *additional* Reduce job step to combine the multiple files into a single file

Performance Benchmarks

- Benchmark Environment
- Original MR task (Grep)
- Analytical Tasks
  - Selection
  - Aggregation
  - Join
  - User-defined-function (UDF) aggregation

Task Description

- From MapReduce paper
  - Input data set: 100-byte records
  - Look for a three-character pattern
  - One match per 10,000 records
- Varying the number of nodes
  - Fix the size of data per node (535MB/node)
  - Fix the total data size (1TB)

Data Loading

- Hadoop:
  - Copy text files into HDFS in parallel
- DBMS-X:
  - Load SQL command executed in parallel: it performs hash partitioning and distributes records to multiple machines
  - Reorganize data on each node: compress data, build index, perform other housekeeping
    - » This happens in parallel
- Vertica:
  - Copy command to load data in parallel
  - Data is re-distributed, then compressed

Data Loading Times

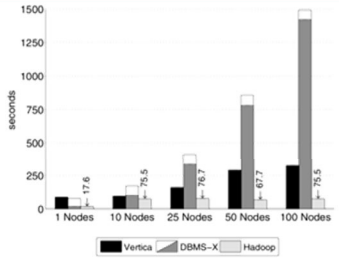


Figure 1: Load Times – Grep Task Data Set (535MB/node)

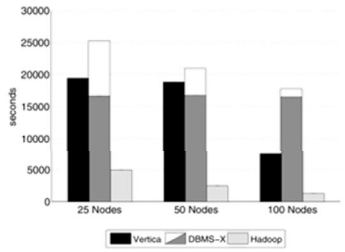


Figure 2: Load Times – Grep Task Data Set (1TB/cluster)

- DBMS-X: grey is loading, white is re-organization after loading
  - Loading is actually sequential despite parallel load commands
- Hadoop does better because it only copies the data to three HDFS replicas

Execution

- SQL:

```
SELECT * FROM data WHERE field LIKE "%XYZ%"
```

  - Full table scan
- MapReduce:
  - Map: pattern search
  - No reduce
  - An additional Reduce job to combine the output into a single file

Execution time

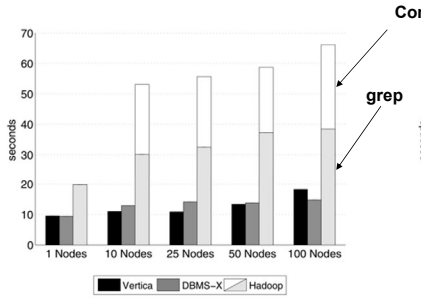


Figure 4: Grep Task Results – 535MB/node Data Set

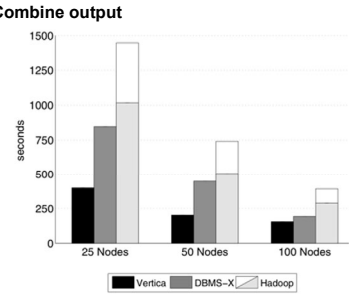


Figure 5: Grep Task Results – 1TB/cluster Data Set

- Hadoop's large start-up cost shows up in Figure 4, when data per node is small
- Vertica's good data compression

Performance Benchmarks

- Benchmark Environment
- Original MR task (Grep)
- Analytical Tasks
  - Selection
  - Aggregation
  - Join
  - User-defined-function (UDF) aggregation

Input Data

- Input #1: random HTML documents
  - Inside an html doc, links are generated with Zipfian distribution
  - 600,000 unique html docs with unique urls per node
- Input #2: 155 million UserVisits records
  - 20GB/node
- Input #3: 18 million Ranking records
  - 1GB/node

Selection Task

- Find the pageURLs in the rankings table (1GB/node) with a pageRank > threshold
  - 36,000 records per data file (very selective)
- SQL:

```
SELECT pageURL, pageRank
FROM Rankings WHERE pageRank > X;
```

- MR: single Map, no Reduce

Selection Task

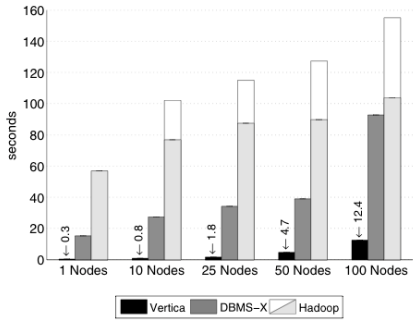


Figure 6: Selection Task Results

- Hadoop's start-up cost
- DBMS uses index
- Vertica's reliable message layer becomes bottleneck

Aggregation Task

- Calculate the total adRevenue generated for each sourceIP in the UserVisits table (20GB/node), grouped by the sourceIP column.
  - Nodes must exchange info for computing groupby
  - Generate 53 MB data regardless of number of nodes
- SQL:

```
SELECT sourceIP, SUM(adRevenue)
FROM UserVisits GROUP BY sourceIP;
```

- MR:
  - Map: outputs (sourceIP, adRevenue)
  - Reduce: compute sum per sourceIP
  - "Combine" is used

Aggregation Task

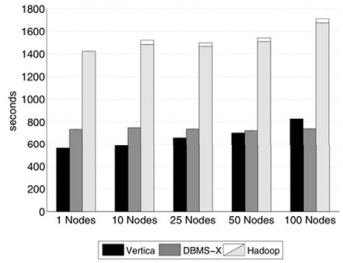


Figure 7: Aggregation Task Results (2.5 million Groups)

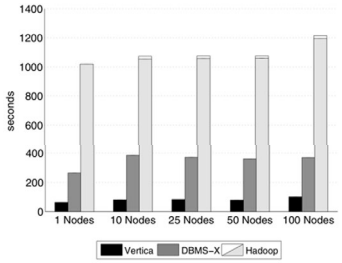


Figure 8: Aggregation Task Results (2,000 Groups)

- DBMS: Local group-by, then the coordinator performs the global group-by; performance dominated by data transfer.

Join Task

- Find the sourceIP that generated the most revenue within Jan 15-22, 2000, then calculate the average pageRank of all the pages visited by the sourceIP during this interval
- SQL:

```
SELECT INTO Temp sourceIP,
                 AVG(pageRank) as avgPageRank,
                 SUM(adRevenue) as totalRevenue
FROM Rankings AS R, UserVisits AS UV
WHERE R.pageURL = UV.destURL
              AND UV.visitDate BETWEEN Date('2000-01-15')
              AND Date('2000-01-22')
GROUP BY UV.sourceIP;

SELECT sourceIP, totalRevenue, avgPageRank
FROM Temp
ORDER BY totalRevenue DESC LIMIT 1;
```


Map Reduce

- Phase 1: filter UserVisits that are outside the desired date range, joins the qualifying records with records from the Ranking file
- Phase 2: compute total adRevenue and average pageRank per sourceIP
- Phase 3: produce the largest record

Join Task

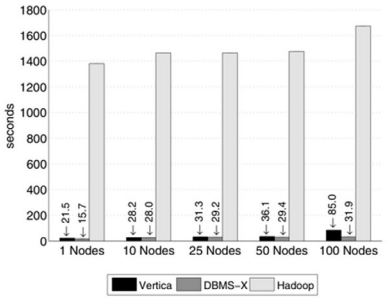


Figure 9: Join Task Results

- DBMS can use index, both relations are partitioned on the join key
- MR has to read all data
- MR phase 1 takes an average 1434.7 seconds
  - 600 seconds of raw I/O to read the table; 300 seconds to split, parse, deserialize; Thus CPU overhead is the limiting factor

UDF Aggregation Task

- Compute inlink count per document
- SQL:

```
SELECT INTO Temp F(contents) FROM Documents;
SELECT url, SUM(value) FROM Temp GROUP BY url;
```

Need a user-defined-function to parse HTML docs (C pgm using POSIX regex lib)

Both DBMS's do not support UDF very well, requiring separate program using local disk and bulk loading of the DBMS – *why was MR always forced to use Reduce to combine results?*

- MR:
  - A standard MR program

UDF Aggregation

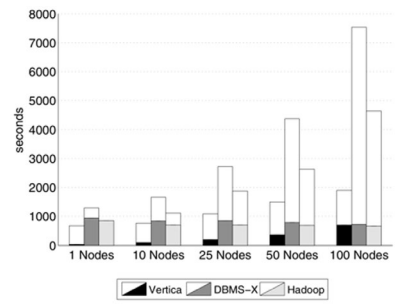


Figure 10: UDF Aggregation Task Results

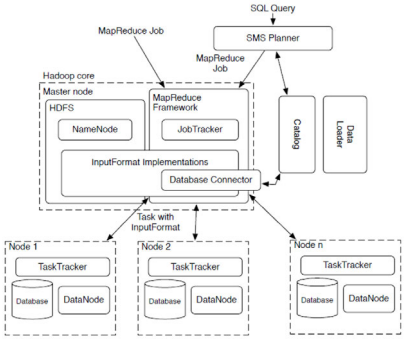
- DBMS: lower – UDF time; upper – other query time
- Hadoop: lower – query time; upper: combine all results into one

Discussion

- Throughput experiments?
- Parallel DBMSs are much more challenging than Hadoop to install and configure properly – DBMSs require professional DBAs to configure/tune
- Alternatives: Shark (Hive on Spark)
  - Eliminates Hadoop task start-up cost and answers queries with sub-second latencies
    - » 100 node system: 10 second till the first task starts, 25 seconds till all nodes run tasks
  - Columnar memory store (multiple orders of magnitude faster than disk
- Compression: does not help in Hadoop?
  - An artifact of Hadoop's Java-based implementation?
- Execution strategy (DBMS), failure model (Hadoop), ease of use (H/D)
- Other alternatives? Apache Hive, Impala (Cloudera) , HadoopDB (Hadapt), …

Alternative: HadoopDB?

- The Basic Idea (An Architectural Hybrid of MR & DBMS)
  - To use MR as the communication layer above multiple nodes running single-node DBMS instances
- Queries expressed in SQL, translated into MR by extending existing tools
  - As much work as possible is pushed into the higher performing single node databases|
- How many of complaints from Comparison paper still apply here?
- Hadapt startup commercializing



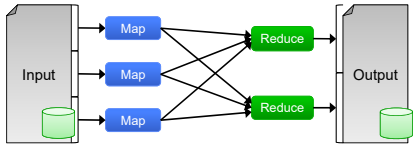
Is this a good paper?

- 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

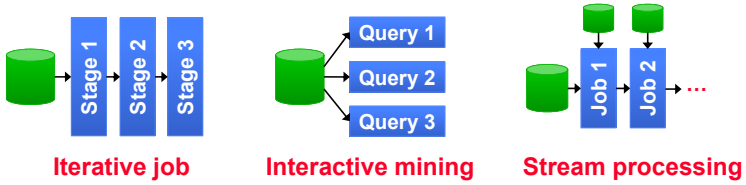
Issues with MapReduce

- 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

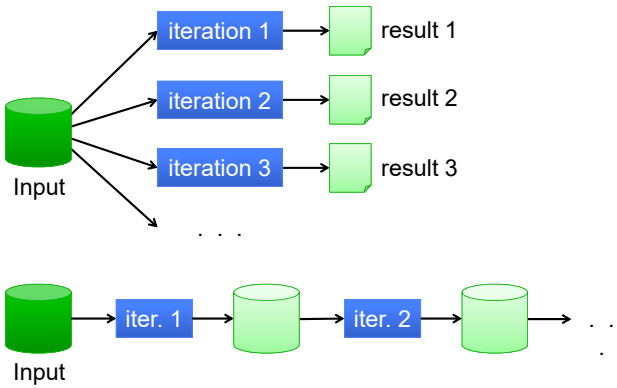


Issues with MapReduce

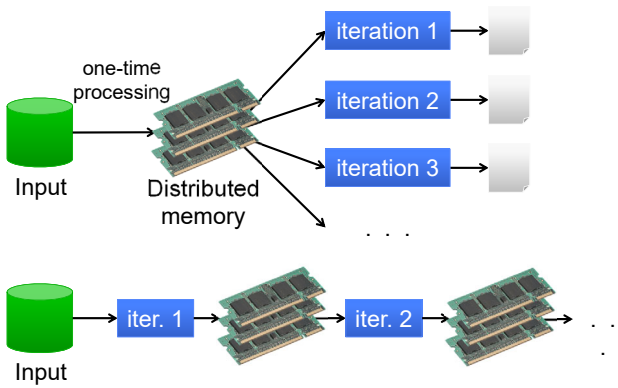
- *Acyclic data flow* from stable storage to stable storage ➔ poor support for applications that need to *reuse* pieces of data (I/O bottleneck and compute overhead)
  - Iterative algorithms (e.g. machine learning, graphs)
  - Interactive data mining (e.g. Matlab, Python, SQL)
  - Stream processing (e.g.,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



Example: Iterative Apps



Goal: Keep Working Set in RAM



Spark Goals

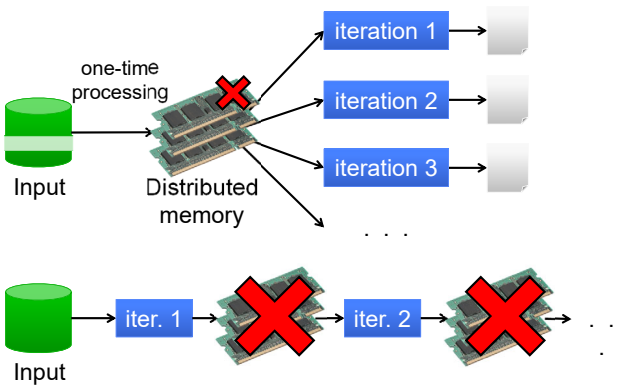
- Support iterative and stream jobs (apps with data reuse) efficiently:
  - Let them keep data in memory
- Experiment with programmability
  - Leverage Scala to integrate cleanly into programs
  - Support interactive use from Scala interpreter
- Retain MapReduce's fine-grained fault-tolerance and automatic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

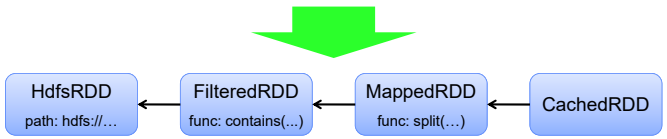
Shared Variables

- 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()
```



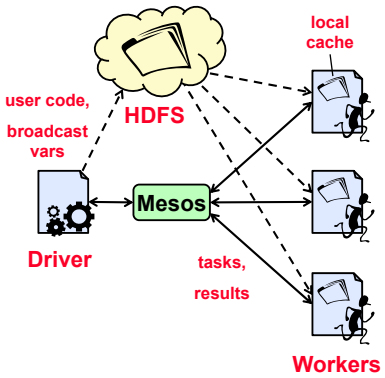
Architecture

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



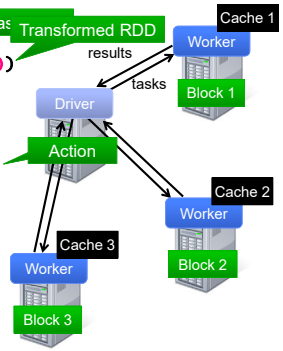
Spark Version of Word Count

```
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

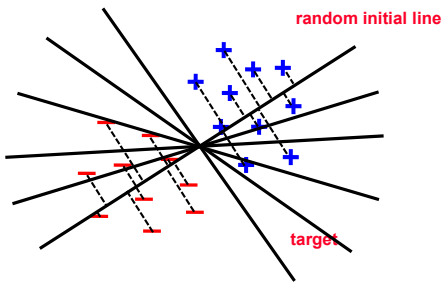
```
lines = spark.textFile("hdfs://...")
errors = lines.filter(_.startsWith("ERROR"))
messages = errors.map(_.split('\t')(2))
cachedMsgs = messages.cache()

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

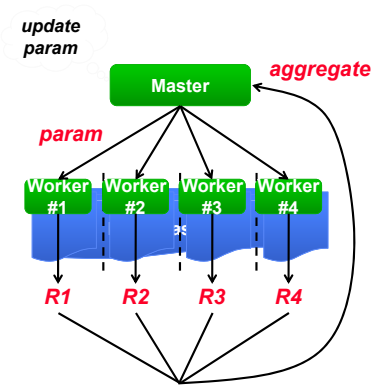
Goal: find best line separating two sets of points



Example: Logistic Regression

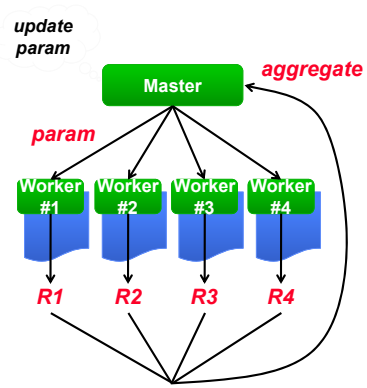
```
val data = spark.textFile(...).map(readPoint).cache()

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

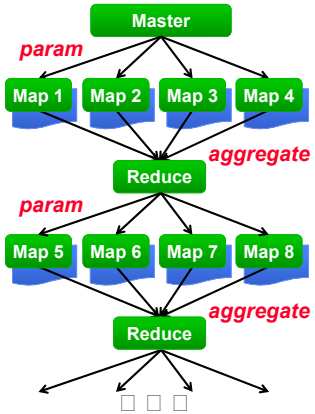


Spark

Job Execution

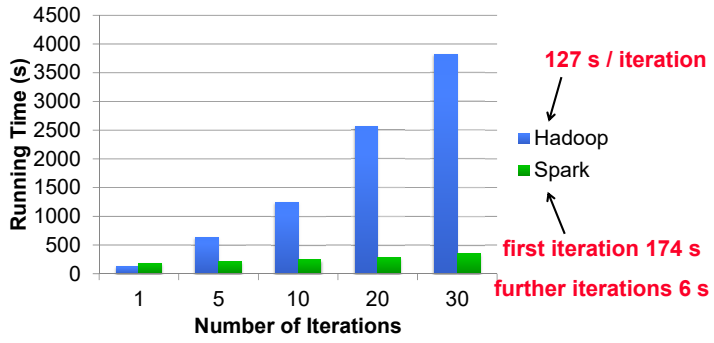


Spark



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

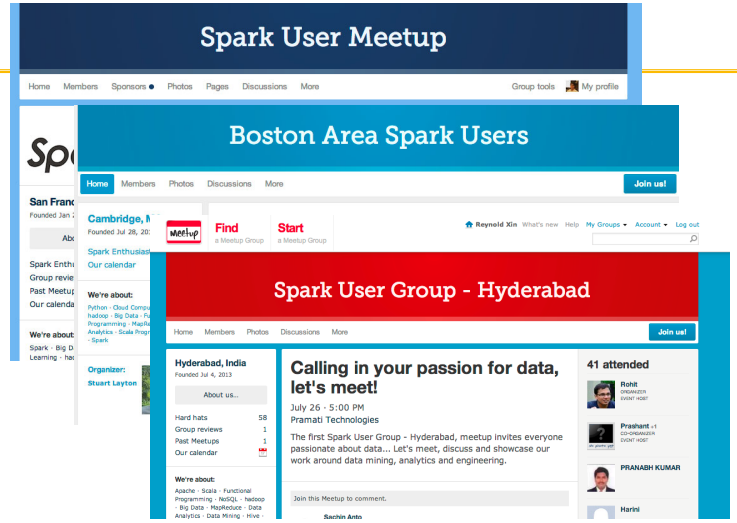
- RDDs work best when an application applies the same operation to many data records
  - Our approach is to just log the operation, not the data
- Will not work well for apps where processes asynchronously update shared state
  - Storage system for a web application
  - Parallel web crawler
  - Incremental 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

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?