MapReduce & Resilient Distributed Datasets

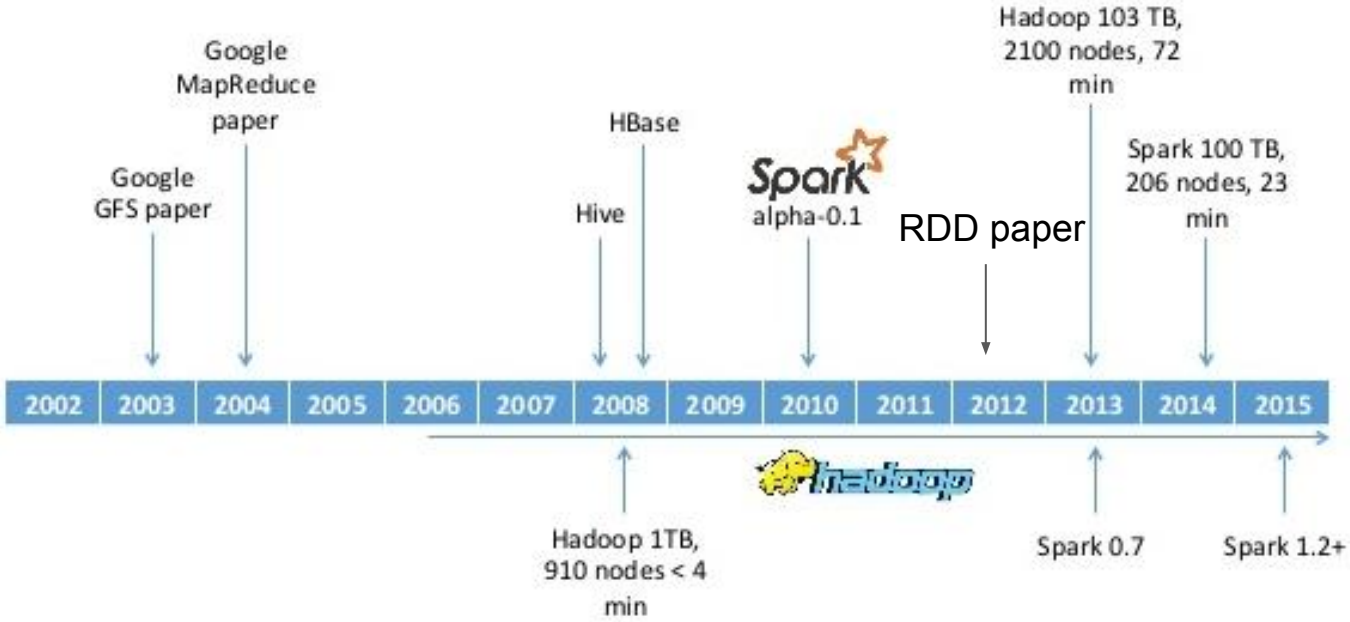
Yiqing Hua, Mengqi(Mandy) Xia

Outline

- MapReduce:
  - Motivation
  - Examples
  - The Design and How it Works
  - Performance
- Resilient Distributed Datasets (RDD)
  - Motivation
  - Design
  - Evaluation
- Comparison

MapReduce: Simplified Data Processing on Large Clusters

Timeline



https://image.slidesharecdn.com/adioshadoopholasparkt3chfestdhiguero-150213043715-conversion-gate01/95/adios-hadoop-hola-spark-t3chfest-2015-9-638.jpg?cb=1423802358

MapReduce: Simplified Data Processing on Large Clusters

OSDI 2004

22,495 citations

ACM Prize in Computing (2012)
2012 ACM-Infosys Foundation Award

- 

  Jeffrey Dean -- Google Senior Fellow in the Systems and Infrastructure Group

  "When Jeff has trouble sleeping, he Mapreduces sheep."

- 

  Sanjay Ghemawat -- Google Fellow in the Systems Infrastructure Group

  Cornell Alumni

Motivation

The need to process large data distributed across hundreds or thousands of machines in order to finish in a reasonable amount of time.

In 2003, Google published the Google File System Paper.

People want to take advantage of GFS and hide the issues of parallelization, fault-tolerance, data distribution and load balancing from the user.

What is MapReduce?

What is MapReduce?

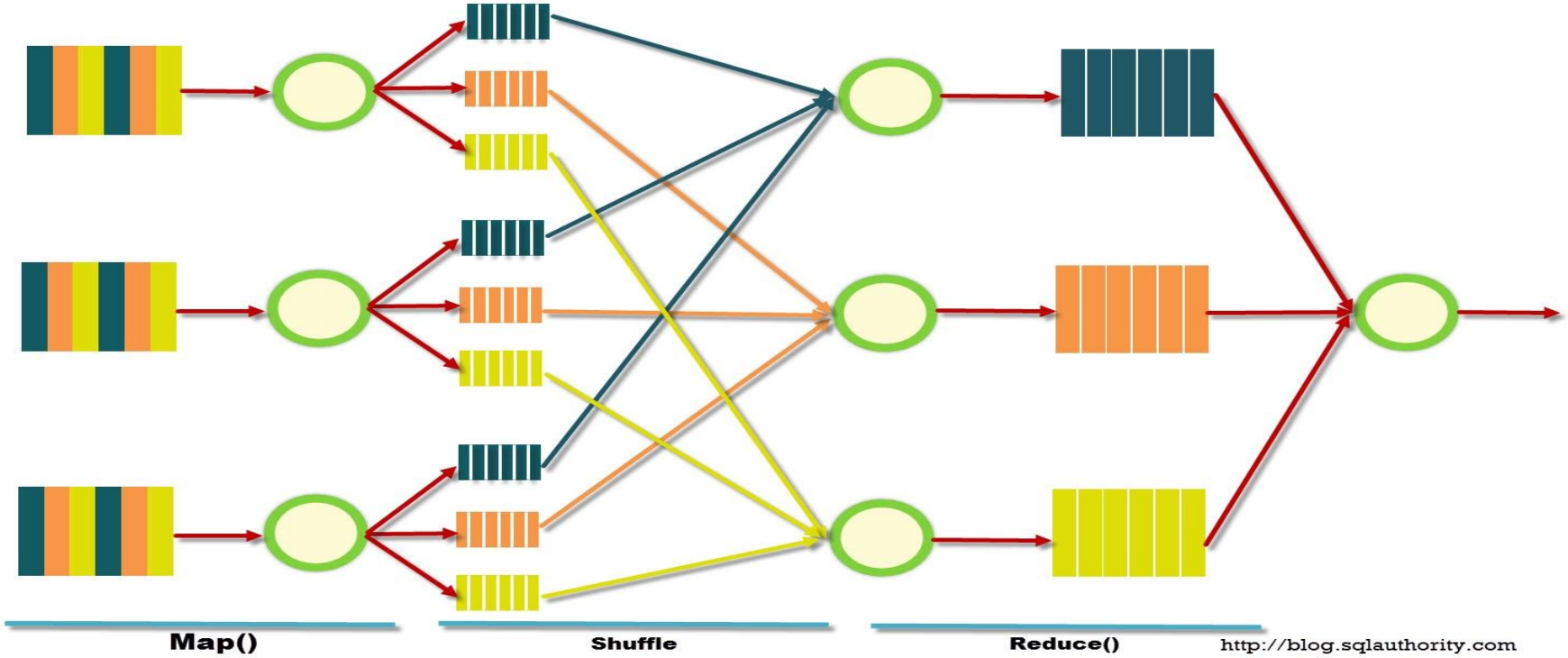
MapReduce is a software framework for easily writing applications which process vast amounts of data (multi-terabyte data-sets) in-parallel on large clusters (thousands of nodes) of commodity hardware in a reliable, fault-tolerant manner.

https://hadoop.apache.org

MR is more like an extract-transform-load (ETL) system than a DBMS, as it quickly loads and processes large amounts of data in an ad hoc manner. As such, it complements DBMS technology rather than competes with it.

MapReduce and Parallel DBMSs: Friends or Foes?
Michael Stonebraker et al.

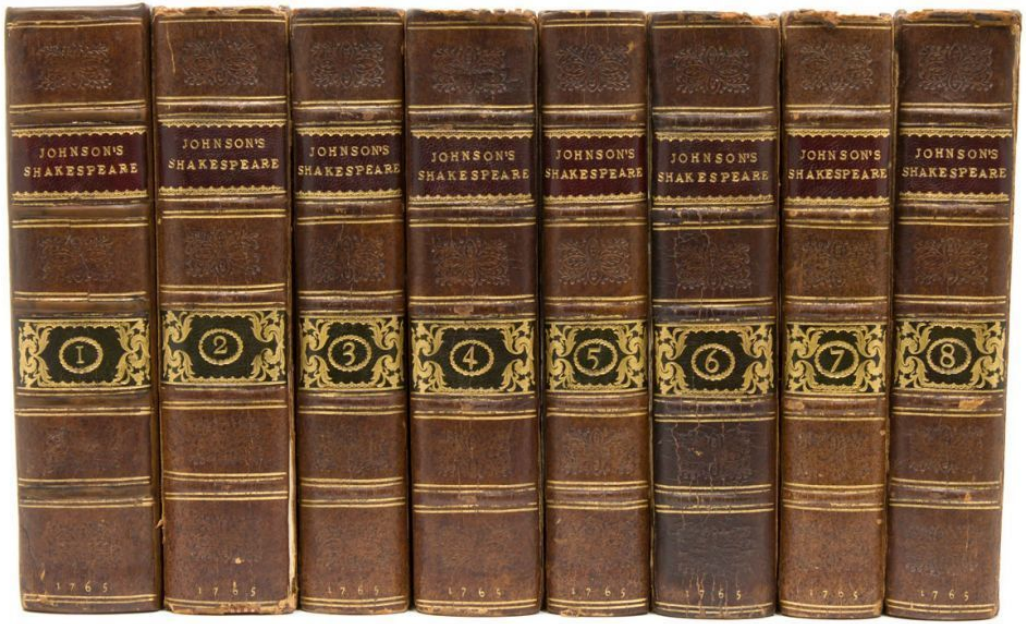
What is MapReduce?



Example: Word Count of the Complete Work of Shakespeare

BERNARDO Who's there?
FRANCISCO Nay, answer me: stand, and unfold yourself.
BERNARDO Long live the king!
FRANCISCO Bernardo?
BERNARDO He.
FRANCISCO You come most carefully upon your hour.
BERNARDO 'Tis now struck twelve; get thee to bed, Francisco.

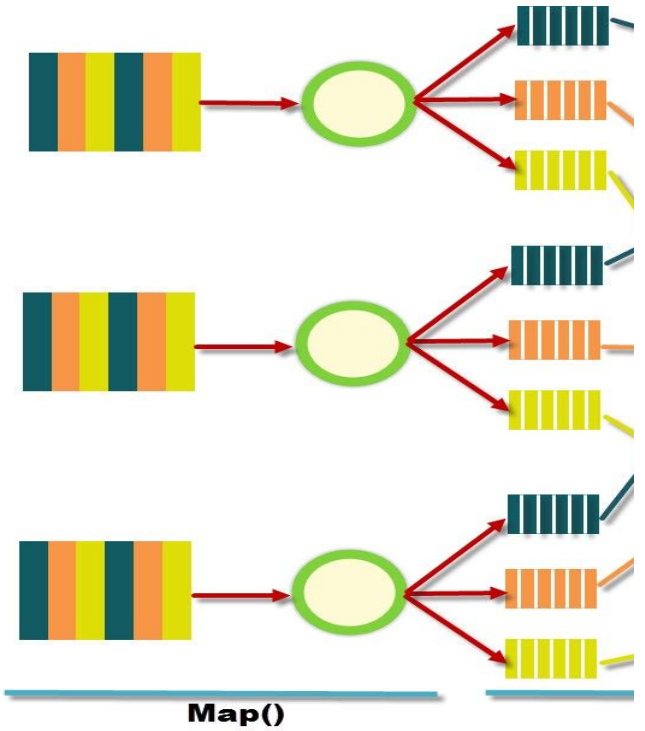
……



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”);
```

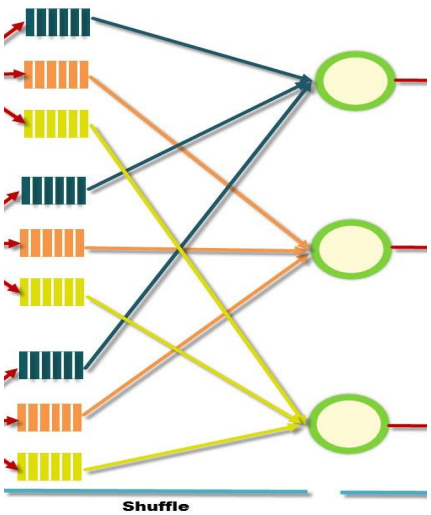
```
map(“Hamlet”, “Tis now strook twelve…”)
    {“tis”: “1”}
    {“now”: “1”}
    {“strook”: “1”}
    …
```



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"}
...
```



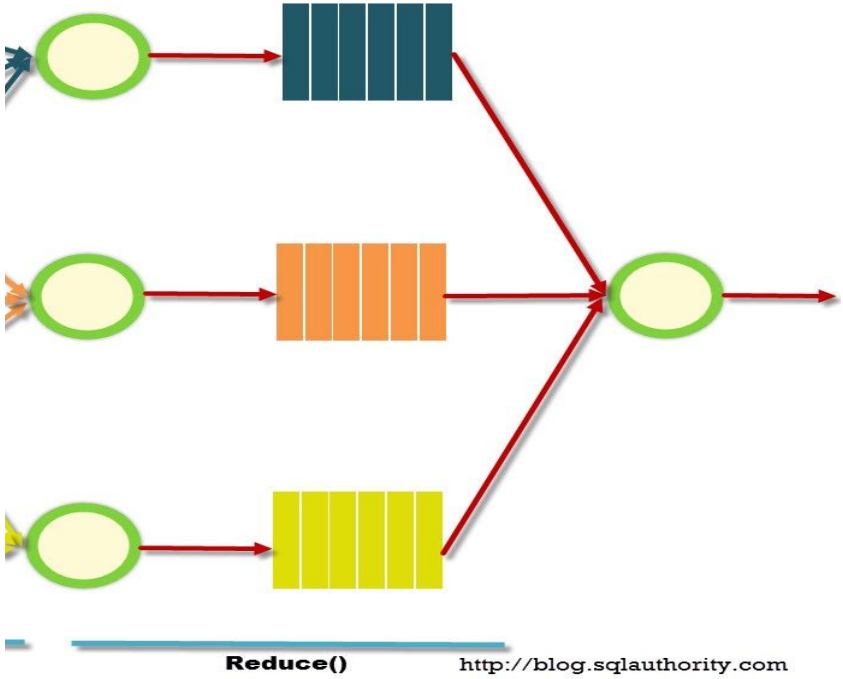

```
{"tis":     ["1","1","1"...]}
{"now":     ["1","1","1"]}
{"strook":  ["1","1"]}
{"the":     ["1","1","1"...]}
{"twelve":  ["1","1"]}
{"romeo":   ["1","1","1"...]}
{"juliet":  ["1","1","1"...]}
...
```

Step 3: Define the Reducer

Aggregates all the results together.

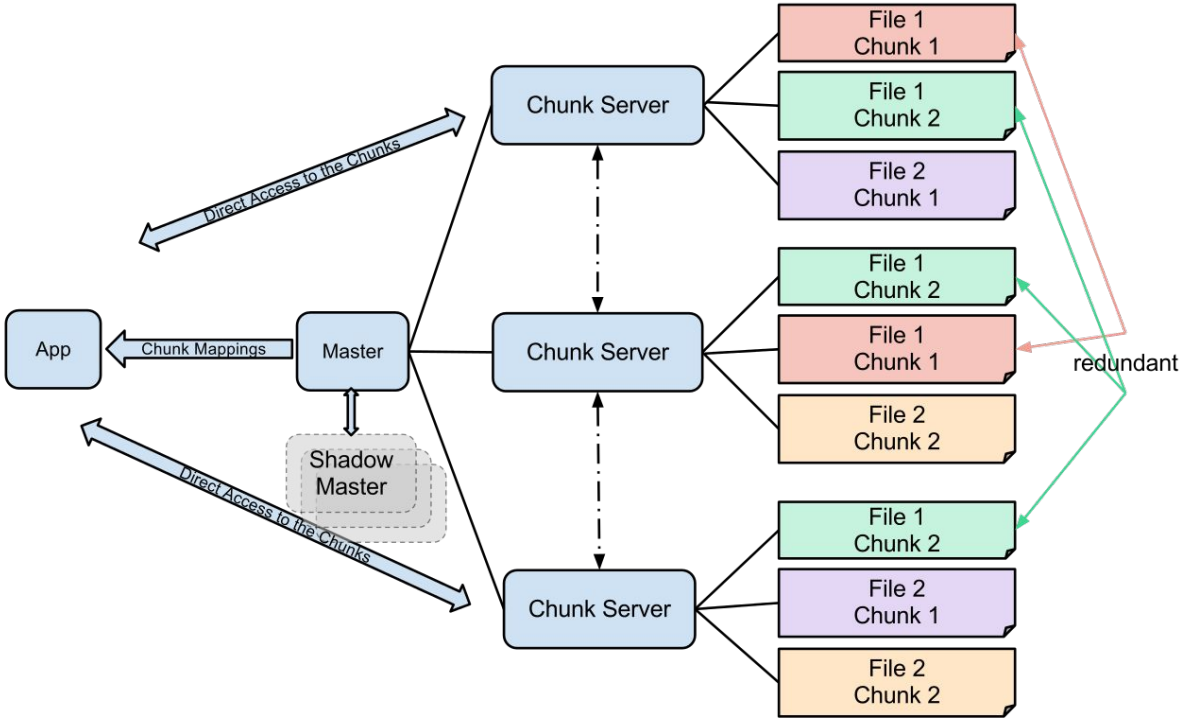
```
reduce(String key, Iterator values):
    // key: a word
    // values: a list of counts
    sum = 0
    for each v in values:
        result += ParseInt(v)
        Emit (AsString(result))
```

```
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"}
    ...
```



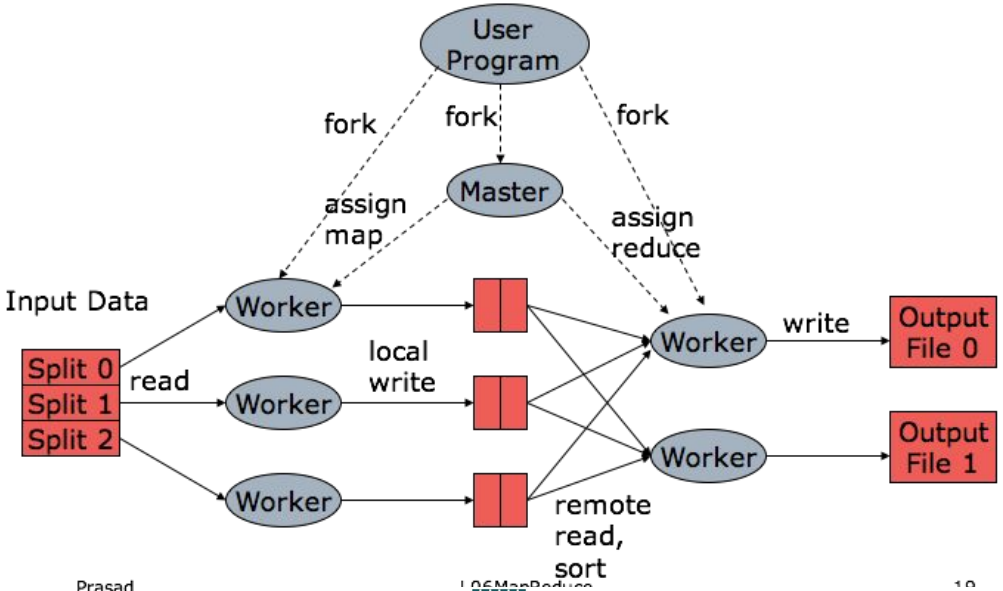
The Design and How it Works

Google File System

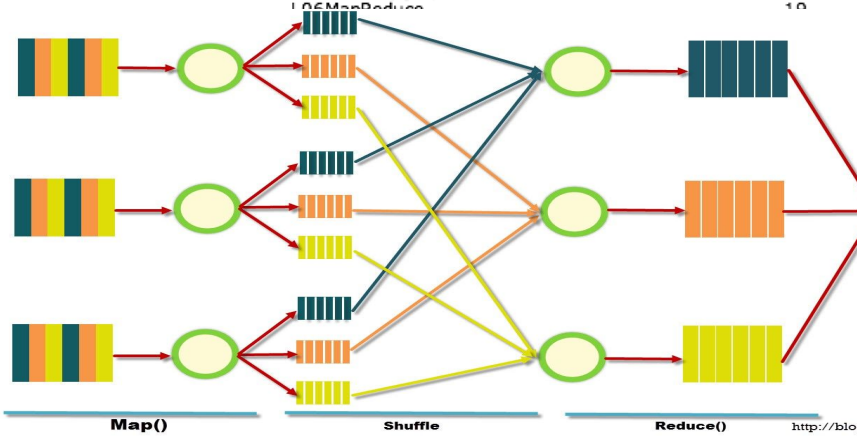


- User-level process running on Linux commodity machines
- Consist of Master Server and Chunk Server
- Files broken into chunks, 3x redundancy
- Data transfer between client and chunk server

Infrastructure



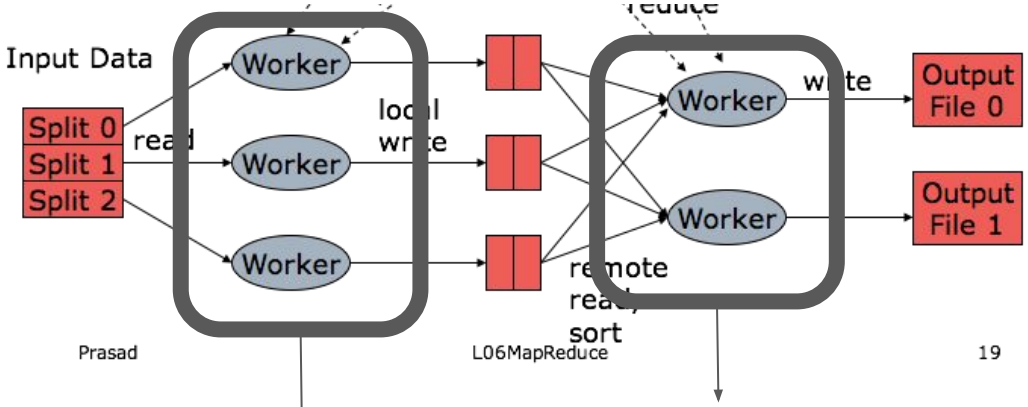
Prasad



Fault Tolerance -- Worker

Periodically Pinged by Master

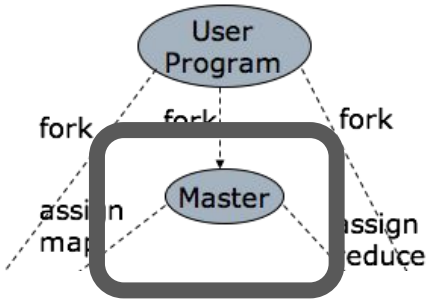
NO response = failed worker => task reassigned



Re-execute incomplete failed task

Re-execute failed task
Notify reducers working on this task

Fault Tolerance -- Master



Master writes periodic checkpoints
→ New master can start from it

Master failure doesn't occur often
→ Aborts the job and leave the choice to client

Fault Tolerance -- Semantics

Atomic Commits of Outputs Ensures

→ Same Result with Sequential Execution of Deterministic Programs

→ Any Reduce Task will have the Same Result with a non-Deterministic Program with Sequential Execution with a Certain Order (But not necessarily the same one for all the reduce tasks)

Locality

Implementation Environment:

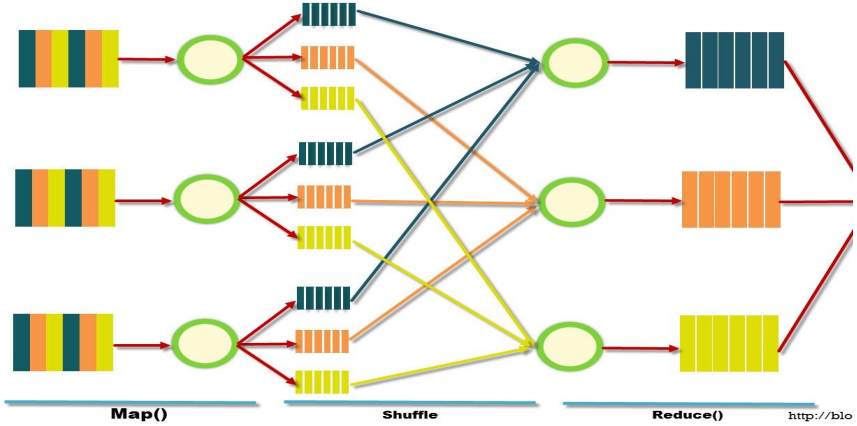
- Storage: disks attached to machines
- File System: GFS

Locality == efficiency

Master node can schedule jobs to machines that have the data

Or as close as possible to the data

Task Granularity



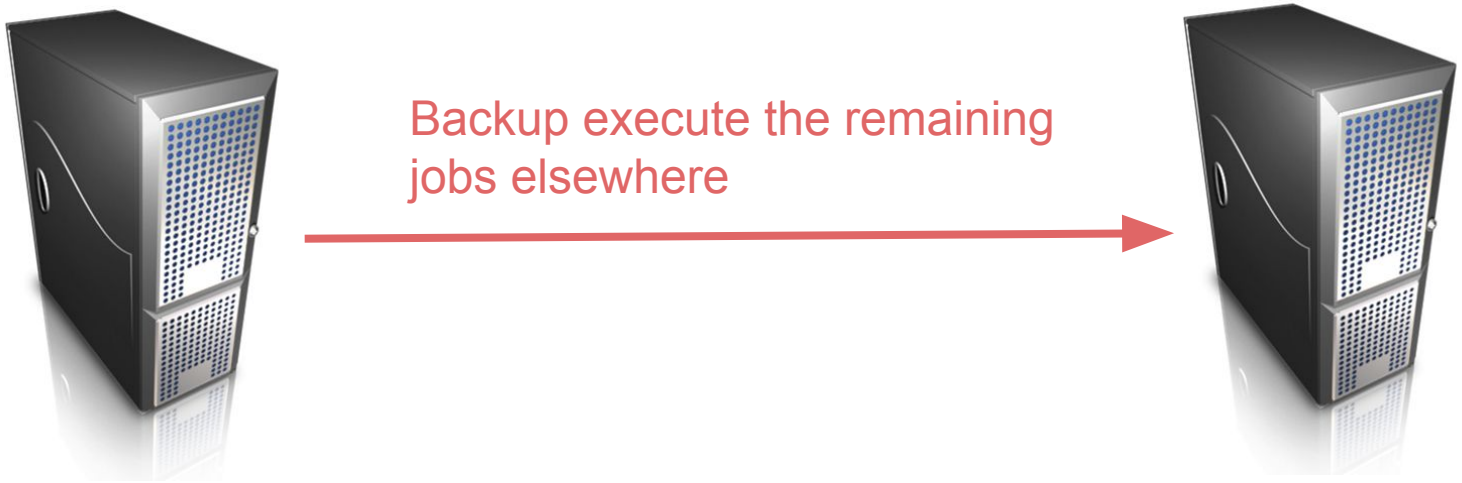
How many map tasks and
how many reduce tasks?

- The more the better → improves dynamic load balancing, speeds up recovery
- Master nodes has a memory limit to keep the states
- Also you probably don’t want tons of output files

Stragglers

The machine running the last few tasks that takes forever

Stragglers



The machine running the last few tasks that takes forever

Refinements

1. Partitioning Function
2. Ordering Guarantees
3. Combiner Function
4. Input and Output Types
5. Side-effects
6. Skipping Bad Records
7. Local Execution
8. Status Information
9. Counters

Refinements

1. **Partitioning Function**
2. Ordering Guarantees
3. Combiner Function
4. Input and Output Types
5. Side-effects
6. Skipping Bad Records
7. Local Execution
8. Status Information
9. Counters

Basically with this you can define your own fancier mapper

Like mapping hostname

Refinements

1. Partitioning Function
2. **Ordering Guarantees**
3. Combiner Function
4. Input and Output Types
5. Side-effects
6. Skipping Bad Records
7. Local Execution
8. Status Information
9. Counters

Intermediate results are sorted in key order:

- Efficient random lookup
- If you want it sorted

Refinements

1. Partitioning Function
2. Ordering Guarantees
3. **Combiner Function**
4. Input and Output Types
5. Side-effects
6. Skipping Bad Records
7. Local Execution
8. Status Information
9. Counters

Partial merge of the data before sending to the network:
In the case of word count, it can be more efficient

Refinements

1. Partitioning Function
2. Ordering Guarantees
3. Combiner Function
4. **Input and Output Types**
5. Side-effects
6. Skipping Bad Records
7. Local Execution
8. Status Information
9. Counters

Supports self defined input output type, as long as you provide a reader interface

Refinements

1. Partitioning Function
2. Ordering Guarantees
3. Combiner Function
4. Input and Output Types
5. **Side-effects**
6. Skipping Bad Records
7. Local Execution
8. Status Information
9. Counters

If you want to have auxiliary files, make the writes atomic and idempotent

Refinements

1. Partitioning Function
2. Ordering Guarantees
3. Combiner Function
4. Input and Output Types
5. Side-effects
6. **Skipping Bad Records**
7. Local Execution
8. Status Information
9. Counters

In this mode, if multiple failures happen on one record, it will be skipped in next attempt

Refinements

1. Partitioning Function
2. Ordering Guarantees
3. Combiner Function
4. Input and Output Types
5. Side-effects
6. Skipping Bad Records
7. **Local Execution**
8. Status Information
9. Counters

Basically allows you debug your mapper and reducer locally

Refinements

1. Partitioning Function
2. Ordering Guarantees
3. Combiner Function
4. Input and Output Types
5. Side-effects
6. Skipping Bad Records
7. Local Execution
8. **Status Information**
9. Counters

Informs the user of running status

Refinements

1. Partitioning Function
2. Ordering Guarantees
3. Combiner Function
4. Input and Output Types
5. Side-effects
6. Skipping Bad Records
7. Local Execution
8. Status Information
9. **Counters**

Mostly used for sanity checking.
Some counters are computed automatically.

Implementation Environment

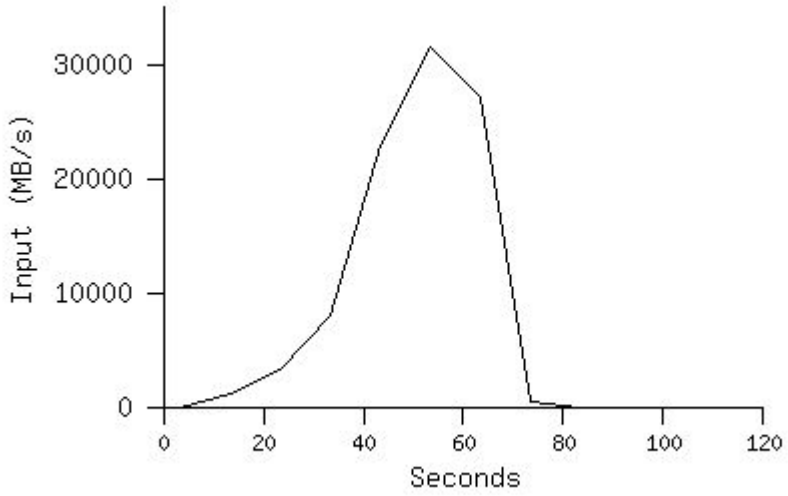
- Machines: dual-processor running Linux, 2-4 GB memory
- Commodity Networking Hardware: 100 MB/s or 1 GB/s, averaging less
- Cluster: hundreds or thousands of machines → Common Machine Failure
- Storage: disks attached to machines
- File System: GFS
- Users submit jobs(consists of tasks) to scheduler,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

Performance

Using 1,800 machines

- Grep: 150 sec through 10^10 100-byte records
- Sort: 891 sec of 10^10 100-byte records

MR_GREP

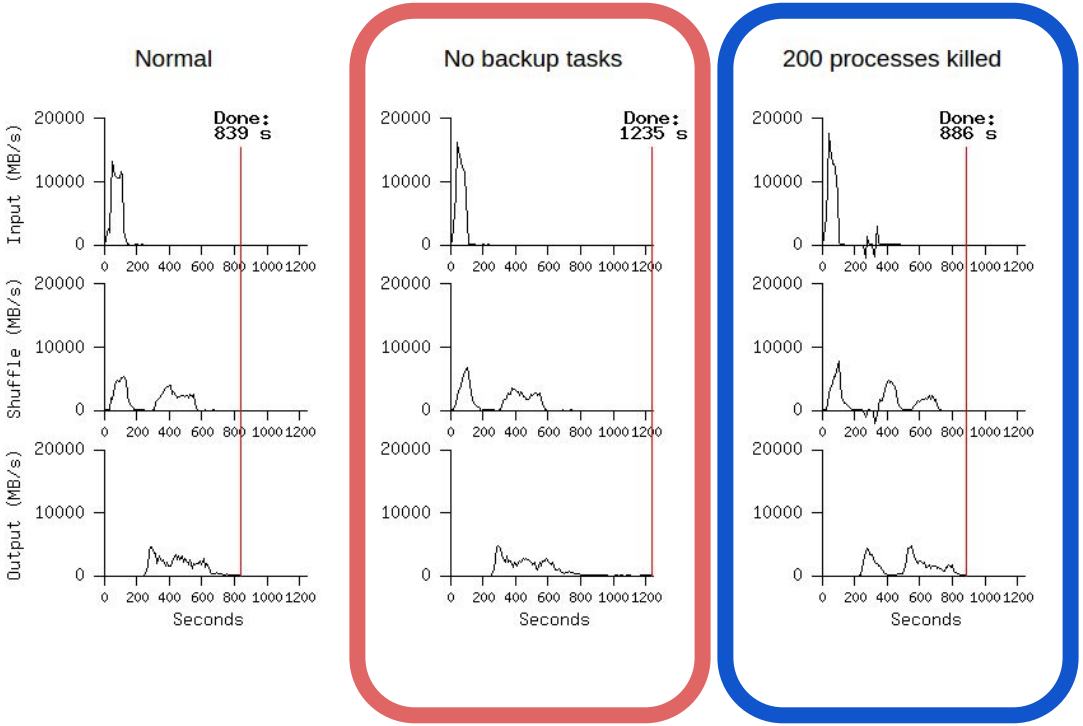


Locality helps:

- 1800 machines read 1 TB of data at peak of ~31 GB/s
- Without this, rack switches would limit to 10 GB/s

Startup overhead is significant for short jobs

MR_SORT



Backup helps

Fault Tolerance Works

What is MapReduce?

MapReduce is a software framework for easily writing applications which process vast amounts of data (multi-terabyte data-sets) in-parallel on large clusters (thousands of nodes) of commodity hardware in a reliable, fault-tolerant manner.

https://hadoop.apache.org

MR is more like an extract-transform-load (ETL) system than a DBMS, as it quickly loads and processes large amounts of data in an ad hoc manner. As such, it complements DBMS technology rather than competes with it.

MapReduce and Parallel DBMSs: Friends or Foes?
Michael Stonebraker et al.

Limitations

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 (e.g. iterative machine learning & graph processing)
2. More interactive ad-hoc queries

These tasks require reusing data between jobs.

Limitations

Iterative algorithms and interactive data queries both require one thing that MapReduce lacks:

> Efficient **data sharing** primitives

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.

Motivation for a new system

Memory is much faster than disk

Goal: keep data in memory and share between jobs.

Challenge: a distributed memory abstraction that is **fault tolerant** and **efficient**

Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing

Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing

NSDI 2012

Awarded Best Paper!

2345 citations

Matei Zaharia, Assistant Professor, Stanford CS

Mosharaf Chowdhury, Assistant Professor, UMich EECS

Tathagata Das, Software Engineer, Databricks

Ankur Dave, PhD, UCB

Justin Ma, Software Engineer, Google

Murphy McCauley, PhD, UCB

Michael J. Franklin, Professor, UCB CS

Scott Shenker, Professor, UCB CS

Ion Stoica, Professor, UCB CS

Resilient Distributed Datasets

Restricted form of distributed shared memory

- Immutable, partitioned collections of records
- Can only be built through **coarse-grained** deterministic operations
  - i.e. **Transformations** (map, filter, join,…)

Efficient fault recovery using **lineage**

- Lineage: transformations used to build a data set
- Recompute lost partitions on failure using the logged functions
- Almost no cost if nothing fails

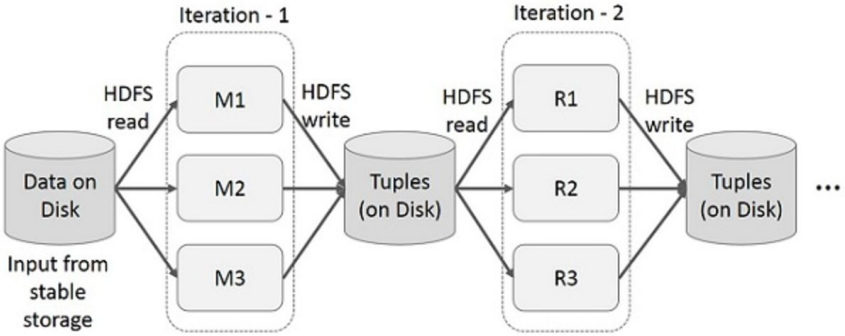
Spark Programming Interface

Provides:

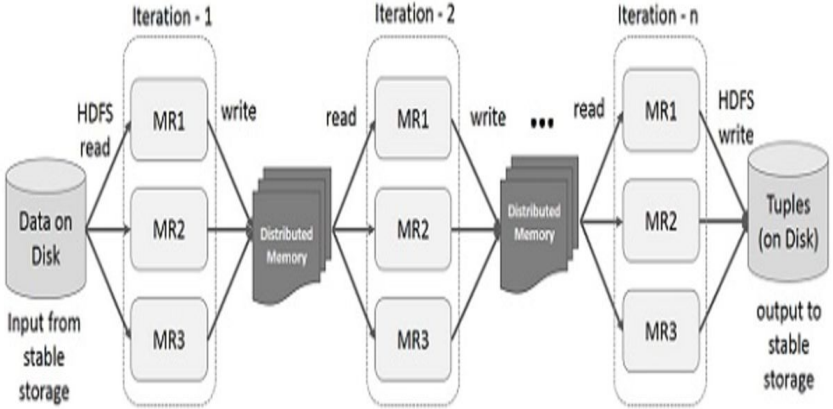
1. Resilient Distributed Datasets (RDDs)
2. Operations on RDDs: transformations (build new RDDs), actions (compute and output results)
3. Control of each RDD's
   a. Partitioning (layout across nodes)
   b. Persistence (storage in RAM, on disk, etc)

Iterative Operations

on MapReduce

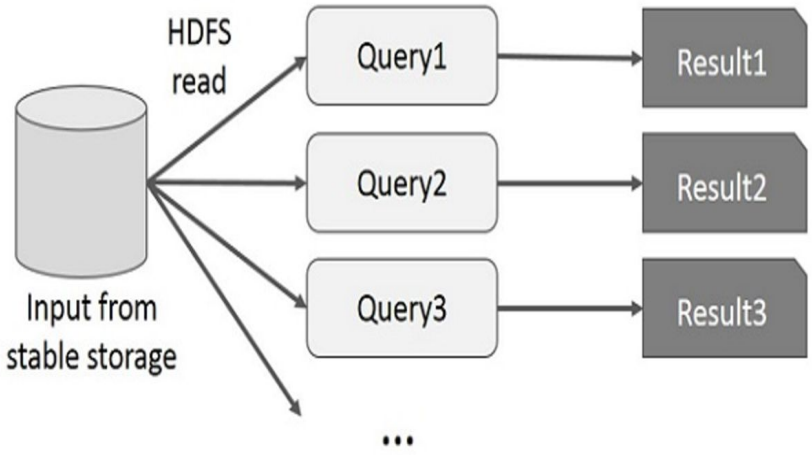


on Spark RDD

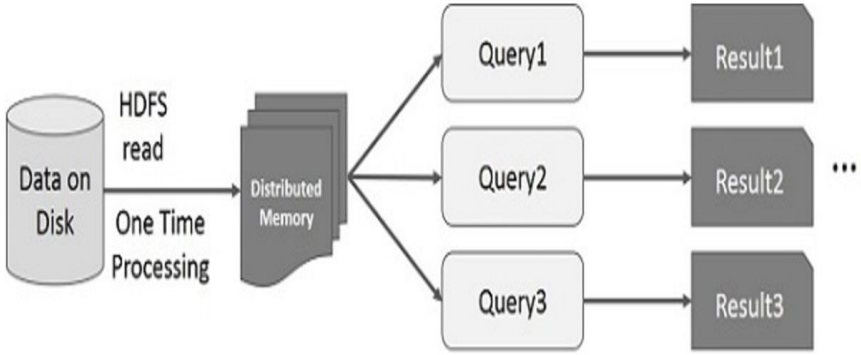


Interactive Operations

on MapReduce

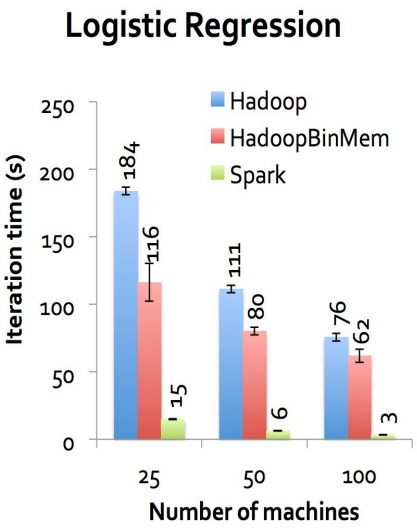


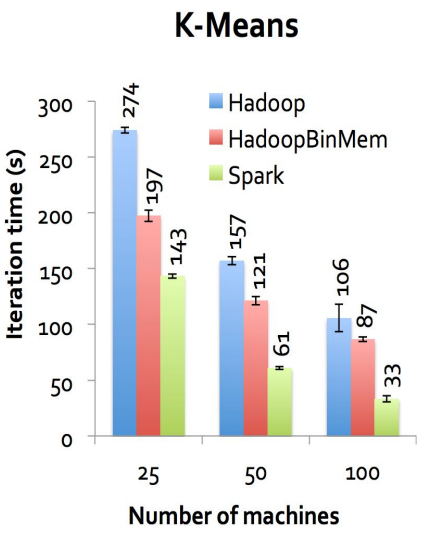
on Spark RDD



Evaluation

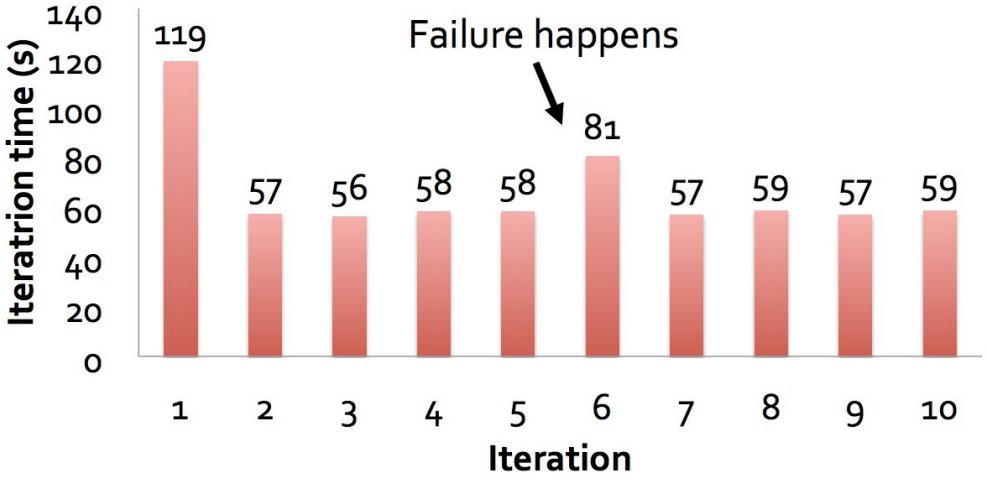
Spark outperforms Hadoop by up to 20x in iterative machine learning and graph applications.





Evaluation

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

Conclusions

1. MapReduce
    a. A simple and powerful interface that enables automatic parallelization and distribution of large-scale computations.
    b. Achieves high performance on large clusters of commodity PCs.
    c. Implemented based on Google's infrastructure. (highly engineered accordingly)
    d. The frequent disk I/O and data replication limits its usage in iterative algorithm and interactive data queries.

2. Spark RDD

    e. A Fault-Tolerant Abstraction for In-Memory Cluster Computing
    f. Recovers data using lineage instead of replication
    g. performs much better on iterative computations and interactive data queries.
    h. Large memory consumption is the main bottleneck.

Reference

1. “Take a close look at MapReduce”, Xuanhua Shi
2. “MapReduce: Simplified Data Processing on Large Clusters”, Jeffery Dean and Sanjay Ghemawat
3. “Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In Memory Cluster Computing”, Matei Zaharia et al.