MapReduce

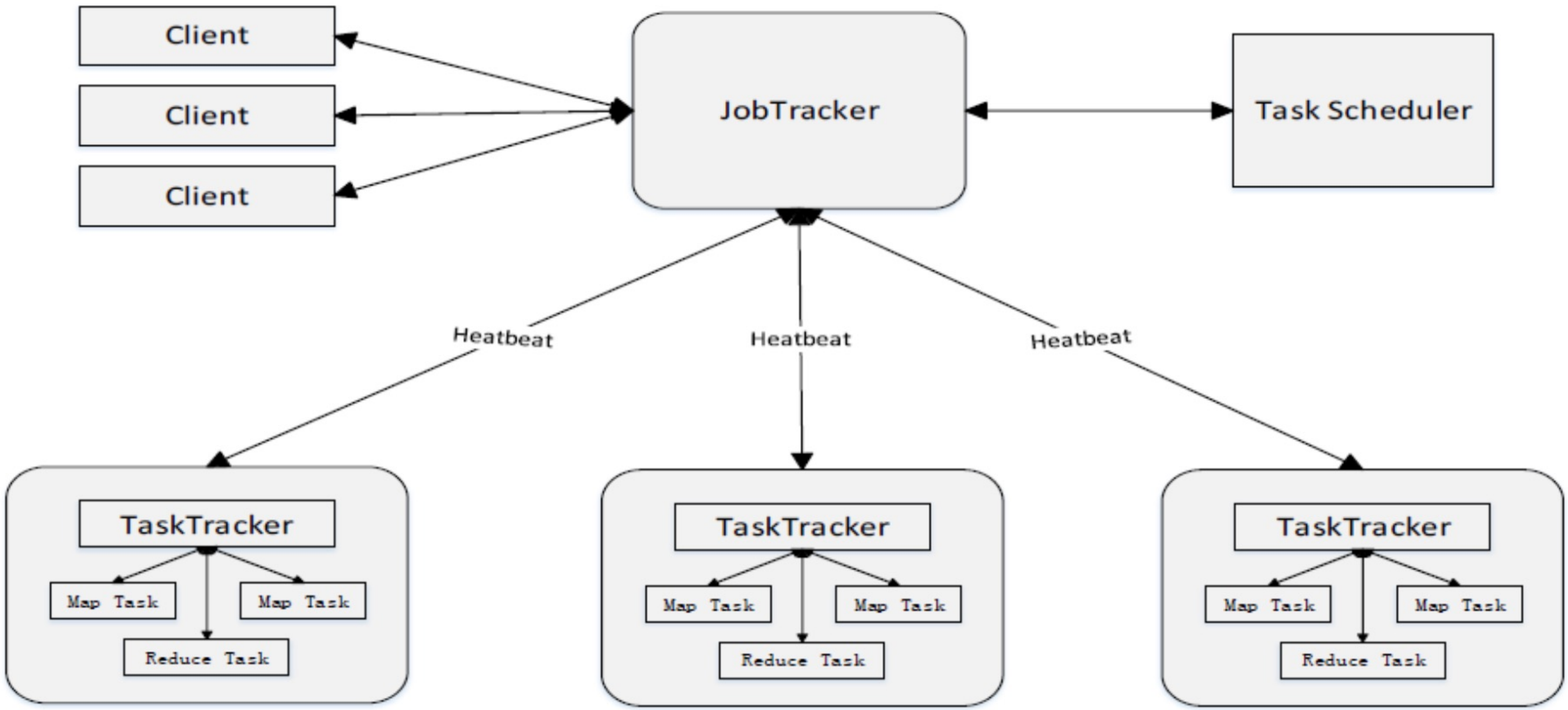
大数据分析 | 何铁科
http: // hetieke.cn
南京大學
NANJING UNIVERSITY

Goals of Mapreduce

1. **Parallelization**
2. **Fault tolerance**
3. **Data distribution**
4. **Load balancing**

Motivation

- ❑ **Need for computing large-scale data**
- ❑ **Hide the details of the library**
- ❑ **Parallelization, Tolerance, Distribution and Load balancing**
- ❑ **Inspired by the map and reduce primitives present in Lisp**



Client
Client
Client
JobTracker
Task Scheduler
Heatbeat
Heatbeat
Heatbeat
TaskTracker
Map Task
Map Task
Reduce Task
TaskTracker
Map Task
Map Task
Reduce Task
TaskTracker
Map Task
Map Task
Reduce Task

Main components

- Client
- JobTracker
- TaskTracker
- Task

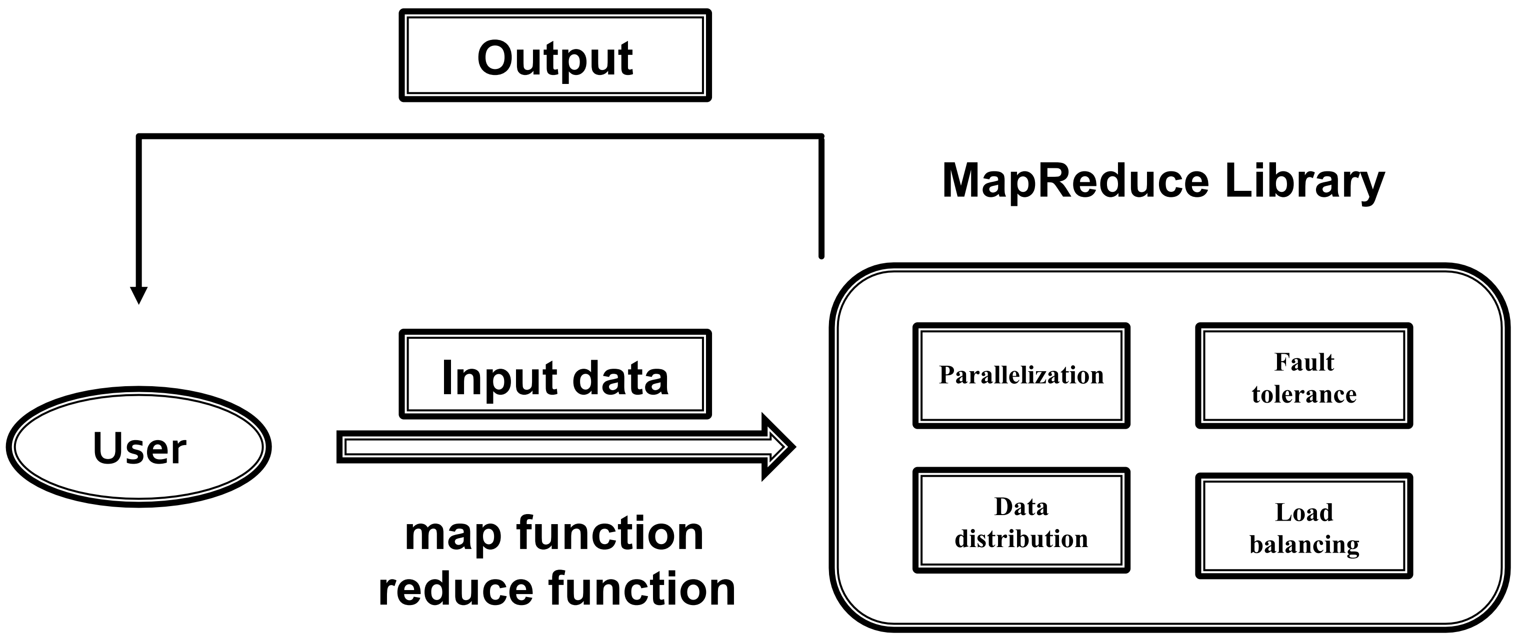
1^{st} Computing

1. The input data is usually large.

2. The computations have to be distributed across hundreds of machines.

3. Obscure the original simple computation.

2nd Hidding



Output
MapReduce Library
User
Input data
Parallelization
Fault tolerance
Data distribution
Load balancing
map function
reduce function

Programming Model

- **MapReduce** is a **programming model** proposed by Google for processing and generating large data sets.

- The framework contains two user-implemented interfaces： **Map and Reduce.**

- Map receives a key-value pair and generates a collection of intermediate key-value pairs.

- Reduce receives this intermediate key and the collection of values of the key, merges the values together, and produces a smaller set of values.

Example

Problem of counting the number of occurrences of each word in a large collection of documents.

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

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

Types

map	`(k1,v1)`	$\rightarrow$ `list(k2,v2)`
reduce	`(k2,list(v2))`	$\rightarrow$ `list(v2)`

- The input keys and values are drawn from a different domain than the output keys and values.

- The intermediate keys and values are drawn from the same domain as the output keys and values.

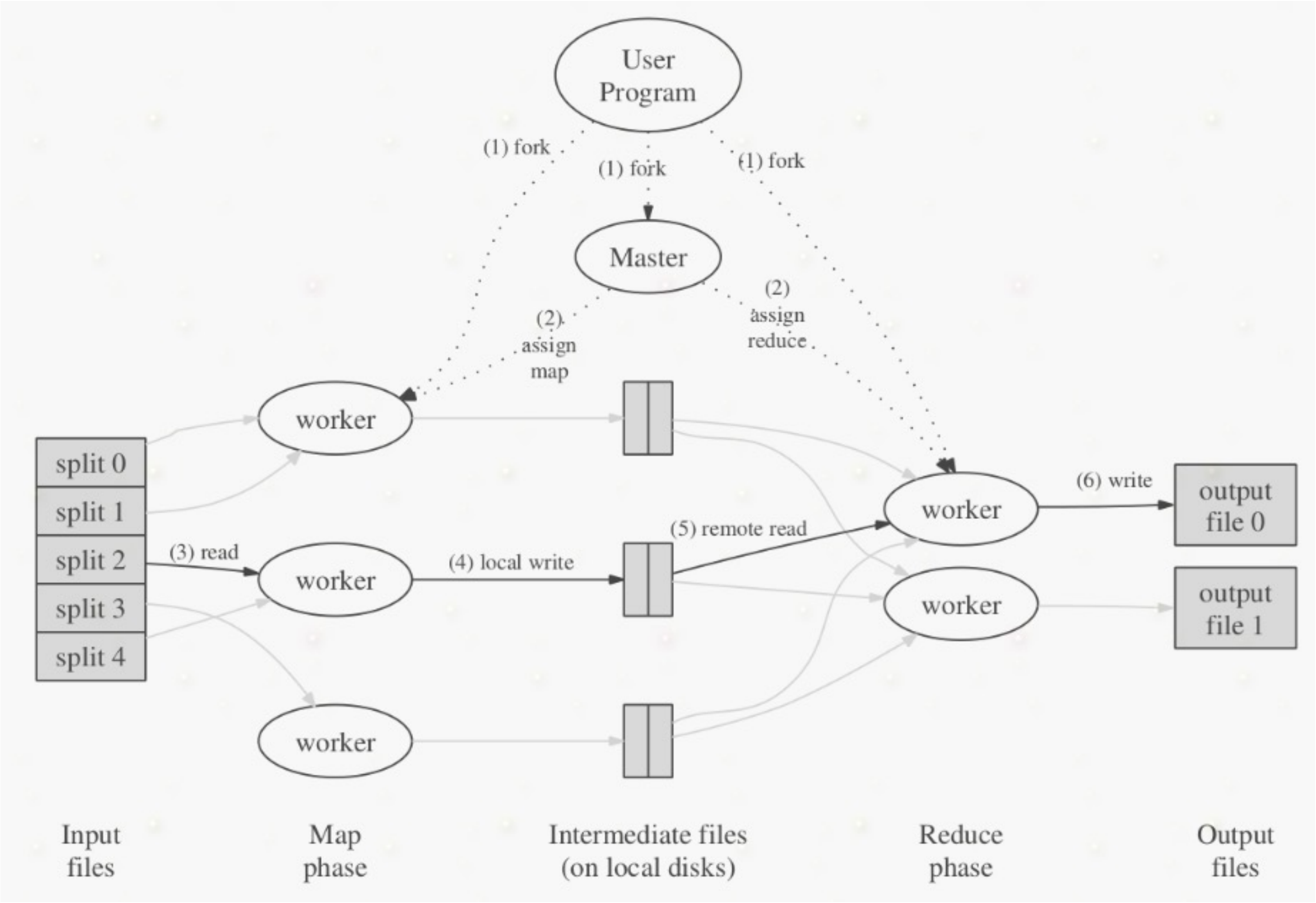
More Examples

1. Distributed Grep
2. Count of URL Access Frequency
3. Reverse Web-Link Graph
4. Term-Vector per Host
5. Inverted Index
6. Distributed Sort

The Whole Design

1. **Execution Overview**
2. **Master Data Structures**
3. **Fault Tolerance**
4. **Locality**
5. **Task Granularity**
6. **Backup Tasks**

1. Execution Overview



2. Master Data Structures

- Map and Reduce task
  - state (idle、in-progress、completed)
  - the identity of the worker machine
- Master is the conduit through which the location of intermediate file regions is propagated from map tasks to reduce tasks

3. Fault Tolerance

- Worker failure
  - All map tasks completed on this worker are reset to idle state and handed over to other workers to execute these map tasks.
  - The *MAP* task or reduce task that is executing on this worker is reset to the idle state and waits for rescheduling.
- Master failure
  - The current implementation chooses to interrupt the MapReduce calculation.

4. Locality

- **The input data is stored on the local hard disks.**
- **GFS splits each file into blocks of size 64MB,**
- **GFS saves multiple copies of each block (usually 3 copies in different machines).**
- **Master will try to execute the map task on the machine that contains a copy of the relevant input data.**
- **If the task fails, the master attempts to save network bandwidth by executing the map task on a neighboring machine.**

5. Task Granularity

- **Map phase into M pieces and reduce phase into R pieces.**
- **Ideally, M and R should be much larger than the number of worker machines to improve dynamic load balancing.**
- **Actually, M and R have limits in practical implementation.**
- **Master must make O(M + R) scheduling decisions and keeps O(M ∗ R)state in memory.**

6. Backup Tasks

- This mode is designed to mitigate the **straggler problem.**

- **Straggler problem**—a machine takes an unusually large amount of time to complete the last few map, resulting in longer computation times.

- **Solution**
  - when a *MapReduce* **is nearing completion**, the master executes a standby task **for the task in progress**,
  - marks task as complete when the task completes, whether the primary task or the standby task completes.

Refinements

- **Partitioning Function**
- **Ordering Guarantees**
- **Combiner Function**
- **Input and Output Types**
- **Side-effects**
- **Skipping Bad Records**
- **Local Execution**
- **Status Information**
- **Counters**

1. Partitioning Function

- **specify the number of reduce tasks/output files that they desire**
- **uses hashing（e.g. “hash(key) mod R”）**
- **result in fairly well-balanced partitions**

2. Ordering Guarantees

- **within a given partition, the intermediate key/value pairs are processed in increasing key order**
- **easy to generate a sorted output file per partition**

3. Combiner Function

- **allow the user to specify an optional Combiner function that does partial merging of this data before it is sent over the network**
- **executed on each machine that performs a map task**
- **the same code as the reduce functions**
- **the only difference between reduce and combiner function is how to handle the output of the function**
  - **reduce function: written to the final output file**
  - **combiner function: written to an intermediate file that will be sent to a reduce task**

4. Input and Output Types

- **provides support for reading input data in several different formats**
- **split input into meaningful ranges for processing as separate map tasks**
- **add support for a new input type by providing an implementation of a simple reader interface**
- **A reader does not necessarily need to provide data read from a file**
- **support a set of output types for producing data in different formats**

5. Side-effects

- **convenient to produce auxiliary files as additional outputs from their map and/or reduce operators**
- **rely on the application writer to make such side-effects atomic and idempotent**
- **the application writes to a temporary file and atomically renames this file once it has been fully generated**
- **do not provide support for atomic two-phase commits of multiple output files produced by a single task**
- **tasks that produce multiple output files with cross-file consistency requirements should be deterministic**
- **this restriction has never been an issue in practice**

6. Skipping Bad Records

- **acceptable to ignore a few bad records**
- **provide an optional mode of execution where the MapReduce library detects which records cause deterministic crashes and skips these records in order to make forward progress**

- **Each worker process installs a signal handler that catches segmentation violations and bus errors**
- **the MapReduce library stores the sequence number of the argument in a global variable**
- **If the user code generates a signal, the signal handler sends a “last gasp” UDP packet that contains the sequence number to the MapReduce master.**
- **When the master has seen more than one failure on a particular record, it means that the record should be skipped**

7. Local Execution

- **debugging problems in Map or Reduce functions can be tricky when the actual computation happens in a distributed system**
- **develop an alternative implementation of the MapReduce library that sequentially executes all of the work for a MapReduce operation on the local machine**

8. Partitioning Function

- **show the progress of the computation**
- **contain links to the standard error and standard output files generated by each task**
- **can use this data to predict how long the computation will take, and whether or not more resources should be added to the computation**
- **be used to figure out when the computation is much slower than expected**
- **the top-level status page shows which workers have failed, and which map and reduce tasks they were processing when they failed**

9. Counters

- **provides a counter facility to count occurrences of various events**
- **To use this facility, user code creates a named counter object and then increments the counter appropriately in the Map and/or Reduce function**
- **eliminates the effects of duplicate executions of the same map or reduce task to avoid double counting**
- **Some counter values are automatically maintained by the MapReduce library**
- **useful for sanity checking the behavior of MapReduce operations**

Performance

- Cluster Configuration
- Grep(Globally search a Regular Expression and Print)
- Sort
- Effect of Backup Tasks
- Machine Failures

1. Cluster Configuration

- executed on a cluster that consisted of approximately 1800 machines
- All the machines:
  - two 2GHz Intel Xeon processors with Hyper-Threading enabled
  - 4GB of memory, two 160GB IDE disks, and a gigabit Ethernet link
  - arranged in a two-level tree-shaped switched network with approximately 100-200 Gbps of aggregate band-width available at the root
  - in the same hosting facility
  - approximately 1-1.5GB was reserved
- were executed on a weekend afternoon

2. Grep

- **scans through 10^10 100-byte records, searching for a relatively rare three-character pattern**
- **split into approximately 64MB pieces (M = 15000), and the entire output is placed in one file (R = 1)**

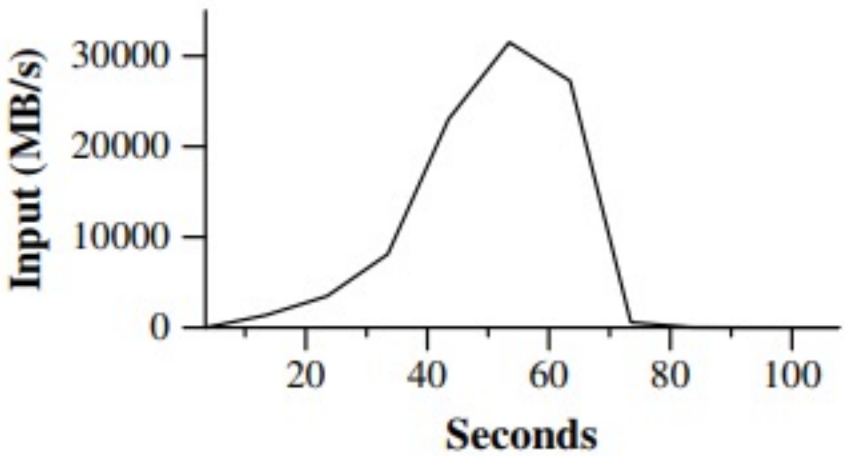
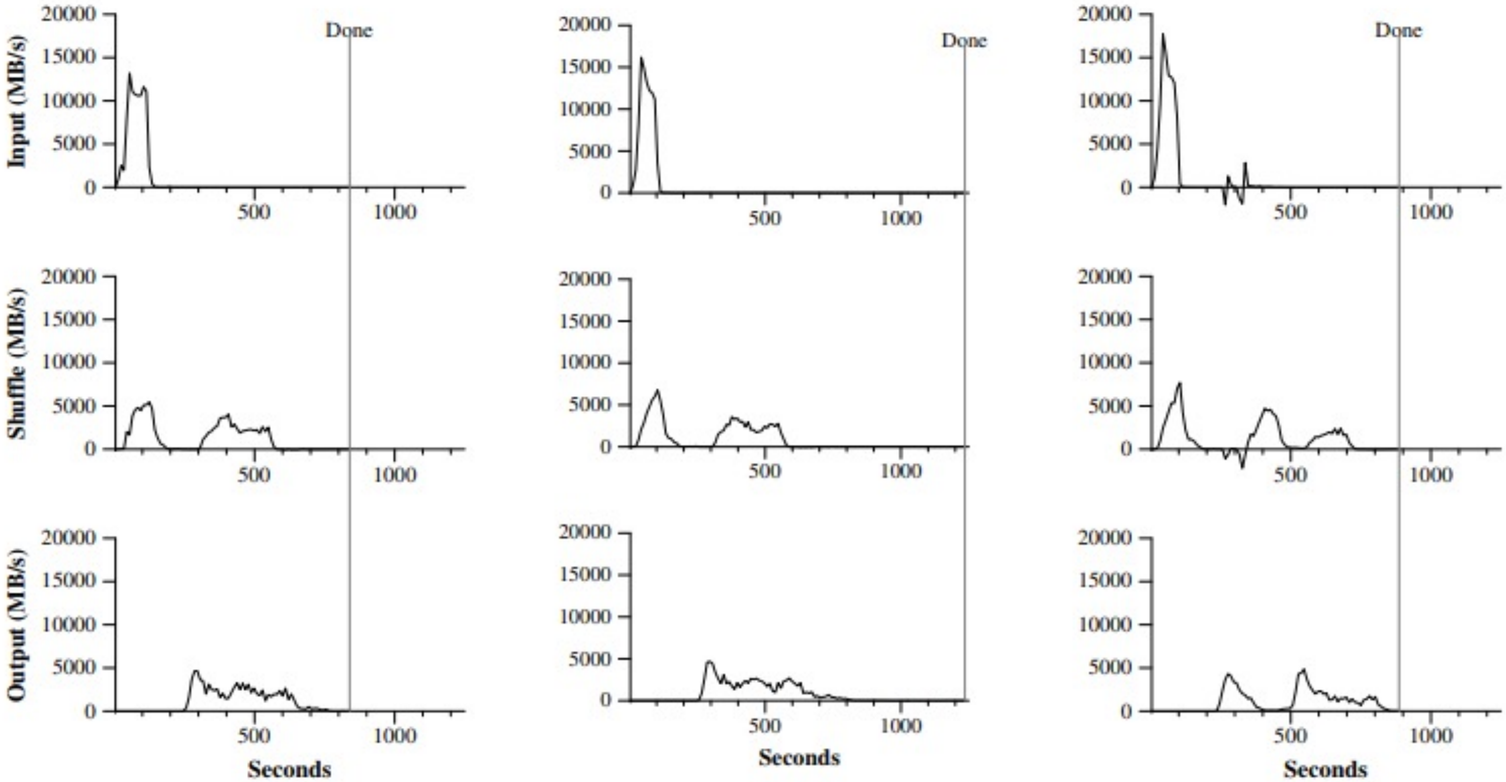


Figure 2: Data transfer rate over time

3. Sort

- **sorts 10^10 100-byte records (approximately 1 terabyte of data)**
- **A three-line Map function extracts a 10-byte sorting key from a text line and emits the key and the original text line as the intermediate key/value pair**
- **used a built-in Identity function as the Reduce operator**
- **passes the intermediate key/value pair unchanged as the output key/value pair**
- **partitioning function for this benchmark has built-in knowledge of the distribution of keys**
- **add a pre-pass MapReduce operation that would collect a sample of the keys**
- **use the distribution of the sampled keys to compute split-points for the final sorting pass**

3. Sort



(a) Normal execution (b) No backup tasks (c) 200 tasks killed

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

4. Effect of Backup Tasks

- **Figure 3 (b) shows an execution of the sort program with backup tasks disabled**
- **After 960 seconds, all except 5 of the reduce tasks are completed**
- **last few stragglers don’t finish until 300 seconds later**
- **the entire computation takes 1283 seconds, an increase of 44% in elapsed time**

5. Machine Failures

- Figure 3 (c) shows an execution of the sort program
- intentionally killed 200 out of 1746 worker processes several minutes into the computation
- the underlying cluster scheduler immediately restarted new worker processes on these machines
- the worker deaths show up as a negative input rate since some previously completed map work disappears and needs to be redone
- the re-execution of this map work happens relatively quickly
- the entire computation finishes in 933 seconds including startup overhead (just an increase of 5% over the normal execution time)

Experience

- **the first version of the MapReduce library in February of 2003, and made significant enhancements to it in August of 2003**
- **used across a wide range of domains within Google, including:**
  - **large-scale machine learning problems**
  - **clustering problems for the Google News and Froogle products**
  - **extraction of data used to produce reports of popular queries**
  - **extraction of properties of web pages for new experiments and products**
  - **large-scale graph computations**

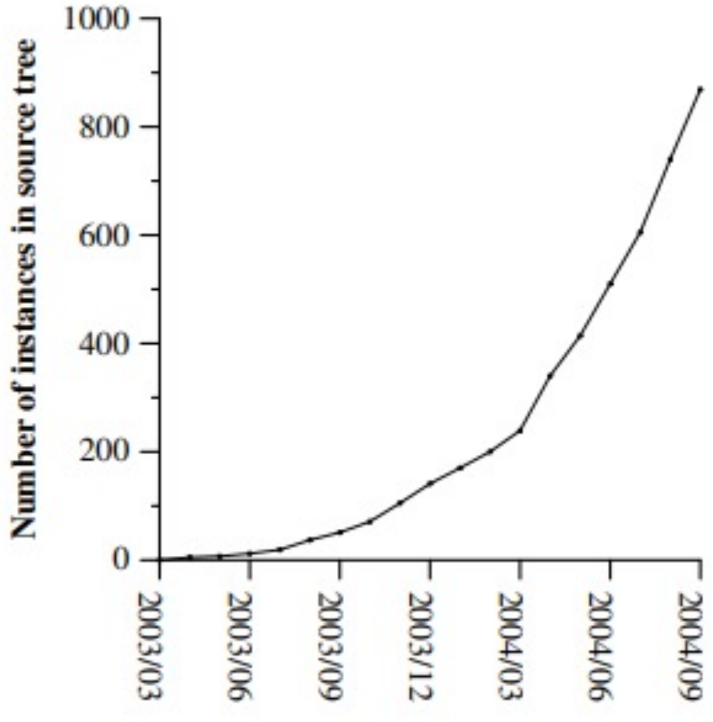


Figure 4: MapReduce instances over time

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

1. Large-Scale Indexing

- One of our most significant uses of MapReduce to date is a complete rewrite of the production indexing system
- provide several benefits：
  - The indexing code is simpler, smaller, and easier to understand
  - keep conceptually unrelated computations separate, instead of mixing them together to avoid extra passes over the data
  - has become much easier to operate
  - easy to improve the performance of the indexing process by adding new machines to the indexing cluster

Apply in google

- **can be considered a simplification and distillation of some of these models based on our experience with large real-world computations**
- **provide a fault-tolerant implementation that scales to thousands of processors**

- **Bulk Synchronous Programming and some MPI primitives provide higher-level abstractions that make it easier for programmers to write parallel programs**
- **exploit a restricted programming model to parallelize the user program automatically and provide transparent fault-tolerance**

- **locality optimization draws its inspiration from techniques such as active disks**
- **backup task mechanism is similar to the eager scheduling mechanism employed in the Charlotte System**
- **relies on an in-house cluster management system that is responsible for distributing and running user tasks on a large collection of shared machines**
- **the sorting facility that is a part of the MapReduce library is similar in operation to NOW-Sort**

- **River provides a programming model where processes communicate with each other by sending data over distributed queues**
- **BAD-FS has a very different programming model from MapReduce, and unlike MapReduce, is targeted to the execution of jobs across a wide-area network**
- **TACC is a system designed to simplify construction of highly-available networked services and relies on re-execution as a mechanism for implementing fault-tolerance**

Summary

MapReduce