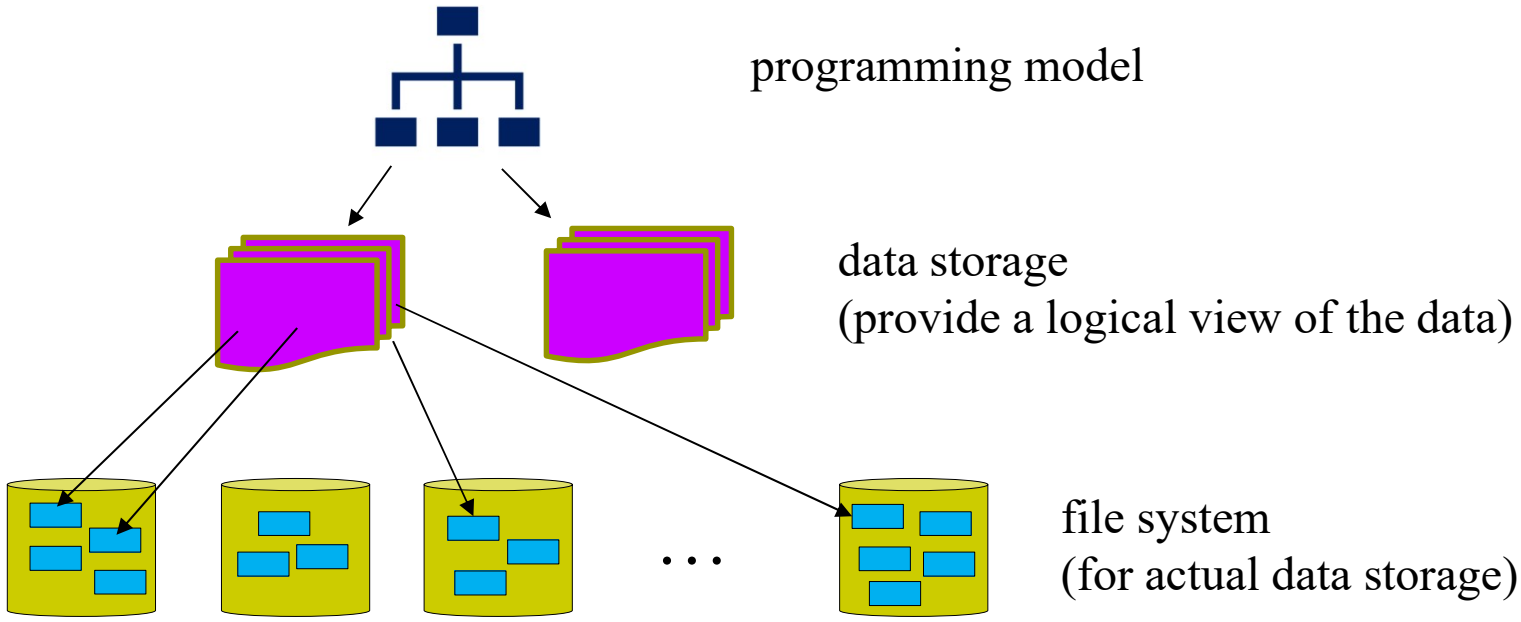
分散式系統雲端與雲端應用開發實務

MapReduce

Yuh-Jzer Joung
Dept. of Information Management
National Taiwan University

Building Bigdata Applications

- You need
  - a **programming model** for processing the data (e.g., MapReduce)
  - a **database-like storage system** for data read/write (e.g., Bigtable)
  - a **file system** to physically store the data (e.g., GFS)



Three parts in this unit

- Large distributed file systems
- Large distributed storage systems
- **MapReduce programming model**

MapReduce [Google 2004]

- A simple and powerful programming model (proposed by Google) that allows programmers to write a simple program to process a huge volume of data efficiently, and exploit thousands of machines without experience with distributed and/or parallel systems.
  - Programs rely only on two operations: **Map** and **Reduce**
  - Implementation hides details of parallelization, fault-tolerance, locality optimization, and load balancing.
  - Google uses it to do lots of things, including the indexing processing of its web search engine.

Reading. MapReduce: Simplified data processing on large clusters, J. Dean & S. Ghemawat, Communications of the ACM 2008, & OSDI 2004.

Typical Problems

- Analyze web server logs to find popular URLs
- Count the number of times each distinct word occurs in the corpus
- The above problems become challenging when the data set size is huge, and across hundreds of machines!

MapReduce Programming model

Two functions inspired by functional programming.

```
- fun map f [ ] = [ ]
    | map f (a::y) = (f a)::(map f y);
```

e.g.

```
map (fn(x)=>x*x, [1,2,3,4])
val it =[1,4,9,16]: int list
```

```
- fun reduce f [] v = v
    | reduce f (a::y) v = f(a, reduce f y v);
```

λ-calculus

e.g.

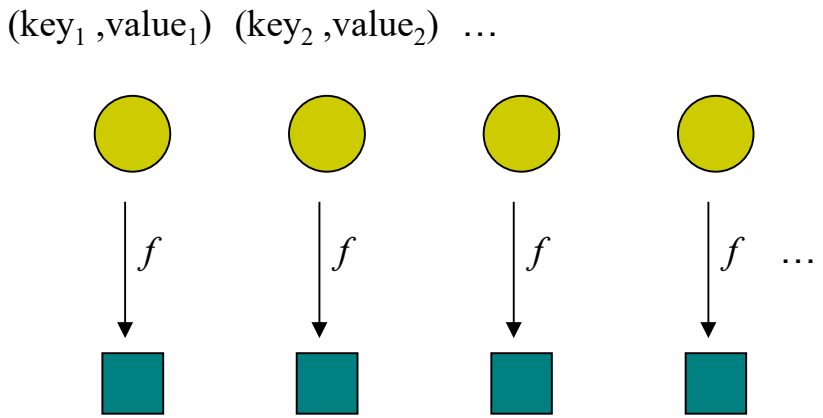
```
reduce (fn(x,y)=>x+y, [1,2,3,4], 0)
val it=10: int

reduce (fn(x,y)=>x*y, [1,2,3,4], 1)
val it=24: int
```

Map

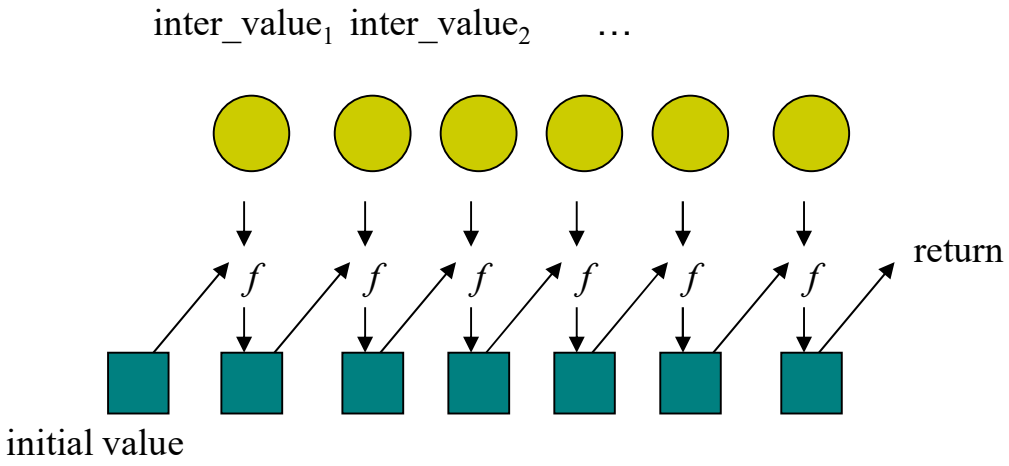
map (in_key, in_value) -> list (out_key, intermediate_value)

- Process input key/value pair
- Produce set of intermediate pairs

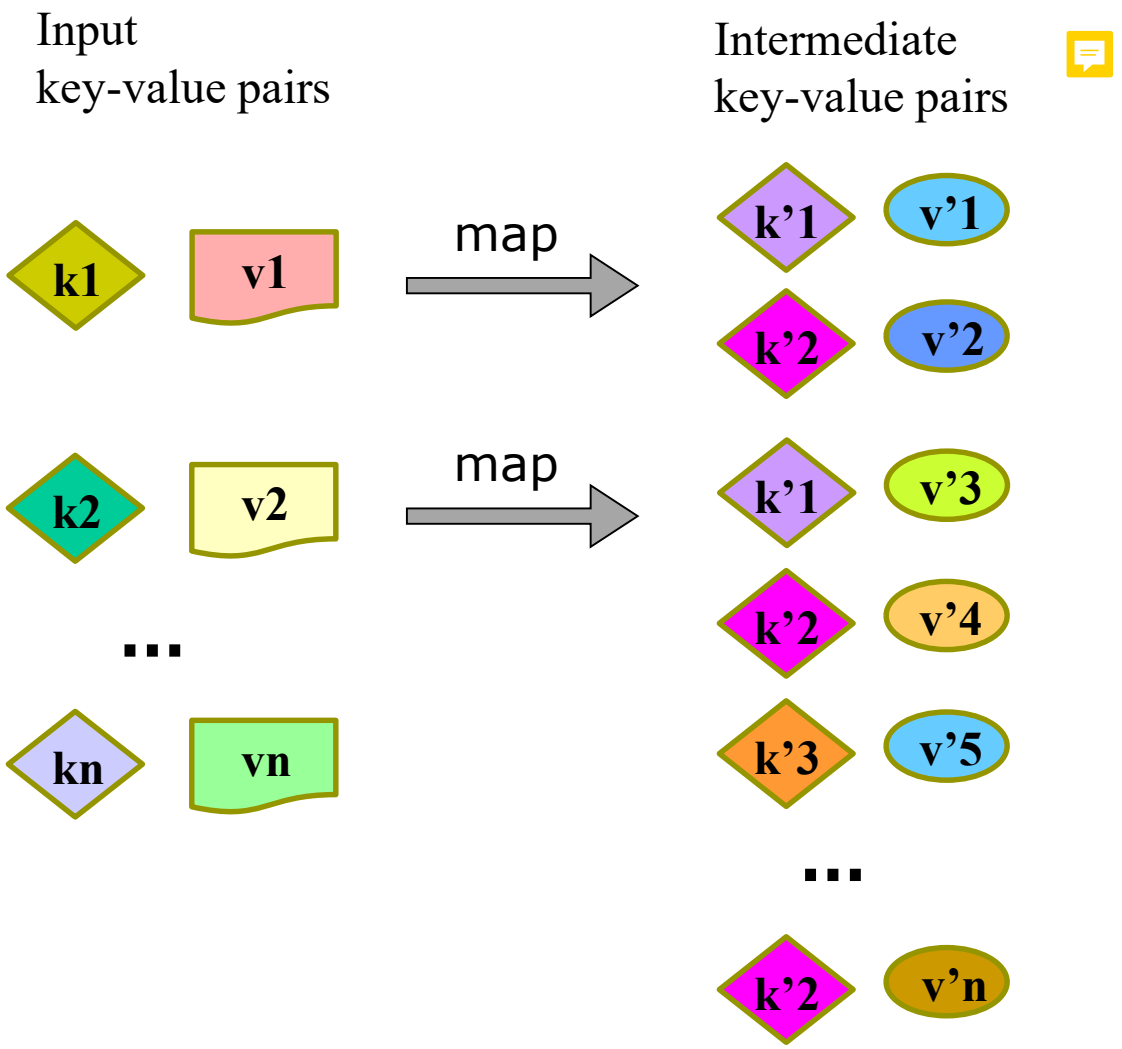


Reduce

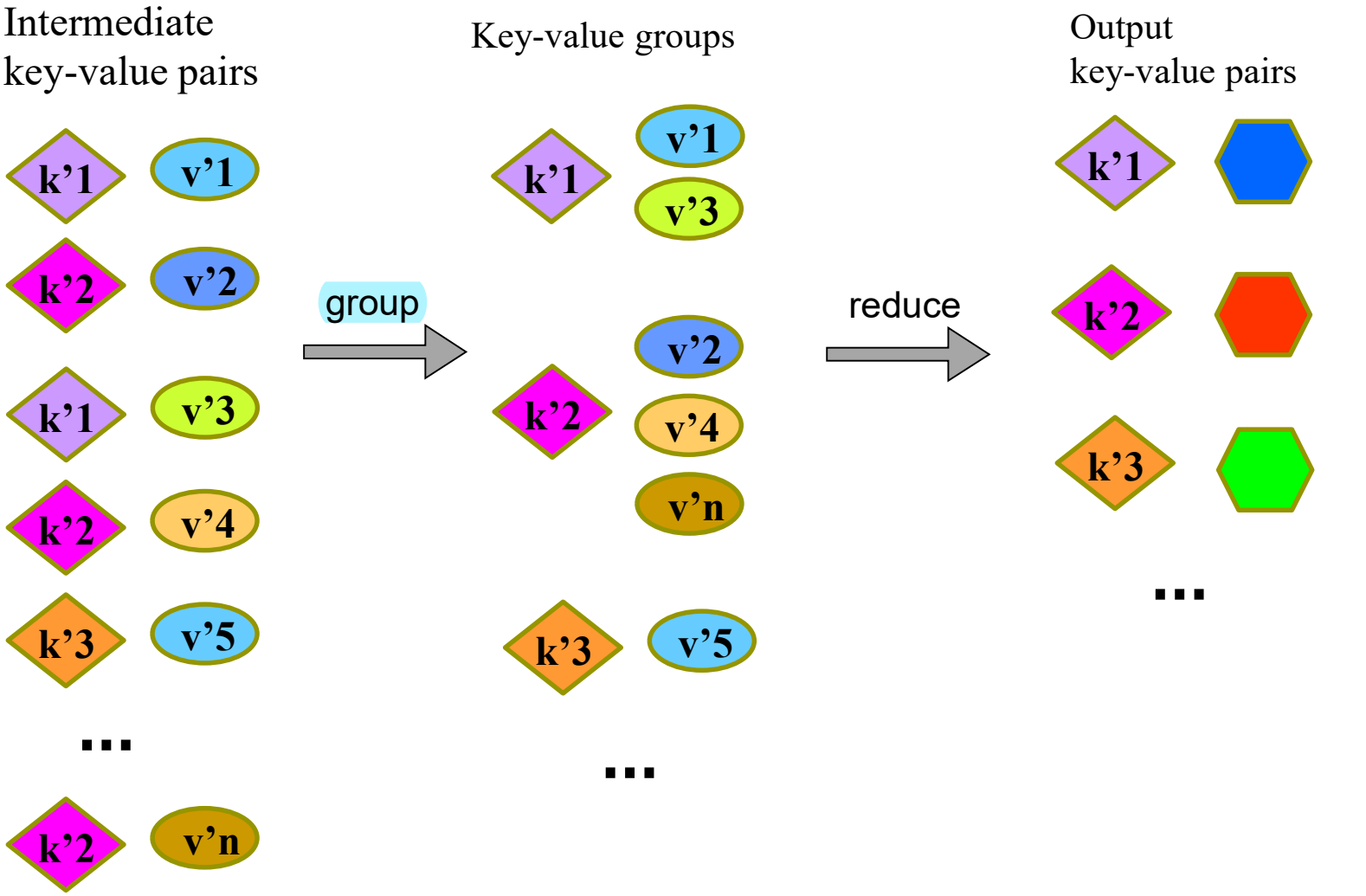
- reduce (out_key, list(intermediate_value)) -> list(out_value)
  - Combine all intermediate values for a particular key
  - Produce a set of merged output values (usually just one)



The Map Step

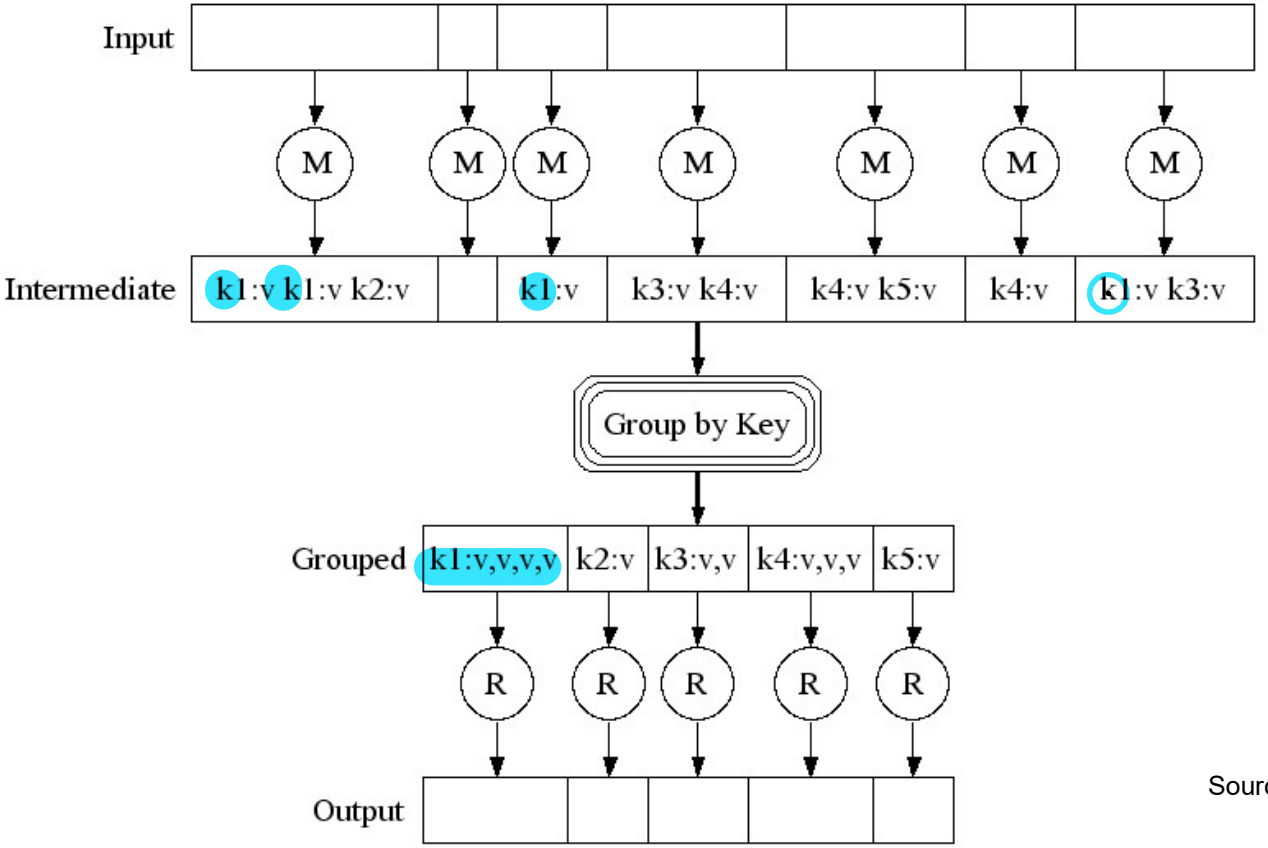

Input
key-value pairs
Intermediate
key-value pairs
k1
v1
map
k2
v2
map
kn
vn
k'1
v'1
k'2
v'2
k'1
v'3
k'2
v'4
k'3
v'5
k'2
v'n
E.g. (doc-id, doc-content)
E.g. (word, wordcount-in-a-doc)

The Reduce Step



E.g.
(word, wordcount-in-a-doc) (word, list-of-wordcount) (word, final-count)

Putting It Together



imagine how to use this to search documents in a corpus!

Source: Google Lab

map() functions run in parallel
reduce() functions may also run in parallel, but they must wait for map()

Example: Word Count (Pseudo Code)

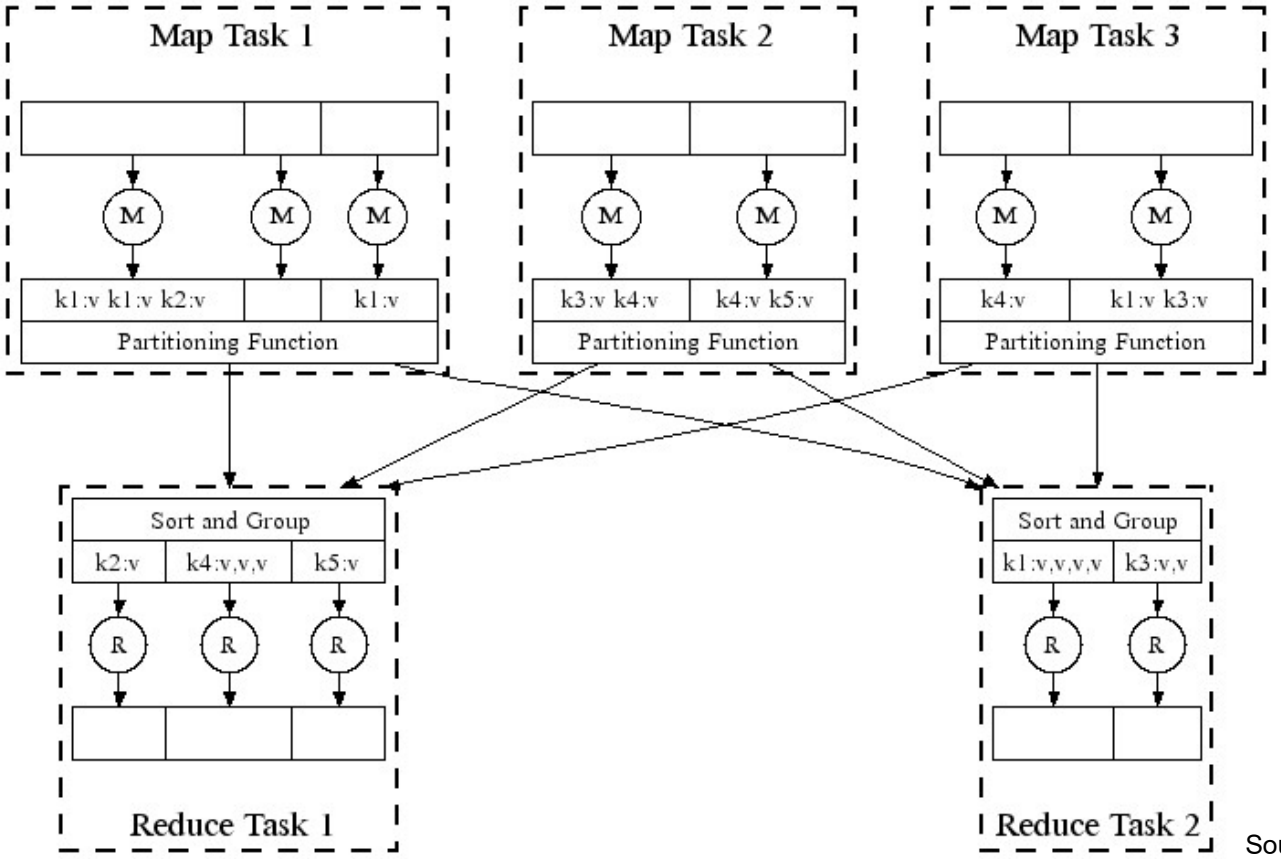
```
map(key, value):
// key: document name; value: text of document
   for each word w in value:
       emit(w, 1)
```

```
reduce(key, values):
// key: a word; value: an iterator over counts
       result = 0
       for each count v in values:
              result += v
       emit(key, result)
```

Examples used in Google

- **indexing process of Google web search service**
  - one phase of the computation dropped from approximately 3800 lines of C++ code to approximately 700 lines when expressed using MapReduce
  - data managed by GFS
- large-scale machine learning problems,
- clustering problems for the Google News and Froogle products,
- extraction of data used to produce reports of popular queries (e.g. Google Zeitgeist),
- extraction of properties of web pages for new experiments and products (e.g. extraction of geographical locations from a large corpus of web pages for localized search)
- large-scale graph computations.

Parallel Execution



Source: Google Lab

MapReduce master takes the location information of the input files into account (GFS supports this) and attempts to schedule a map task on a machine that contains a replica of the corresponding input data.

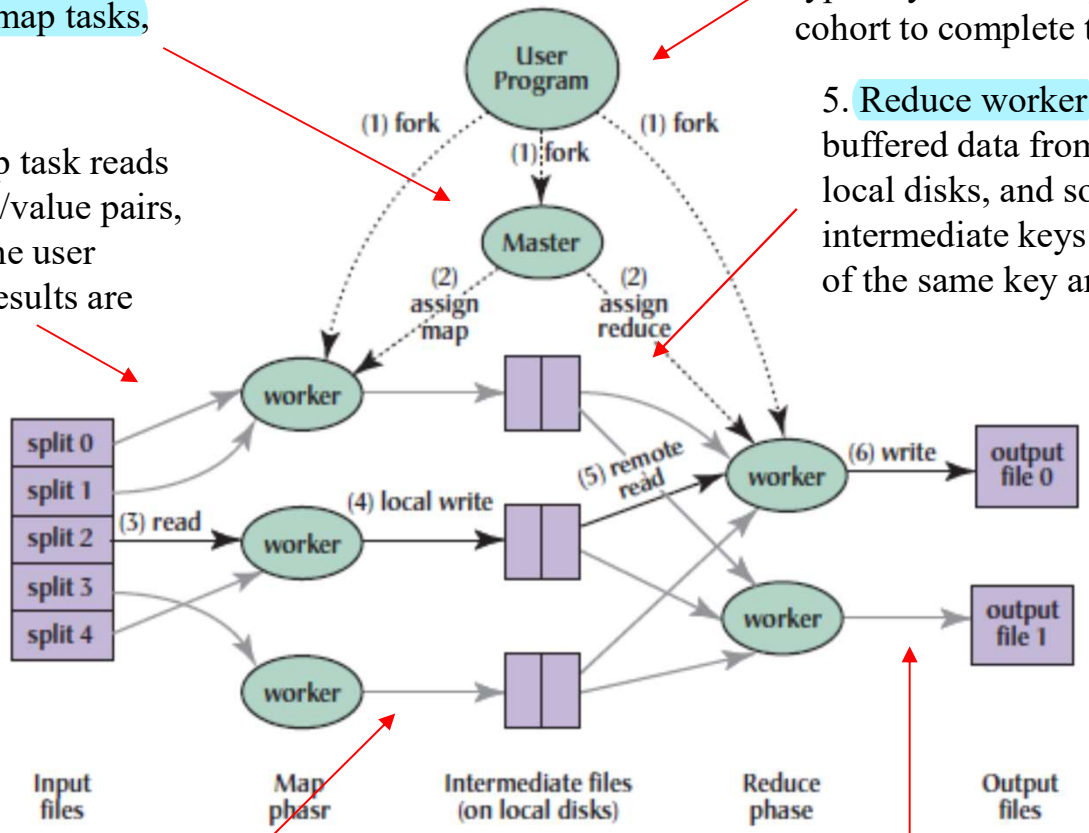
Implementation

2. Master-slave architecture: 1 master *N* workers to execute *M* map tasks, and *R* reduce tasks

1. splits the input files into *M* pieces of typically 16-64MB piece, and initiates a cohort to complete the task

3. Worker assigned a map task reads the input part, parses key/value pairs, and passes each pair to the user defined Map function. Results are buffered in memory.

5. Reduce worker uses RPC to read the buffered data from the map workers' local disks, and sort them by the intermediate keys so that all occurrences of the same key are grouped together.



7. When all the tasks are completed, the master wakes up the user program, and then the MapReduce call in the user program returns back to the user code.

4. The buffered pairs are periodically written to local disk, partitioned into *R* regions, with their locations passed back to the master, who then forwards to the reduce workers.

6. For each unique intermediate key encountered, the reduce worker passes the key and the corresponding set of intermediate values to the user's Reduce function. The output is appended to a final output file for this reduce partition.

Source: MapReduce: Simplified data processing on large clusters, J. Dean & S. Ghemawat, CACM, 2008

How many Map and Reduce tasks?

- *M* map tasks, *R* reduce tasks Rule of thumb:
  - Make *M* and *R* much larger than the number of nodes in cluster
  - One file chunk per map is common
  - Improves dynamic load balancing and speeds recovery from worker failure
  - *R* is specified by user and is usually smaller than *M*, because output is spread across *R* files
  - often perform MapReduce computations with M = 200,000 and R = 5, 000, using 2,000 worker machines (as in OSDI 2004)

Coordination

- Master data structures
  - Task status: (idle, in-progress, completed)
  - Idle tasks get scheduled as workers become available
  - When a map task completes, it sends the master the location and sizes of its R intermediate files, one for each reducer
  - Master pushes this info to reducers
- Master pings workers periodically to detect failures

Fault Tolerance

- Map worker failure
  - Map tasks completed or in-progress at worker are reset to idle
  - Reduce workers are notified when task is rescheduled on another worker
- Reduce worker failure
  - Only in-progress tasks are reset to idle
- Master failure
  - MapReduce task is aborted and client is notified (implementation in OSDI 2004)

Combiners

- Often a map task will produce many pairs of the form (k, $v1$), (k, $v2$), … for the same key k
  - E.g., popular words in Word Count
- Can save network time by pre-aggregating at mapper
  - combine($k1$, *list*($v1$)) → $v2$
  - Usually same as reduce function
- Works only if reduce function is commutative and associative

Partition Function

- Inputs to map tasks are created by contiguous splits of input file
- For reduce, we need to ensure that records with the same intermediate key end up at the same worker
- System uses a default partition function e.g., hash(key) mod R
- Sometimes useful to override
  - E.g., hash(hostname(URL)) mod R ensures URLs from a host end up in the same output file

Online Tutorial

Apache Hadoop 3.3.2 – MapReduce Tutorial