MapReduce

Kate Donahue

[Some slides taken from Yiqing Hua and Mengqi Xia's presentation]

Overview

- MapReduce
  - Timeline
  - Core idea
  - Examples
  - Other design choices
  - Demonstrated Results
- Comparison
  - RDD paper
  - Friends or Foes? paper

Timeline

- 1998: Google founded
- 2004: Google IPO
- 2004: MapReduce paper
- 2006: Hadoop released
- 2010: “MapReduce and Parallel DBMSs: Friends or Foes?” paper
- 2012: “Resilient Distributed Datasets” paper

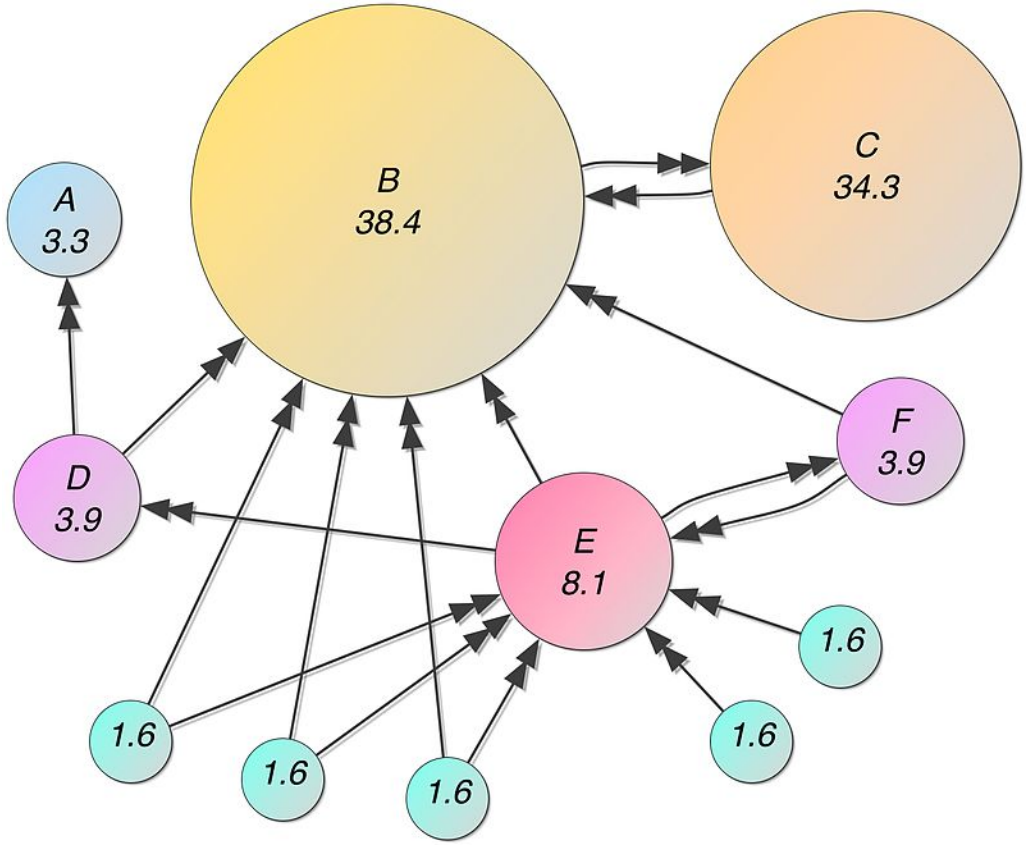
Authors

- Jeff Dean
  - Now head of Google AI
- Sanjay Ghemawat
  - Now senior fellow in Google Systems group
  - Went to Cornell but doesn’t donate enough

- Both joined Google early and were responsible for many core contributions, even by the time this paper was written.

Engineering need

- Google's core business: search
- Core search tool: PageRank
- PageRank calculates importance of webpages based off of links to other pages.
- "Join": matrix with non-zero entries if there is a link from one webpage to another



Research needs (Discussion questions)

- Engineering need:
  - Key question: *How do we compute PageRank on the entire web downloaded onto Google machines?*
- Developer need:
  - Parallelization: thinking about it is tricky
  - Key question: *How can we make it very easy for engineers to use many worker machines to solve core Google problems?*

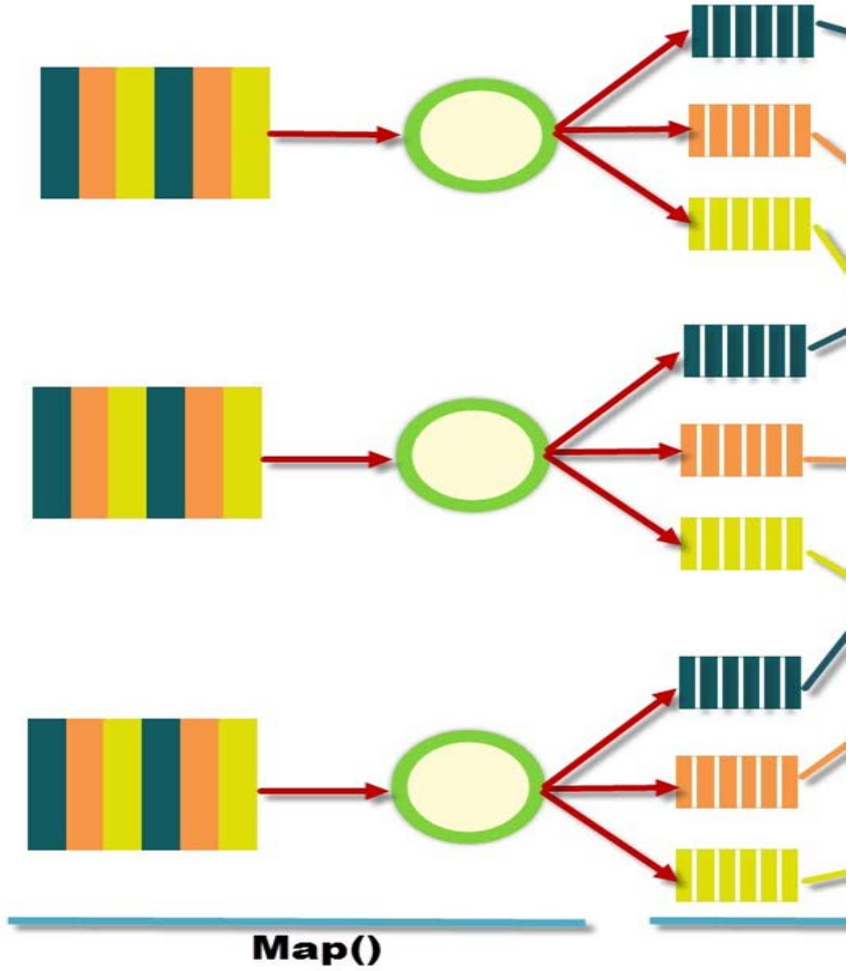
MapReduce

- A very simple framework with multiple implementations
- Map
  - Simple function taking in instances, calculating output associated with key
  - Write intermediate data
- (Shuffle)
  - Optimal step: rewrite instances so identical keys are located closer together
- Reduce
  - Combine results associated with same key
- Example: Word counts across documents

Step 1: define the “mapper”

```
map(String key, String value):
   // key: document name
   // value: document contents
   for each word w in document:
   EmitIntermediate (w, “1”);
```

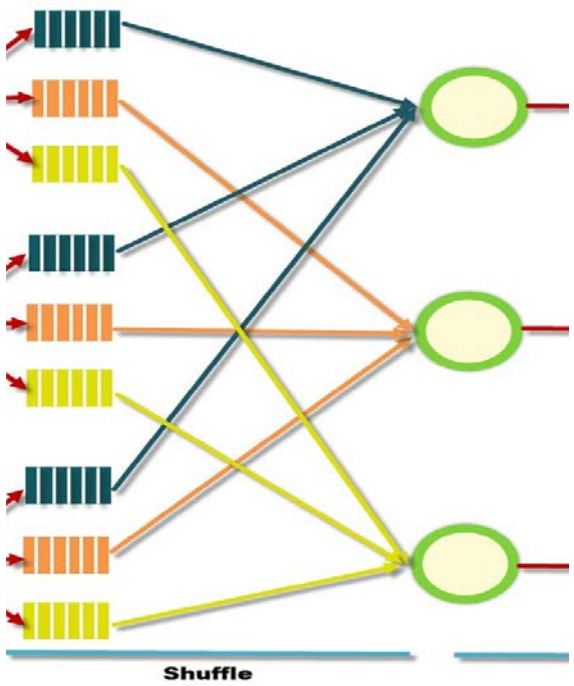
```
map(“Hamlet”, “Tis now strook twelve…”)
     {“tis”: “1”}
     {“now”: “1”}
     {“strook”: “1”}
     …
```



Step 2: Shuffling

The shuffling step aggregates all results with the same key **together** into a single list. (Provided by the framework)

```
{“tis”: “1”}
{“now”: “1”}
{“strook”: “1”}
{“the”: “1”}
{“twelve”: “1”}
{“romeo”: “1”}
{“the”: “1”}
…
```



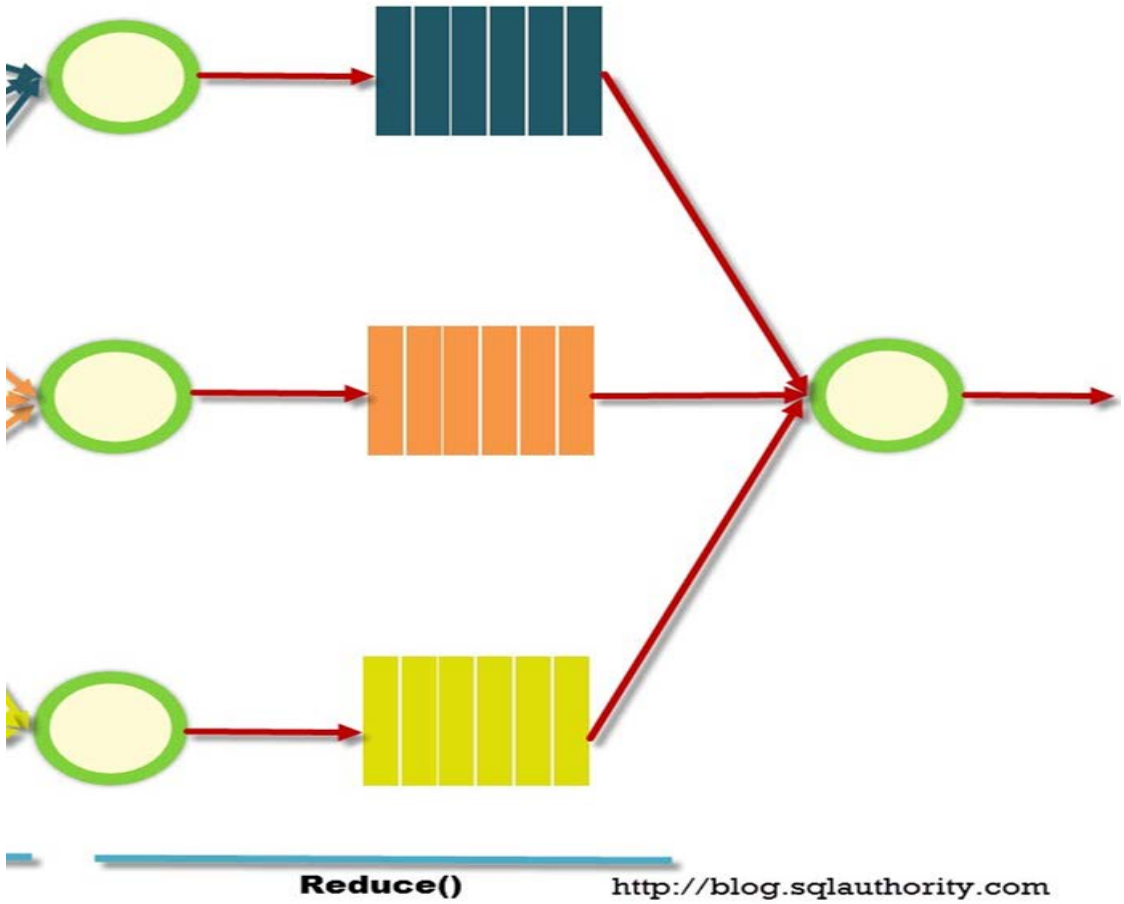

```
{“tis”:     [“1”,“1”,“1”...]}
{“now”:     [“1”,“1”,“1”]}
{“strook”:  [“1”,“1”]}
{“the”:     [“1”,“1”,“1”...]}
{“twelve”:  [“1”,“1”]}
{“romeo”:   [“1”,“1”,“1”...]}
{“juliet”:  [“1”,“1”,“1”...]}
…
```

Step 3: Define the Reducer

Aggregates all the results together.

```
reduce(String key, Iterator values):
    // key: a word
    // values: a list of counts
    sum = 0
    for each v in values:
      result += ParseInt(v)
      Emit (AsString(result))
```

```
reduce("tis", ["1","1","1","1","1"])
    {"tis": "5"}
reduce("the", ["1","1","1","1","1","1","1"...])
    {"the": "23590"}
reduce("strook", ["1","1"])
    {"strook": "2"}
    ...
```



MapReduce

- A very simple framework with multiple implementations
- Map
  - Simple function taking in instances, calculating output associated with key
  - Write intermediate data
- (Shuffle)
  - Optimal step: rewrite instances so identical keys are located closer together
- Reduce
  - Combine results associated with same key
- Example: Word counts across documents

Other examples

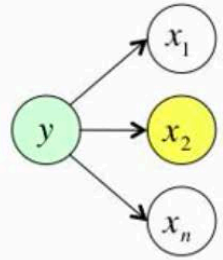
- Reverse Web-link graph: <target, list of sources>
  - Map: Ingests a source and produces <target, source> pairs for each target
  - Shuffle: Sort by targets
  - Reduce: Concatenate to produce <target, list(source)> output.

Other examples

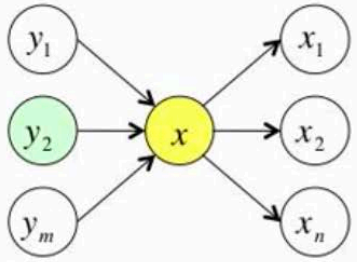
- Calculate PageRank algorithm:
  - Iterative process – repeated.
  - Map: Ingests a source and produce <target, calculated PR> for each target.
  - Shuffle: sort by targets
  - Reduce: Combine PR from all sources for a given target

PageRank using MapReduce

- **Mapper**: $\langle y, \{x_1 \cdots x_n\} \rangle$ node + out-links
  - for j=1…n: emit $\left\langle x_j, \frac{PR(y)}{out(y)} \right\rangle$



- **Reducer**: $\left\langle x, \left\{ \frac{PR(y_1)}{out(y_1)}, \cdots, \frac{PR(y_m)}{out(y_m)} \right\} \right\rangle$ node + ΔPR from in-links
  - compute: $PR(x) = \frac{1-\lambda}{N} + \lambda \sum_{y \to x} \frac{PR(y)}{out(y)}$
  - for j=1…n: emit $\left\langle x_j, \frac{PR(x)}{out(x)} \right\rangle$



Other examples

- Distributed sort:
  - Map: Ingests record, produces <key, record> pair.
  - Shuffle: Sort by key.
  - Reduce: Identity function.
- Calculate mean by key:
  - Map: Ingests <key, value> pair and produces same pair: is identity map.
  - Shuffle: Sort by key.
  - Reduce: Calculate mean for each key.

Implementation Environment

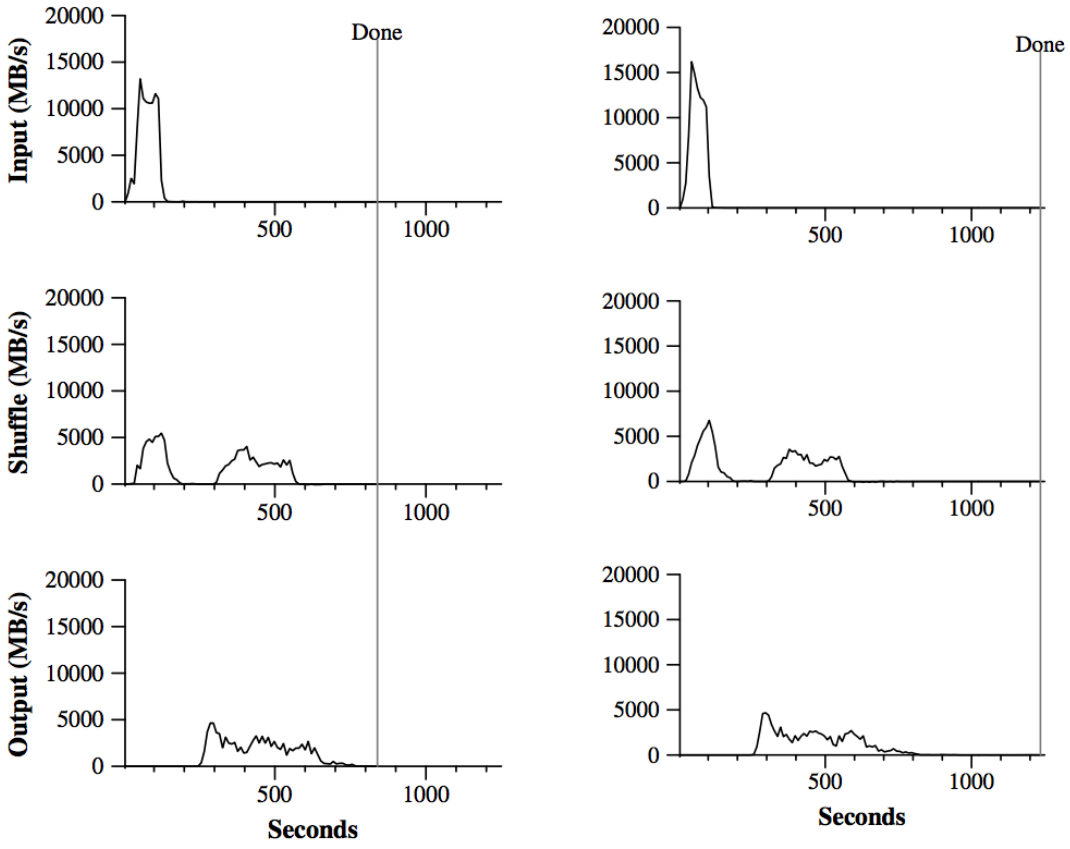
- Machines: dual-processor running Linux, 2-4 GB memory
- Commodity Networking Hardware: 100 MB/s or 1 GB/s, averaging less
- Cluster: hundreds or thousands of machines → Common Machine Failure
- Storage: disks attached to machines
- File System: GFS
- Users submit jobs (consists of tasks) to scheduler, scheduler schedules to machines within a cluster.

Design choices in paper implementation

- M: number of map tasks (should be much larger than the total number of machines – heterogenous machines and tasks)
- R: number of reduce tasks (should be a small multiple of number of machines – GFS restrictions)
- “Combiner” function does local reduction for commutative reduction tasks (like addition).
- If a worker fails to respond, re-assign its task to another worker.

Stragglers experiment

- If a worker fails to respond, re-assign its task to another worker.
- Sort example:
- Two humps for shuffle around the mapping and reduction steps.
- Without backup steps, takes 44% longer to run.

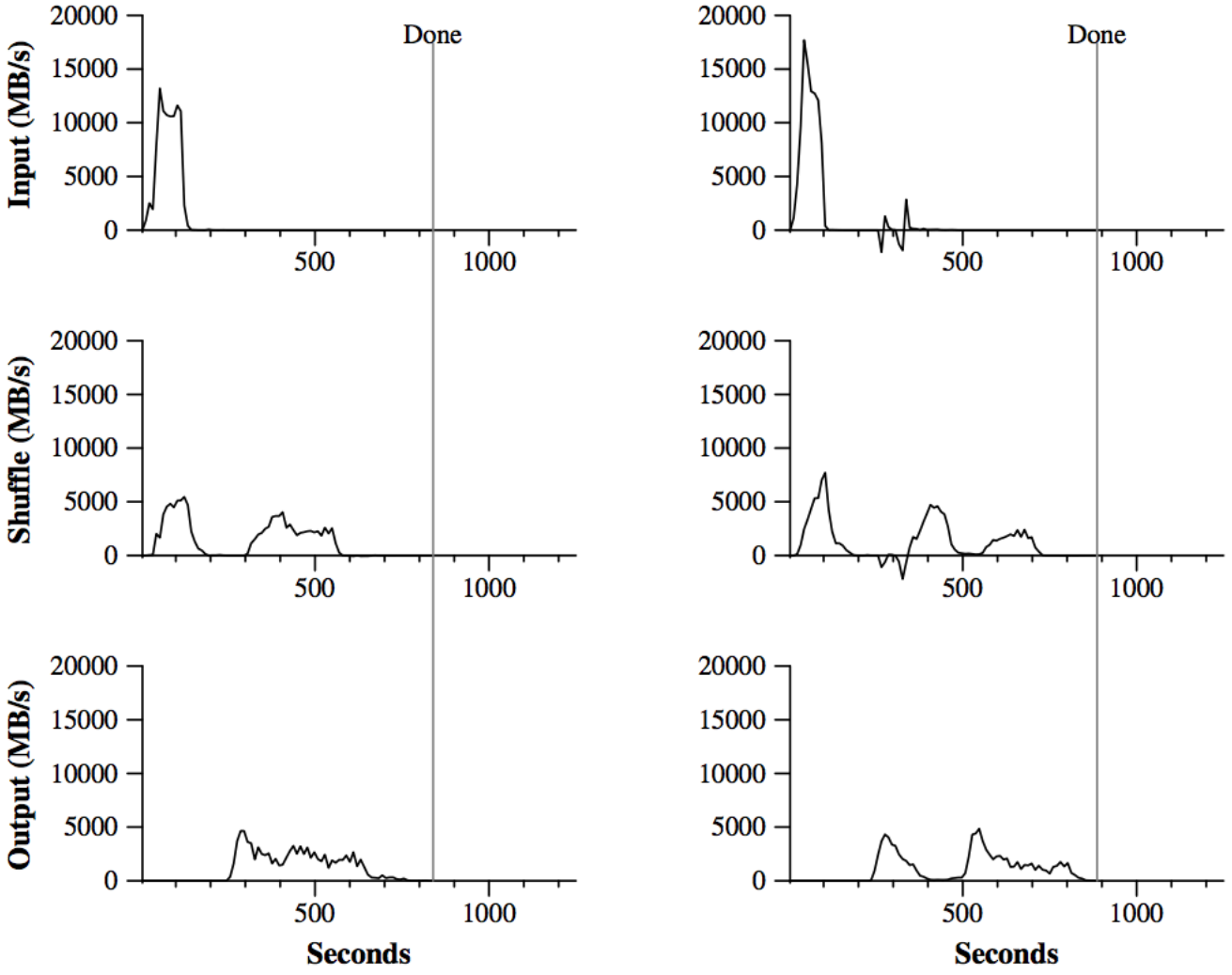


(a) Normal execution (b) No backup tasks

Figure 3: Data transfer rates over time for different executions

Killing workers experiment

- Kill 200 workers while process has begun.
- Tasks re-assigned, only 5% longer to complete.



(a) Normal execution

(c) 200 tasks killed

Usage at Google

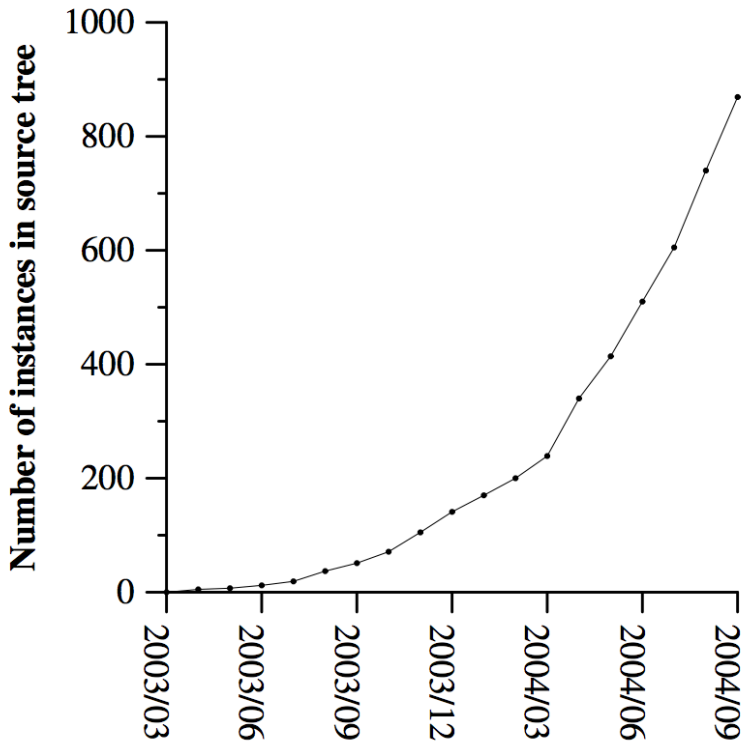


Figure 4: MapReduce instances over time

- Increasing usage at Google up to publication of this paper
- Use cases:
  - Machine learning
  - Clustering for Google News
  - Graph computations
  - Extracting properties from web pages
- Ease of use cited as helping widespread utilization

MapReduce Falling Behind User Desires

MapReduce greatly simplified “big data” analysis on large, unreliable clusters
But as soon as it got popular, users wanted more:

1. More complex, multi-stage applications (for example, iterative machine learning)
2. More interactive ad-hoc queries

Iterative algorithms and interactive data queries both require one thing that MapReduce lacks:
Efficient **data sharing** primitives

Limitations

MapReduce shares data across jobs by writing to stable storage.
This is **SLOW** because of replication and disk I/O, but necessary for fault tolerance in MapReduce's framework
However, this isn't necessary for fault tolerance in all frameworks – foreshadowing to Spark later!

Research question

- Why did Google invent MapReduce rather than just using databases and database processing algorithms?

“MapReduce and Parallel DBMSs: Friends or Foes?”

“It was not until we received expert support from one of the vendors that we were able to get one particular DBMS to run queries that completed in minutes, rather than hours or days.”

- MapReduce (Hadoop implementation)
  - Extract-Transform-Load functionality
  - Easier to get started with, free.
  - Allows unstructured data, more flexible code structure than SQL.
  - Potentially better for “quick and dirty” one-off runs of data.
- Parallel database management systems
  - Database systems
  - Much trickier to get up and running.
  - Structured data and SQL queries.
  - Better when repeated queries are likely or results need to be stored.

MapReduce vs. Parallel DBMS

	Hadoop	DBMS-X
Grep	284s	194s
Web Log	1,146s	740s
Join	1,158s	32s

- Replicated tasks from original MapReduce paper, specifically chosen so indexing or other database techniques wouldn't be helpful
- Despite this, DBMS had much higher performance!

Potential Explanations

- Mainly architectural decisions, not inherent limitations of MapReduce.
- Repetitive record parsing: data stored in same form it was originally stored in (requires repeatedly converting from text).
- Compression appears to help DBMS much more than MR.
- Scheduling: DBMS has pre-build query plan, so easier to optimize.
- In DBMS, data is sent directly from one worker to another, rather than being written to disk.
- *“The two technologies are complementary, and we expect MR-style systems performing ETL to live directly upstream from DBMSs”*

Is there a better way to do MapReduce?

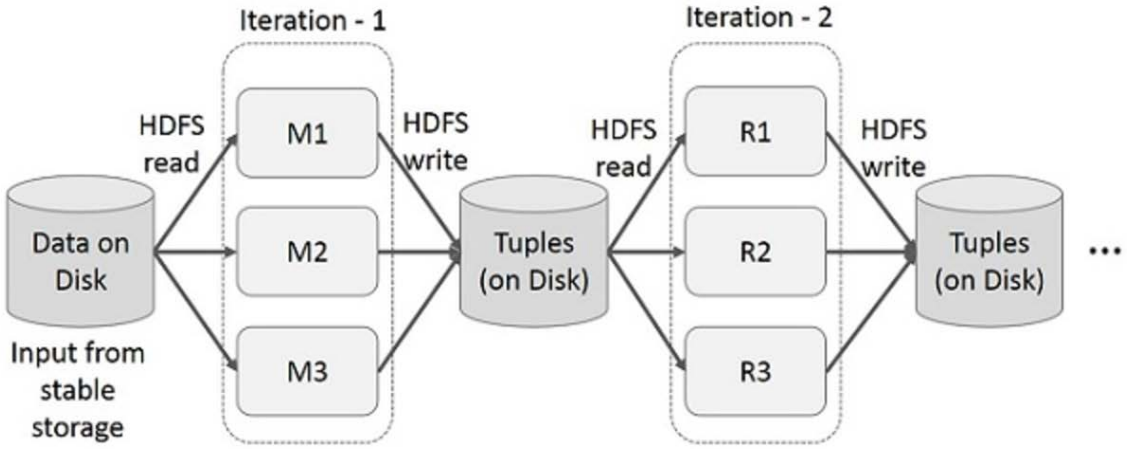
- Research goal: Can we get the advantages of MapReduce over databases without the slowdown?

Resilient Distributed Datasets (Spark)

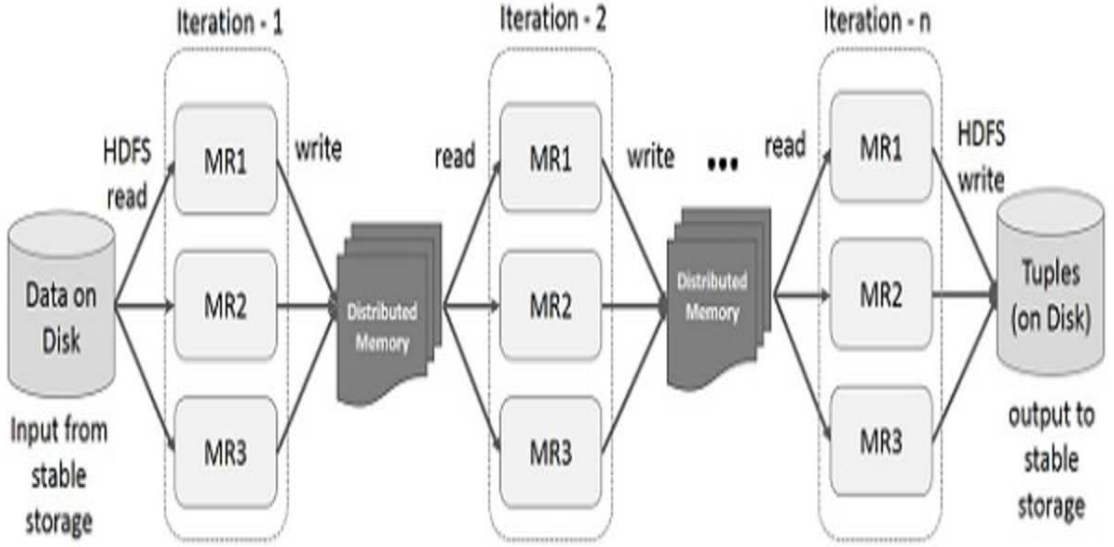
- Keep intermediate results in memory. For fault-tolerance, keep “lineage” of steps required to produce data. In case of a fault, it is easier to reproduce data from versions still in memory: look at the specific input lines needed to produce the output lines that were lost.
- Only coarse-grained operations (join, map, filter) rather than cell-level manipulation. Easier to maintain a log of transformations, but restricts actions you can take.
- No checkpointing (writing of intermediate steps) necessary.
- Users can ask for certain outputs to persist.

Iterative Operations

MapReduce

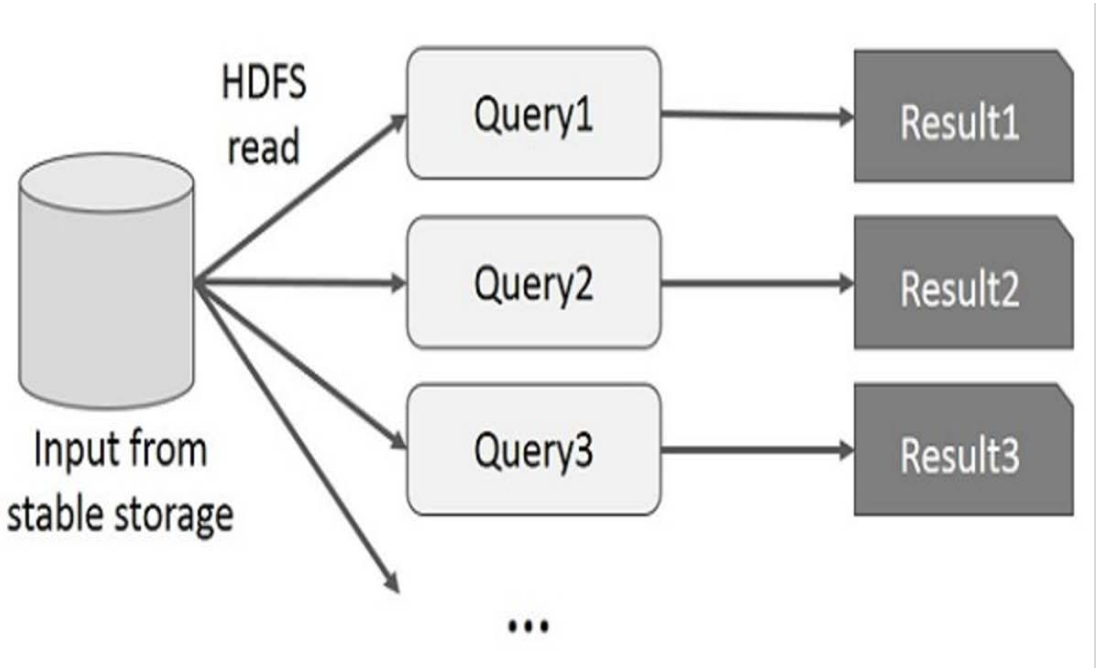


Spark RDD

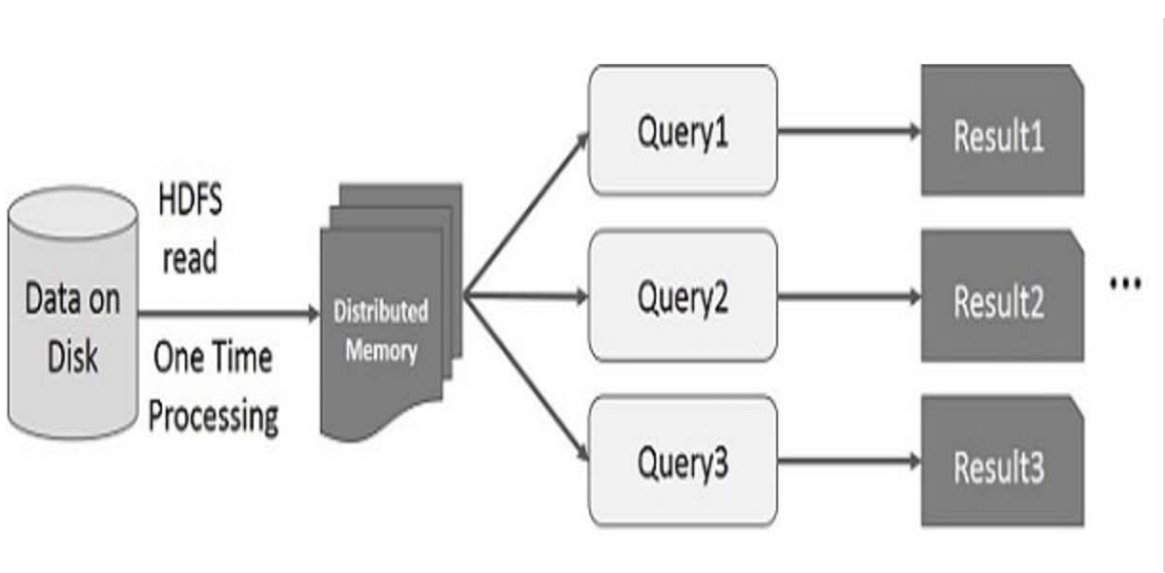


Interactive Operations

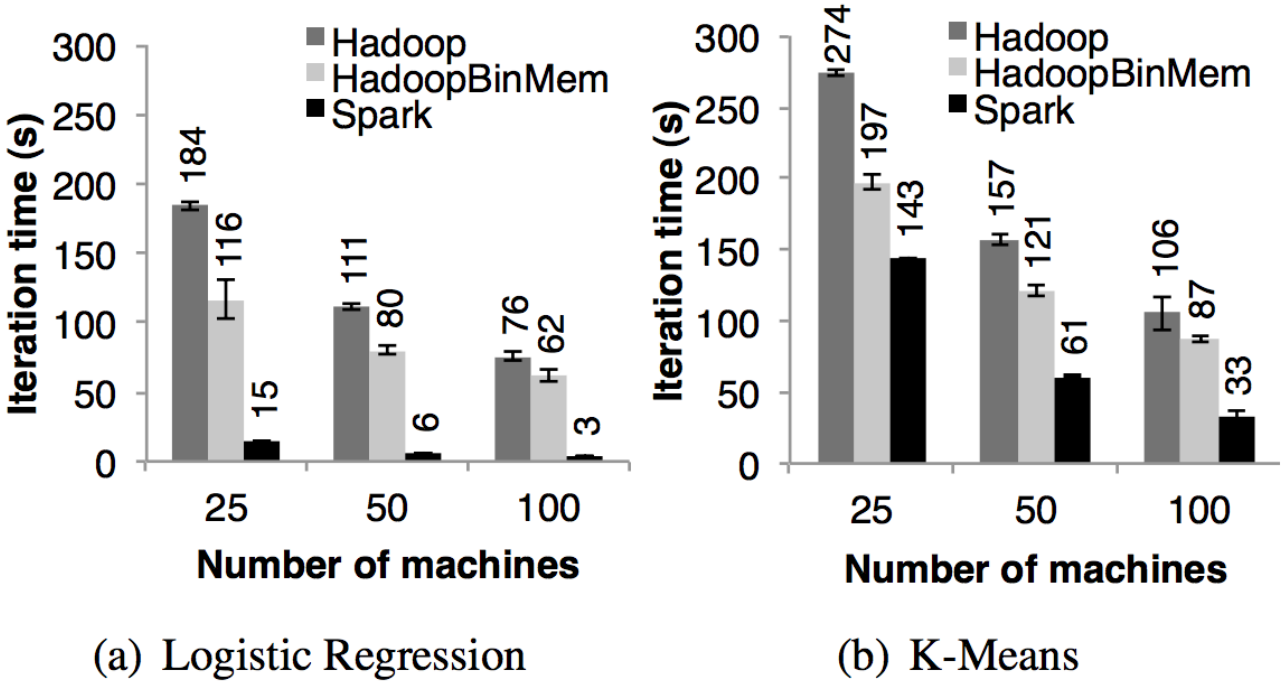
MapReduce



Spark RDD



Performance: Time



(a) Logistic Regression (b) K-Means

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

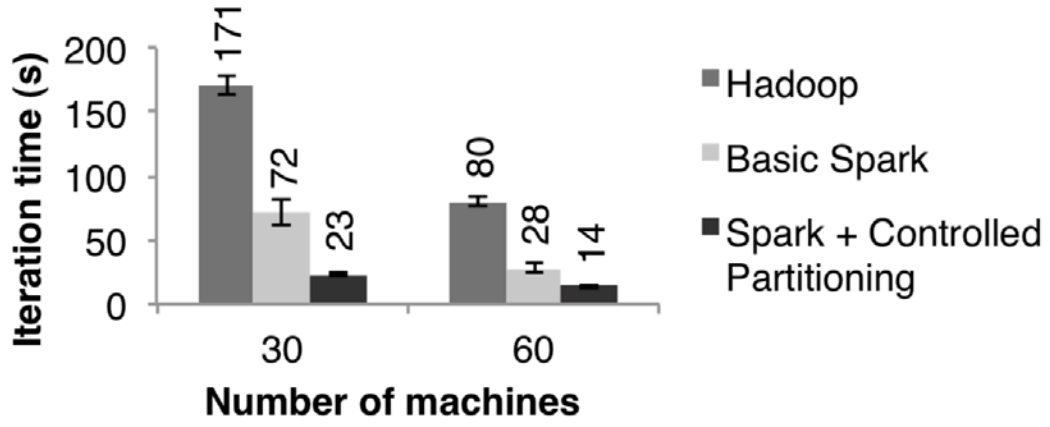
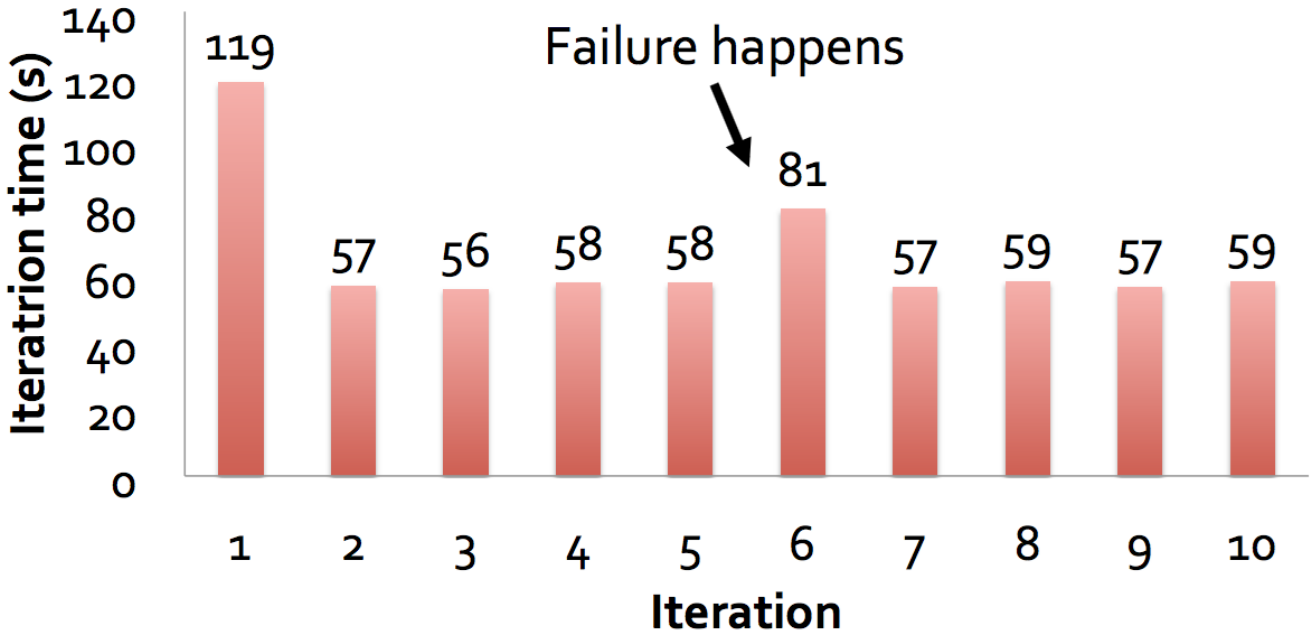


Figure 10: Performance of PageRank on Hadoop and Spark.

Performance: Fault-resilience

When nodes fail, Spark can recover quickly by rebuilding only the lost RDD partitions.



Limitations

1. RDDs are best suited for batch applications that apply the same operation to all elements of a dataset. RDDs are not suitable for applications that make asynchronous fine-grained updates to shared state.
2. Spark loads a process into memory and keeps it for the sake of caching. If the data is too big to fit entirely into the memory, then there could be major performance degradations.

MapReduce vs Spark

	MapReduce	Spark
Developed at	Google	UC Berkeley
Designed for	Batch processing	Real time processing that involves iterative/interactive operations
In-memory processing support	No	Yes
Intermediate results are stored in	Hard disk	Memory
Fault tolerance is ensured by	Data replication	Transformation log
Bottle neck	Frequent disk I/O	Large memory consumption

Perspectives

- MapReduce, databases, Spark, all differ:
  - In engineering needs for a particular application
  - In human needs in a particular situation
- Selecting between them is likely a process of balancing these objectives.