MapReduce:

Simplified Data Processing on Large Clusters

Jeffrey Dean and Sanjay Ghemawat
Google Inc.
OSDI 2004

Presented by
Drumil Deliwala, Xin He, Yunsong Liu, Yue Cheng Tsang, Yifan Zhao

AGENDA

Introduction + Programming Model

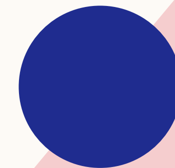
Implementation

Refinements

Performance + Experience

Related Work

Conclusions



INTRODUCTION

What is MapReduce?
A programming model created by Google to simplify large-scale data processing

Problem It Solves
Managing massive data, such as web docs and logs, across many machines is complex
Traditional methods need careful handling of parallel tasks, failures, and data sharing

INTRODUCTION

Key Idea

- Splits large data processing into smaller tasks
- Handles distribution and errors automatically
- Runs tasks across many machines in parallel

Impact

- Scales to thousands of machines, processing terabytes daily
- Widely used across Google for data mining, machine learning, and indexing

PROGRAMMING MODEL

Model

- User defines two main functions: Map and Reduce
  - **Map**: breaks data into key/value pairs
  - **Reduce**: combines values for each unique key from Map output

Execution Flow

- Split Data -> Map Phase -> Shuffle -> Reduce Phase

Handling Failures

- Failed tasks are automatically re-run on other machines

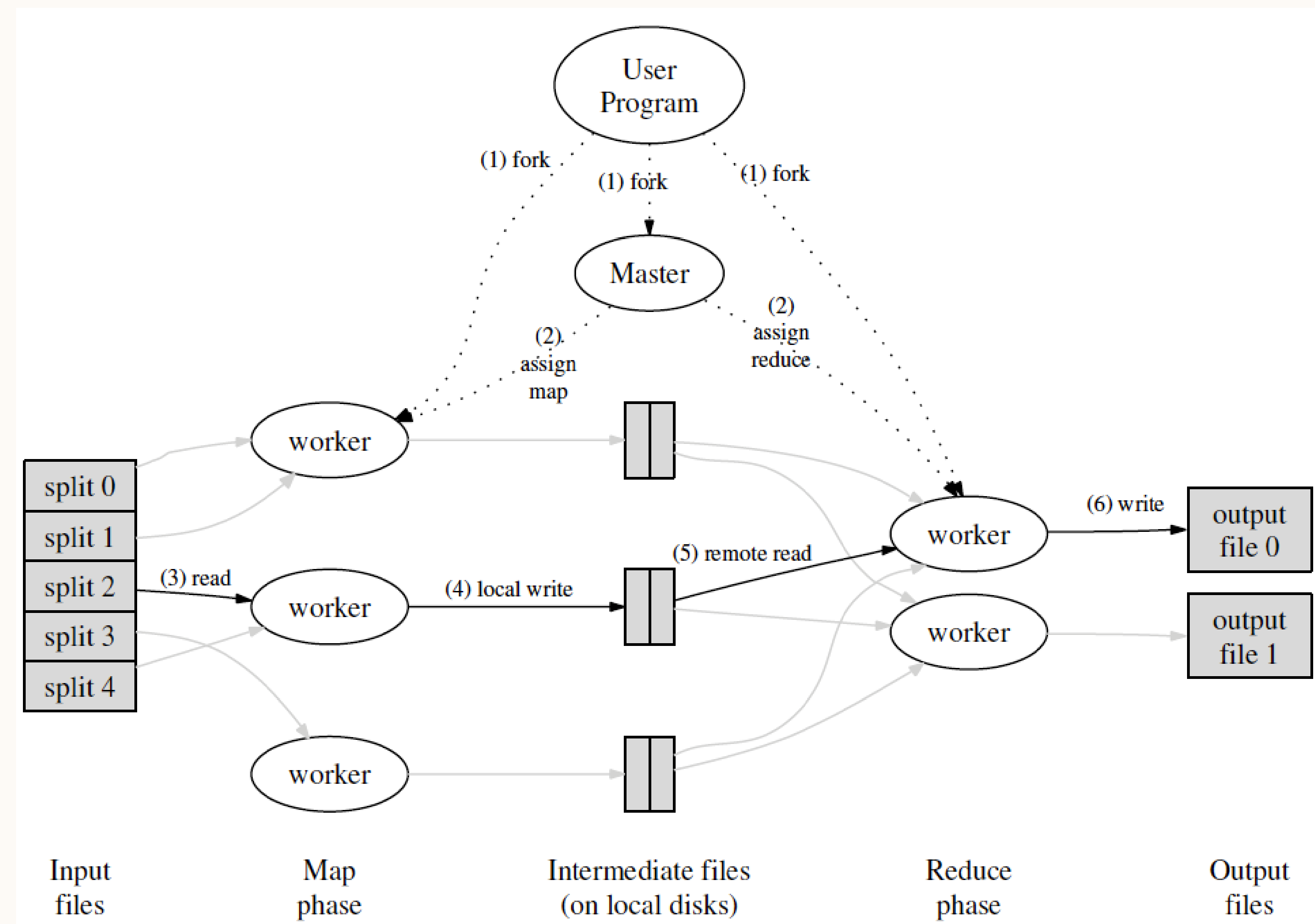


Figure 1: Execution overview

PROGRAMMING MODEL

Example: Word Count

- Map:

```
Input: (file, content)
"Hello world hello"
Output:
(hello, 1)
(world, 1)
(hello, 1)
```

- Reduce:

```
Input: (hello, [1,1])
Output: (hello, 2)
Input: (world, [1])
Output: (world, 1)
```

Types

- Map: `map (k1,v1) → list(k2,v2)`

  k1,v1: input key-value pairs

  k2,v2: intermediate output key-value pairs

- Reduce: `reduce (k2,list(v2)) → list(v2)`

  k2: intermediate key

  list(v2): list of all values associated with that key

  output: merged list of values

IMPLEMENTATION

1. EXECUTION OVERVIEW

1. **Splitting and Initialization:**

   MapReduce splits input into chunks, with a master assigning tasks to workers.

2. **Map Execution:**

   Workers parse data, apply the Map function, and save results to disk in partitions.

3. **Data Retrieval:**

   Reduce workers fetch and sort intermediate data by keys.

4. **Reduce Execution:**

   Reduce function processes keys and writes final output files.

5. **Completion:**

   Master signals job completion, producing R output files.

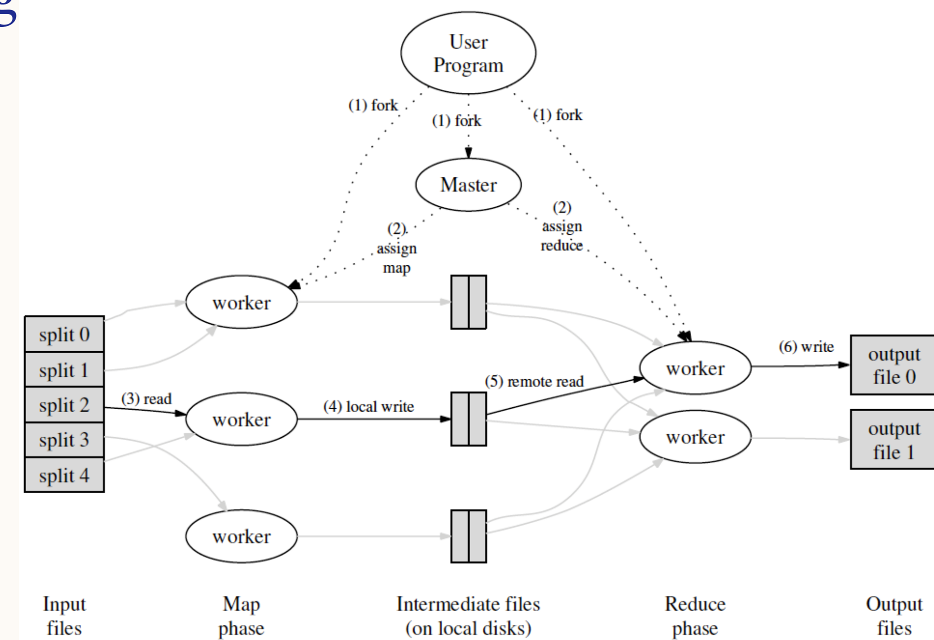


Figure 1: Execution overview

2. MASTER DATA STRUCTURES

The master tracks task status (idle, in-progress, completed) and assigned workers,

along with locations of intermediate data.

As map tasks complete, it updates reduce workers for efficient data access.

3. FAULT TOLERANCE

- **Worker Failure**:
  - The master pings workers to monitor status.
  - If a worker fails, tasks are reassigned;
  - map tasks are redone, while reduce tasks remain as their output is in a global file system.

- **Master Failure**:
  - Master failure typically aborts the job,
  - though checkpoints enable recovery if used, and clients can retry.

- **Failure Semantics**:
  - MapReduce ensures consistent results with deterministic functions,
  - using temporary files for sequential consistency despite failures.

4. LOCALITY

- **Local Data Storage:**
  - Data blocks are stored on local machine disks.
- **Optimized Scheduling:**
  - Master assigns tasks to machines with or near the data.
- **Reduced Network Load:**
  - Local processing minimizes network usage.

5. TASK GRANULARITY

- **Task Subdivision and Load Balancing:**
  - Map and reduce phases are split into many small tasks,
  - usually more than the number of workers,
  - for better load balancing and quick reassignment after failures.
- **Practical Limits:**
  - M and R values depend on master memory,
  - with common task sizes of 16-64 MB and
  - R often matching the worker count.

6. BACKUP TASK

- **Straggler Management:**
  The master launches backup copies for slow tasks (stragglers) near job completion, reducing delays with minimal extra resources.

REFINEMENTS

- Enhancement to the core MapReduce framework
- Address efficiency, flexibility, and anomaly handling
- Optimize performance for large-scale data processing

REFINEMENT
A. PARTITIONING FUNCTION

- **Default:** Hash function for balanced data distribution
- **Custom partitioning** option for specific data organization
  - **Example:** Grouping URLs by host
- **Benefit:** Efficient data management and load balancing

REFINEMENT
B. COMBINER FUNCTION

- **Purpose**: Local aggregation to reduce data sent to Reduce phase
    - **Example**: Word count with partial sums on each map node
- **Impact**: Lowers network traffic, reduces load on reducers
- **Key Benefit**: Enhanced efficiency in cases with redundant data

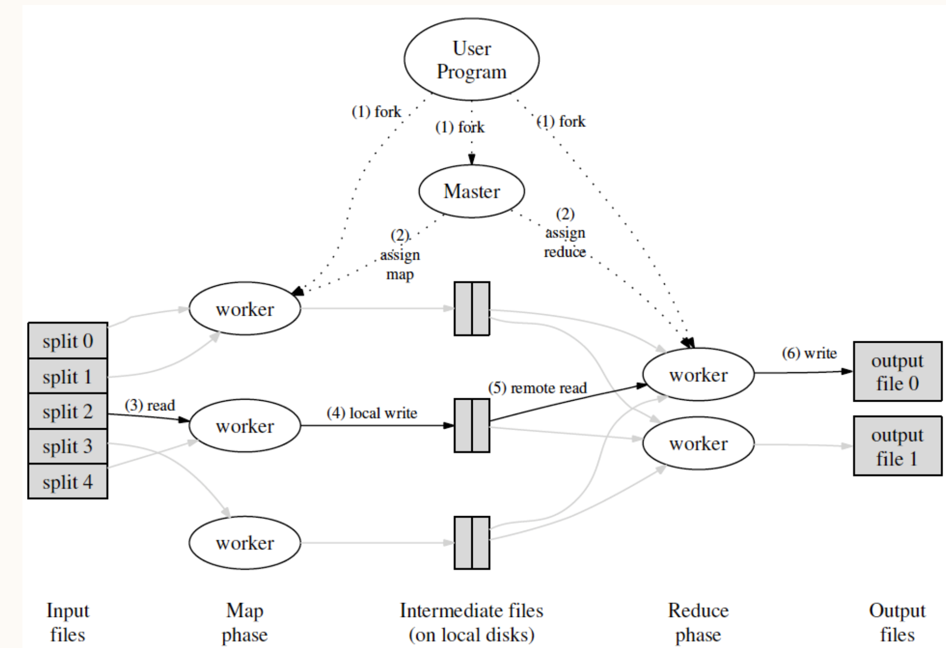


Figure 1: Execution overview

REFINEMENT
C. STATUS INFORMATION

- Real-time monitoring of job progress and worker performance
- Key metrics: Task completion, input/output rates, worker status
- **Value**: Enables efficient job management and timely intervention

REFINEMENT
D. COUNTERS

- Track occurrences of specific events during execution
- Custom counters for metrics
- **Example**: Counting records by language in text processing
- **Benefit**: Quality control, performance tuning, and debugging insights

REFINEMENT E. OTHERS

- **Ordering Guarantees**: Sorted processing within each partition
- **Input/Output Types**: Support for various formats, including databases
- **Side Effects**: Auxiliary file outputs, consistency not guaranteed
- **Skipping Bad Records**: Skip problematic records to ensure job completion

PERFORMANCE + EXPERIENCE

TESTING SETUP

- Grep program and Sort program
- Testing cluster
  - ~1800 machines
  - 4GB memory
  - Two 160GB disks
  - Gigabit Ethernet link
- Test Data – 1TB of data as 100-byte records
- Parameters – 15000 Map tasks, 4000 Reduce tasks

PERFORMANCE

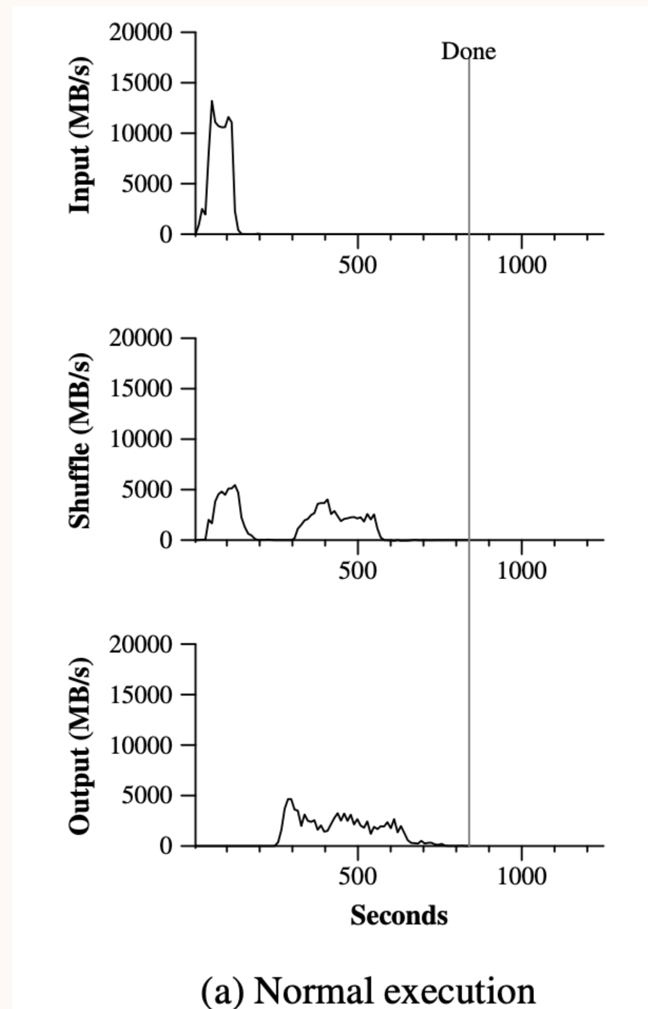


Figure 3: Data transfer rates over time for different executions of the sort program

- Input rate increases as Map tasks begin and then decreases as tasks finish
- Input rate increases as Reduce tasks fetch intermediate KV pairs, falls when Reduce tasks are in progress, and rises again when earlier Reduce tasks finish and new Reduce data.
- Input rate increases when the completed Reduce tasks write to output

Total time = 891s (similar to the best benchmark performance at the time)

OTHER TESTS AND INDICATORS

- Backup tasks disabled ---> 44% higher execution time
- 200 Machine failures ---> only 5% overhead
- Ease of use
  - Sort program took <50 lines of user code
  - Wide adoption at Google
  - Rewrite of the Google search engine indexing system using MapReduce

Number of jobs	29,423
Average job completion time	634 secs
Machine days used	79,186 days
Input data read	3,288 TB
Intermediate data produced	758 TB
Output data written	193 TB
Average worker machines per job	157
Average worker deaths per job	1.2
Average map tasks per job	3,351
Average reduce tasks per job	55
Unique *map* implementations	395
Unique *reduce* implementations	269
Unique *map/reduce* combinations	426

Table 1: MapReduce jobs run in August 2004

RELATED WORKS

- **Google File System**(GFS), a distributed storage for storing large datasets; **Bigtable**, a distributed database optimized for storing structured data.
- **Hadoop** (2006): open-source project that fills MapReduce's gaps in underlying storage and cluster management components; the distributed storage component is inspired by GFS.
- **Dryad** (Microsoft Research, 2007): a more general version of MapReduce. Instead of fixed Map & Reduce programs, Dryad can have a Directed Acyclic Graph (DAG) of programs.

CONCLUSION

- Simplified large-scale data processing through a clean abstraction for parallelization
- Enabled processing of petabytes of data on commodity hardware clusters
- Proved highly scalable and fault-tolerant, becoming the foundation for modern big data systems

STUDY QUESTIONS

- **Question 1:** *Why was the MapReduce model considered groundbreaking for large-scale data processing at the time it was introduced?*

- **Question 2 :** *How does the MapReduce model achieve fault tolerance, and why is re-execution chosen over traditional checkpointing methods?*

Q & A

Thank you!