General Purpose Computing Systems I: MapReduce and Hadoop

Shiow-yang Wu (吳秀陽)

CSIE, NDHU, Taiwan, ROC

Lecture material is mostly home-grown, partly taken with permission and courtesy from Professor Shih-Wei Liao of NTU.

Outline

- What is MapReduce? What is it used for? Why MapReduce?
- MapReduce concepts, models and examples
- Hadoop cluster for MapReduce
- Execution details and internals
- Problems with MapReduce

Outline (cont.)

- Problem solving and algorithm design with MapReduce
- Some MapReduce algorithms
- Data mining with MapReduce
- MapReduce and Hadoop (Assignment)

What is MapReduce?

- Data-parallel programming model for clusters of commodity machines
  - Designed for scalability and fault-tolerance
- Pioneered by Google
  - Processes 20 PB of data per day
- Popularized by open-source Hadoop project
  - Used by Yahoo!, Facebook, Amazon, …

What is MapReduce Used for?

- At Google:
  - Index building for Google Search
  - Article clustering for Google News
  - Statistical machine translation
- At Yahoo!:
  - Index building for Yahoo! Search
  - Spam detection for Yahoo! Mail
- At Facebook:
  - Data mining
  - Ad optimization
  - Spam detection

What is MapReduce Used for?

- In research:
  - Analyzing Wikipedia conflicts (PARC)
  - Natural language processing (CMU)
  - Bioinformatics (Maryland)
  - Particle physics (Nebraska)
  - Ocean climate simulation (Washington)
  - <Your application here>

MapReduce History

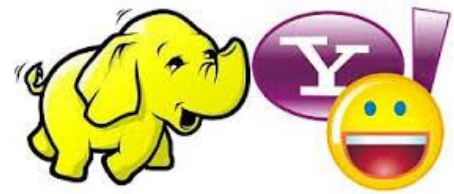
- The foundation stone: The Google File System by Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung in 2003. (*19th ACM Symposium on Operating Systems Principles*)
- The paper that started everything – MapReduce: Simplified Data Processing on Large Clusters by Jeffrey Dean and Sanjay Ghemawat in 2004. (*6th Symposium on Operating System Design and Implementation*)

MapReduce History

- Shortly after the MapReduce paper, open source pioneers Doug Cutting and Mike Cafarella started working on a MapReduce implementation to solve the scalability problem of Nutch (an open source search engine)
- Over the course of a few months, Cutting and Cafarella built up the underlying file systems and processing framework that would become Hadoop (in Java)

MapReduce History

- In 2006, Cutting went to work with Yahoo.
- They spun out the storage and processing parts of Nutch to form Hadoop (named after Cutting's son's stuffed elephant).
- Over time and heavy investment by Yahoo!, Hadoop eventually became a top-level Apache Foundation project.



MapReduce Today

- Today, numerous independent people and organizations contribute to Hadoop.
- Every new release adds functionality and boosts performance.
- Several other open source projects have been built with Hadoop at their core, and this list is continually growing.
- Some of the more popular ones: Pig(programming tool), Hive(warehousing), HBase(NoSQL DB), Mahout(machine learning), and ZooKeeper(distributed systems and services).

Why MapReduce?

- Problem: Lots of data!
- Example: Word frequencies in Web pages
- This is how the world's first search engine was done (Archie)
- World's first search engine vs.
  A search engine from Stanford called Google

Word Frequencies in Pages

- 130 trillion web pages x 1KB/page = 130PB
- One computer can read 750 MB/sec from disk
  - 5+ years to read the web
  - 13K hard drives(10TB HDD) to store the web
- Even more: To do something with the data
  - Compute the word frequencies for each word in each website

Basic Solution: Spread the work over many machines

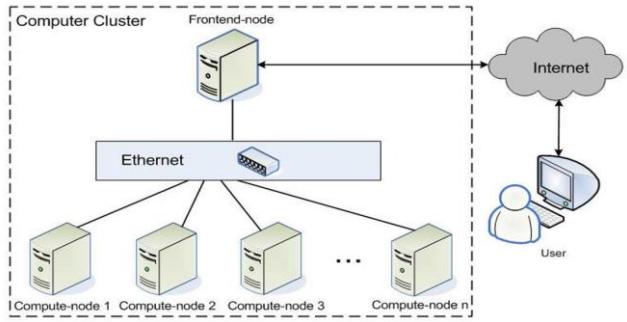
- Same problem with 10,000 machines: 4+ hours
- New problems: Extra programming works
  - communication and coordination
  - recovering from machine failure
  - status reporting
  - debugging
  - optimization
  - locality
- Those works repeat for every problem you want to solve

MapReduce Design Goals

1. **Scalability to large data volumes:**
   - Scan 100 TB on 1 node @ 50 MB/s = 24 days
   - Scan on 1000-node cluster = 35 minutes
   - => 1000's of machines, 10,000's of disks
2. **Cost-efficiency:**
   - Commodity machines (cheap, but unreliable)
   - Commodity network
   - Automatic fault-tolerance (fewer administrators)
   - Easy to use (fewer programmers)

Computing Clusters

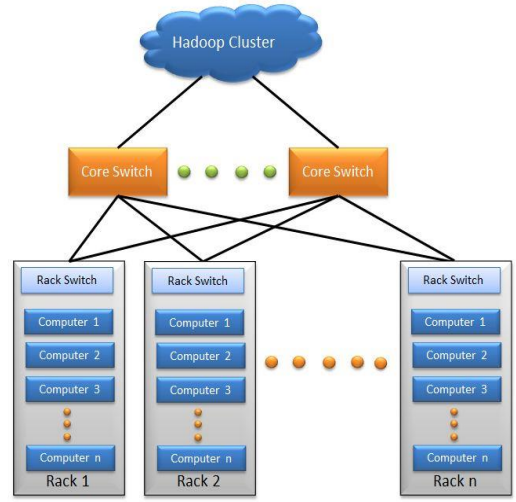
- Many racks of computers, thousands of machines per cluster
- Limited bisection bandwidth between racks



Hadoop Cluster

Typical Hadoop Cluster

- 40 nodes/rack, 1000-4000 nodes in cluster
- 1 Gbps within rack, 10 Gbps across racks
- Node specs: 2 quad core 2-2.5GHz CPUs, 12-24GB RAM, 4-6 disks(4TB)



Implications of Computing Environment

- Single-thread performance doesn't matter
  - **Large problems** and **total throughput/$** are more important than peak performance
- Stuff Breaks
  - More nodes imply higher probability of breaking down
- "Ultra-reliable" hardware doesn't really help
  - At large scales, super-fancy reliable hardware still fails, albeit less often
    - software still needs to be fault-tolerant
    - commodity machines without fancy hardware give better perf/$

Challenges

1. **Cheap nodes fail, especially if you have many**
   - Mean time between failures for 1 node = 3 years
   - Mean time between failures for 1000 nodes = 1 day
   - Solution: Build fault-tolerance into system
2. **Commodity network = low bandwidth**
   - Solution: Push computation to the data
3. **Programming distributed systems is hard**
   - Solution: Data-parallel programming model: users write "map" & "reduce" functions, system distributes work and handles faults

MapReduce

- A simple **programming model** that applies to many large-scale computing problems
- Hide messy details of distributed programs behind MapReduce runtime library:
  - Automatic parallelization
  - Load balancing
  - Network and disk transfer optimization
  - Handling of machine failures
  - Robustness
  - Improvements to core library

MapReduce Basics

- Semantics borrowed from ***function programming languages***
- FP language are usually **stateless,** which is very good for parallelism
  - No need to worry about synchronization
- Even not using MapReduce, many programming languages like Ruby and Python provides such semantics.
  - Simplicity
  - Chance for implicit optimization

Functional Abstractions Hide Parallelism

- The ideas of functions, mapping and reducing are from functional programming languages (eg. Lisp)
- Map()
  - In FP: [1,2,3,4] - (*2) -> [2,4,6,8]
  - Process a key/value pair to generate intermediate key/value pairs
- Reduce()
  - In FP: [1,2,3,4] - (sum) -> 10
  - Merge all intermediate values associated with the same key
- Both Map and Reduce are easy to parallelize

MapReduce in Functions

- Data type: key-value *records*
- Map function:

$$(K_{in}, V_{in}) \rightarrow list(K_{inter}, V_{inter})$$

- Reduce function:

$$(K_{inter}, list(V_{inter})) \rightarrow list(K_{out}, V_{out})$$

Programming Model

1. Read a lot of data
2. **Map**: extract something you care about from each record
3. Shuffle and Sort
4. **Reduce**: aggregate, summarize, filter, or transform
5. Write the results

Outline stays the same,
map and reduce change to fit the problem

Programming Model: More Specifically

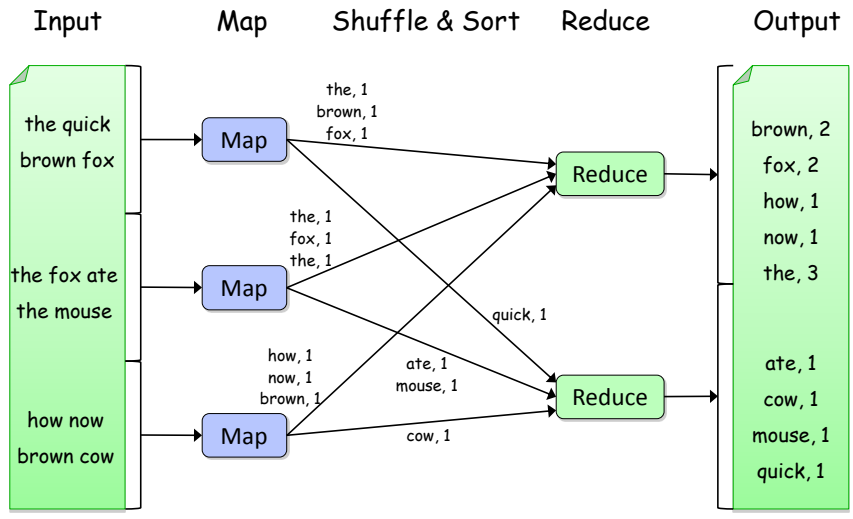
- Programmer specifies two primary methods:
  - `map(k, v) -> <k', v'>*`
  - `reduce(k', <v'>*) -> <k'', v''>*`
- All v' with same k' are reduced together ***in order***
- Can also specify:
  - partition(k', total partitions) -> partition for k'
    - often a simple hash of the key
    - allows reduce operations for different k' to be parallelized

The Word Count Example

```
def mapper(line):
    foreach word in line.split():
        output(word, 1)

def reducer(key, values):
    output(key, sum(values))
```

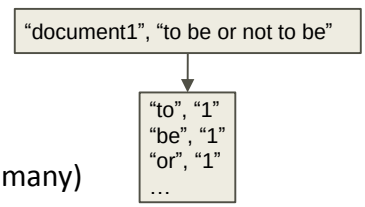
Word Count Execution

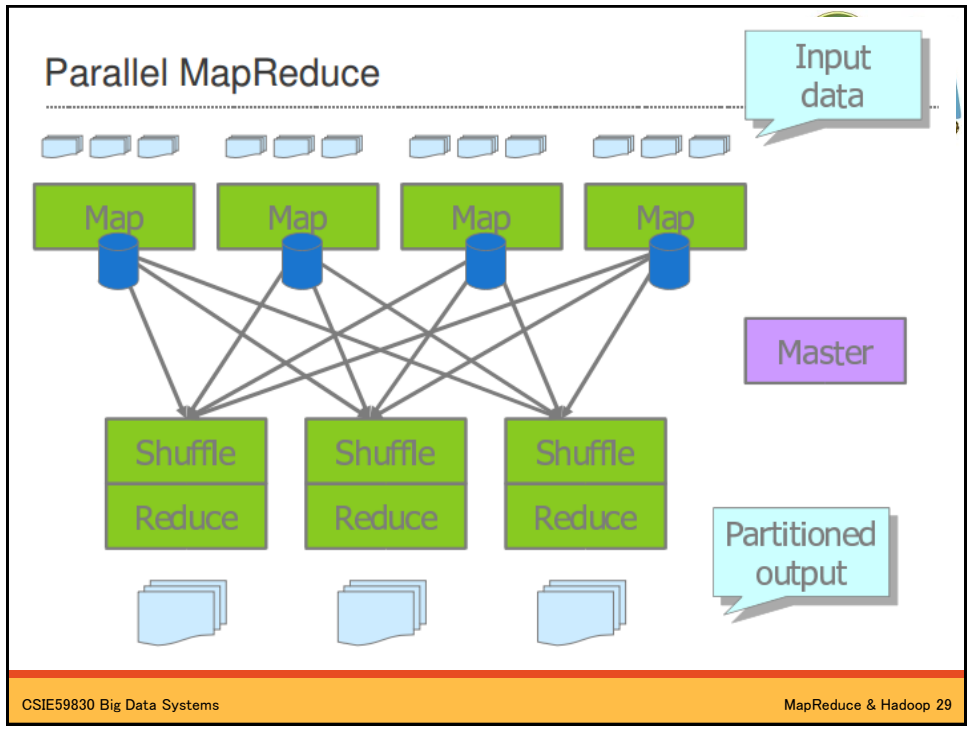


Web Pages Example

Example: Word Frequencies in Web Pages

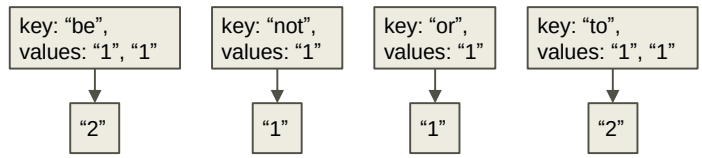
- Input is files with one document per record
- Specify a map function that takes a key/value pair
  - key: document URL
  - value: document contents
- Output of map function is (potentially many) key/value pairs.
- In our case, output (word, "1") once per word in the document





Web Pages Example (cont.)

- MapReduce library gathers together all pairs with the same key (shuffle/sort)
- The reduce function combines the values for a key. In our case, compute the sum



- Output of reduce paired with key and saved

"be", "2"
"not", "1"
"or", "1"
"to", "2"

CSIE59830 Big Data Systems MapReduce & Hadoop 30

Pseudo-Code

```
map(String input_key, String input_value):
// input_key: document name
// input_value: document contents
   for each word w in input_value:
   emitIntermediate(w, "1");

reduce(String key, Iterator intermediate_values):
// key: a word, same for input and output
// intermediate_values: a list of counts
   int result = 0;
   for each v in intermediate_values:
     result += ParseInt(v);
   emit(asString(result));
```


How MapReduce Works

- User to do list:
  - indicate:
    - Input/output files
    - **M**: number of map tasks
    - **R**: number of reduce tasks
    - **W**: number of machines
  - Write *map* and *reduce* functions
  - Submit the job
- This requires no knowledge of parallel/distributed systems!!!
- What about everything else?

Data Distribution

- Input files are split into **M** pieces on distributed file system
  - Typically ~ 64 MB blocks
- Intermediate files created from *map* tasks are written to local disk
- Output files are written to distributed file system

Assigning Tasks

- Many copies of user program are started
- Tries to utilize data localization by running *map* tasks on machines with data
- One instance becomes the Master
- Master finds idle machines and assigns them tasks

Execution (map)

- *Map* workers read in contents of corresponding input partition
- Perform user-defined *map* computation to create intermediate <key,value> pairs
- Periodically buffered output pairs written to local disk
  - Partitioned into **R** regions by a partitioning function

Partition Function

- Example partition function: hash(key) mod **R**
- Why do we need this?
- Example Scenario:
  - Want to do word counting on 10 documents
  - 5 *map* tasks, 2 *reduce* tasks

Execution (reduce)

- Reduce workers iterate over ordered intermediate data
  - Each unique key encountered – values are passed to user's reduce function
  - eg. <key, [value1, value2,..., valueN]>
- Output of user's *reduce* function is written to output file on distributed file system
- When all tasks have completed, master wakes up user program

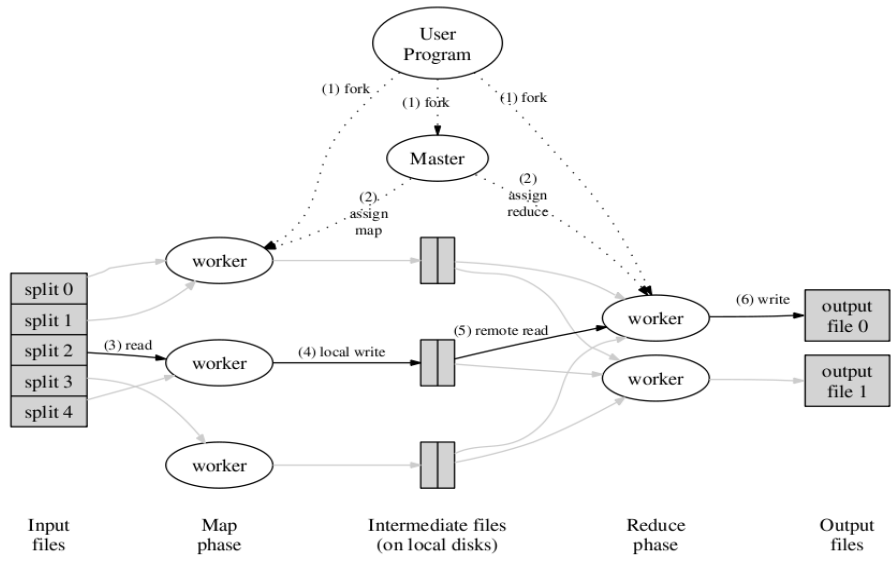
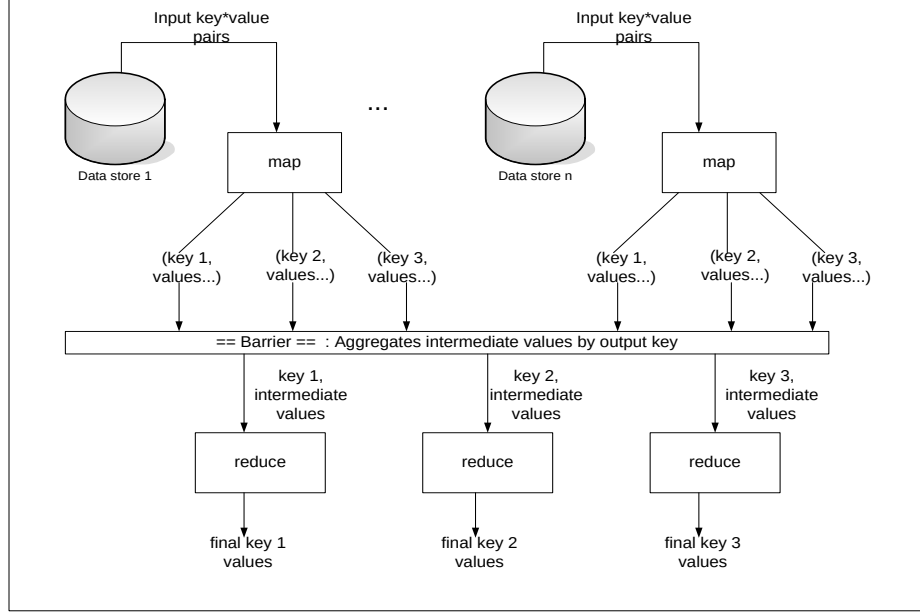


Figure 1: Execution overview

Observations

- No *reduce* can begin until *map* is complete
- Tasks scheduled based on location of data
- If *map* worker fails any time before *reduce* finishes, task must be completely rerun
- Master must communicate locations of intermediate files
- MapReduce library does most of the hard work for us!



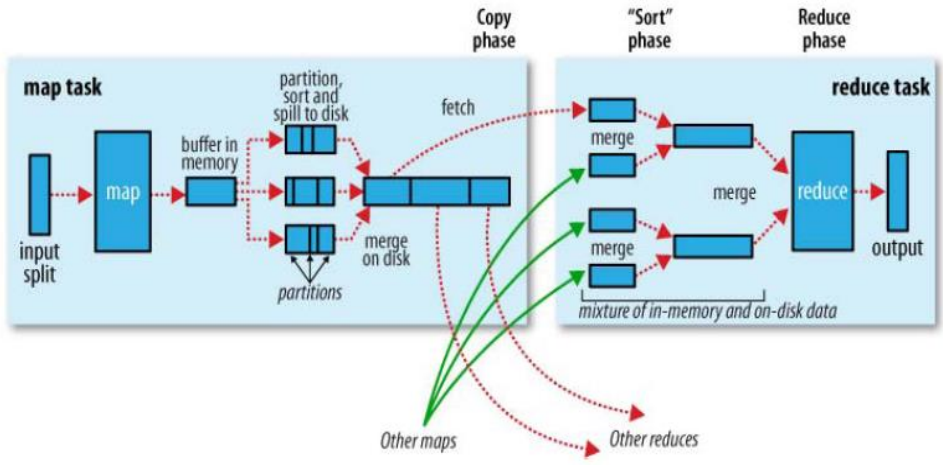
MapReduce Execution Details

- Single *master* controls job execution on multiple *slaves*
- Mappers preferentially placed on same node or same rack as their input block
  - Minimizes network usage
- Mappers save outputs to local disk before serving them to reducers
  - Allows recovery if a reducer crashes
  - Allows having more reducers than nodes

Shuffle & Sort

- The process between map and reduce
- Intermidiate output <k' , v'> are partitioned according to a ***partition function***
  - Usually a simple hash function for load balance
  - Specify your own if special purpose needed
- Exchange intermidiate output if needed
- Guarantees key order **within a reducer**

Shuffle & Sort



Shuffle & Sort: more details

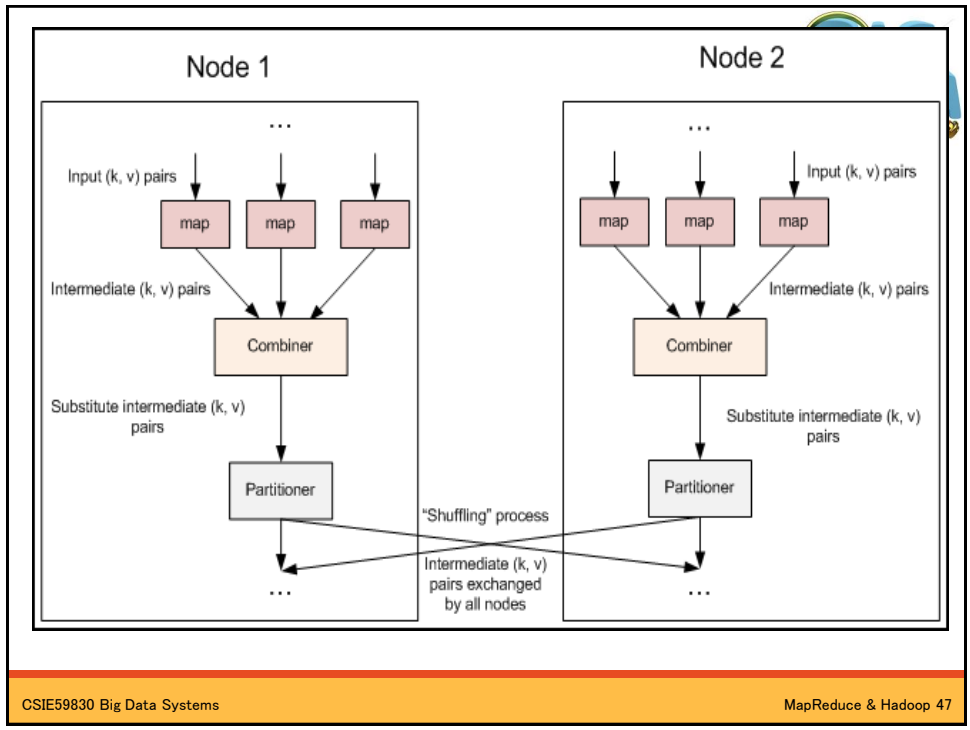
- Within a mapper:
  a. Keep emitting (k' , v') pairs to buffer until the **spill rate** of the buffer exceeds. After exceeding, the part of buffer is locked.
  b. An independent thread sorts the data within the locked buffer and **spill it out** to disk as a temporary document
  c. During the sorting, the mapper is only allowed to write to the remaining part of the buffer.
  d. After the map phase is done, combine the temporary documents into 1 document – the output of the mapper

Back to word count

- Consider “aaa aaa aaa aaa aaa bbb ccc…”
- Lots of (aaa, 1) are emitted by the mapper
- Extra overhead
  - Disk spill out
  - Network

The Combiner

- A pass executed between map and reduce
- A “mini-reduce” process that takes data from one machine only
  - But probably different from your reduce function
- To compress / trim the output from the map
- Optional: depends on your application
  - O: word count, min/max…
  - X: median

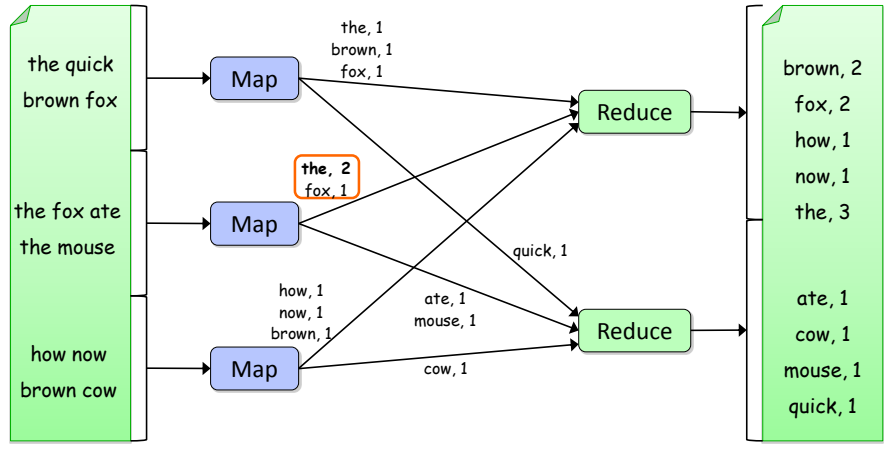


Combiner Example

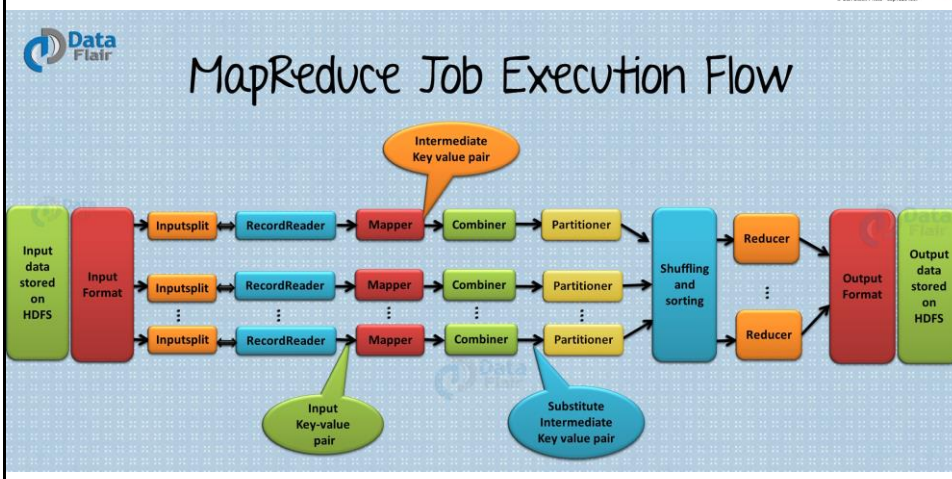
- A combiner is a local aggregation function for repeated keys produced by same map
- Works for associative functions like sum, count, max
- Decreases size of intermediate data
- Example: map-side aggregation for Word Count:

```
def combiner(key, values):
    output(key, sum(values))
```

Word Count with Combiner



MapReduce Job Execution Flow



Advanced Issues: Scheduling

- One master, many workers
  - Input data split into M map tasks (typically 64 MB in size)
  - Reduce phase partitioned into R reduce tasks
  - Tasks are assigned to workers dynamically
  - Often: M=200,000; R=5,000; workers=2,000
- Master assigns each map task to a free worker
  - Considers locality of data to worker when assigning task
  - Worker reads task input (often from local disk!)
  - Worker produces R local files containing intermediate k/v pairs

Advanced Issues: Scheduling (cont.)

- Master assigns each reduce task to a free worker
  - Worker reads intermediate k/v pairs from map workers
  - Worker sorts & applies user's Reduce op to produce the output
- Fine granularity tasks: many more map tasks than machines
  - Minimizes time for **fault recovery**
  - Possible to have **pipelined shuffling** with map execution
  - Better **dynamic load balancing**
  - Why not as many map task as possbile?

Advanced Issues: Fault Tolerance

On worker failure:

- Detect failure via periodic **heartbeats**
- Re-execute completed and in-progress map tasks
- Re-execute in progress reduce tasks
- Task completion committed through master

On master failure:

- State is checkpointed to GFS: new master recovers & continues

Refinement: Backup Tasks

- Problem: Slow workers significantly lengthen completion time
  - Resouce contentions with other jobs
  - Bad disks and soft errors
  - Processor cache disabled
- Solution: Near end of phase, spawn backup copies of tasks
- Stragglers(流浪者) problem: a small number of mappers or reducers takes significantly longer than the others to complete

Refinement: Locality Optimazation

- Replicate input file blocks
- Split tasks into the size of a GFS block
- Map tasks scheduled to the same machine or same rack with the blocks of input data

  => Each job can be done on the same machine
- Effect: Thousands of machines read input at local disk speed
  - Without this, rack switches limit read rate

Refinement: Skipping Bad Records

- Problem: Functions sometimes fail for particular inputs
- Solution: Skip them!
  - On seg fault, send UDP packet to inform master about which input caused the fault.
  - If master sees K failures for same record, skip the record afterwards

Implications for Multi-core Processors

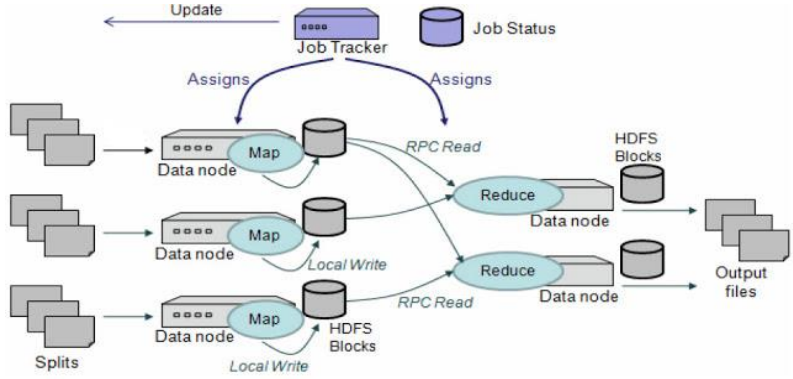
- Multi-core processors require parallelism
  - But many programmers are uncomfortable writing parallel programs
- MapReduce provides an easy-to-understand programming model for a very diverse set of computing problems
  - users don't need to be parallel programming experts
- Optimizations useful even in single machine, multi-core environment

Problems with MapReduce

- It's hard & low-level for developers to write
  - Most developers are familiar with SQL
  - Solution: Apache Hive
- Expensive cost for fault recovery
  - Re-execute whole MR programs
  - Solution: Apache Spark's ***lineage***
- Requires intensive disk I/O
  - Intermidiate data is always written to local disk
  - Solution: Apache Spark's in-memory computing

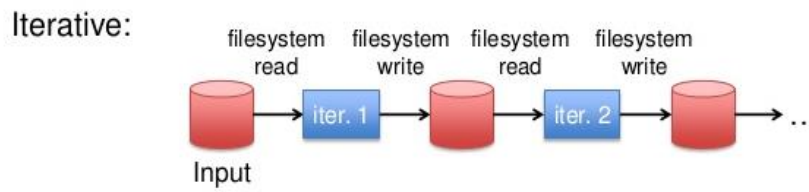
Problems with MapReduce

- MapReduce Framework



Problems with MapReduce

- When doing iterative computation
  - Bad performance due to replication and disk I/O



Problems with MapReduce

- Apache Spark's in-memory computing
  - 10-100X faster than disk

Iterative:

Problems with MapReduce

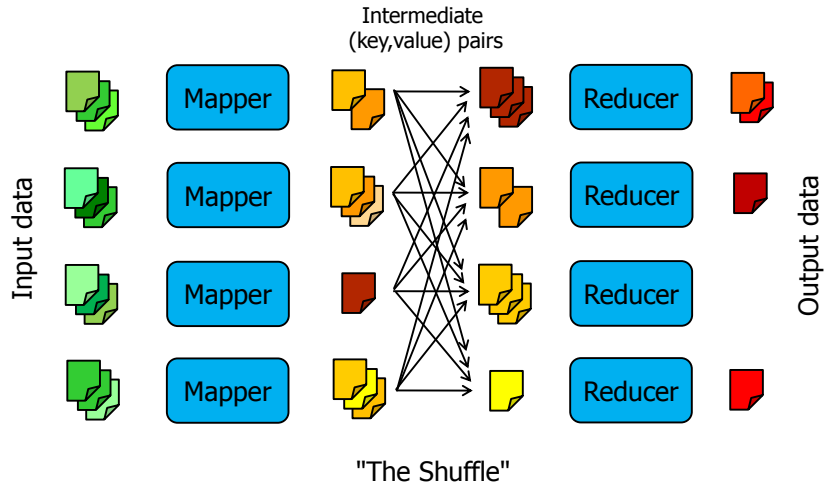
- Spawning each Mapper/Reducer takes time
  - Solution: Worker Pool in Google Tenzing(SQL query engine on Hadoop), it contains running processes as Mapper/Reducer
- Not very good for iterative graph computing
  - Solution: Google Pregel for large scale graph processing
- Not very good for interactive ad hoc queries
  - Solution: Google Dremel and BigQuery

Problem Solving with MapReduce

- How to design MapReduce algorithms for the following tasks
  - **Search**: Output lines matching certain patterns
  - **Sort**: Sorting numbers, words, …
  - **Inverted index**: build index from words to documents
  - **Data mining** algorithms (sequential pattern mining)
  - **BFS** on graph*
  - **PageRank***

MapReduce Algorithm Design

Recap: MapReduce Dataflow



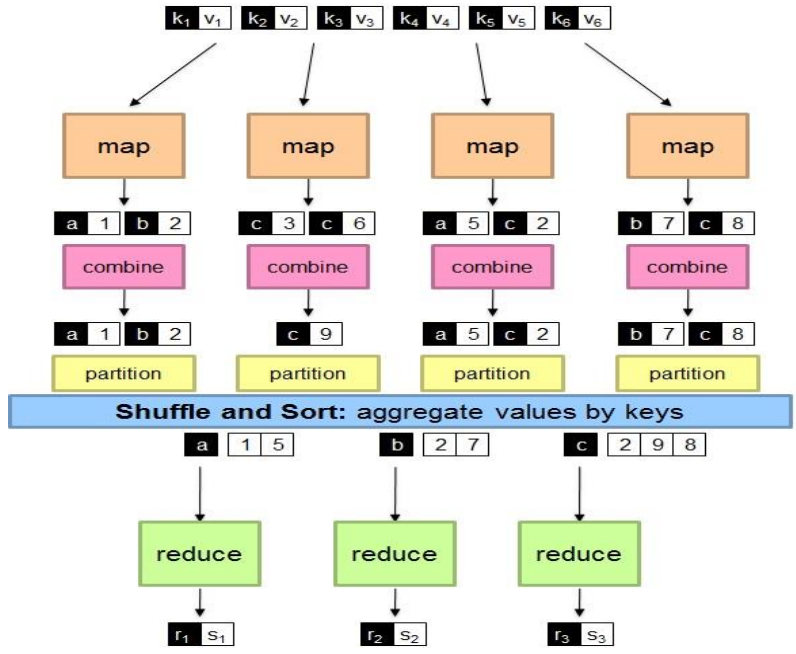
Recap: MapReduce

- Programmers must specify:
  map (k, v) → list(<k', v'>)
  reduce (k', list(v')) → <k'', v''>
  - All values with the same key are reduced together
- Optionally, also:
  partition (k', number of partitions) → partition for k'
  - Often a simple hash of the key, e.g., hash(k') mod n
  - Divides up key space for parallel reduce operations

  combine (k', v') → <k', v'>*
  - Mini-reducers that run in memory after the map phase
  - Used as an optimization to reduce network traffic
- The execution framework handles everything else…

"Everything Else"

- The execution framework handles everything else…
  - Scheduling: assigns workers to map and reduce tasks
  - "Data distribution": moves processes to data
  - Synchronization: gathers, sorts, and shuffles intermediate data
  - Errors and faults: detects worker failures and restarts
- Limited control over data and execution flow
  - All algorithms must expressed in m, r, c, p
- You don't know:
  - Where mappers and reducers run
  - When a mapper or reducer begins or finishes
  - Which input a particular mapper is processing
  - Which intermediate key a particular reducer is processing



Recap: Word Count

These types depend on the input data

```
map(key:URL, value:Document)
{
  String[] words = value.split(" ");
  foreach w in words
    emit(w, 1);
}

reduce(rkey:String, rvalues:Integer[])
{
  Integer result = 0;
  foreach v in rvalues
    result = result + v;
  emit(rkey, result);
}
```

Produces intermediate key-value pairs that are sent to the reducer

reduce gets all the intermediate values with the same rkey

These types can be (and often are) different from the ones in map()

Both map() and reduce() are stateless: Can't have a global variable that is preserved across invocations!

Any key-value pairs emitted by the reducer are added to the final output

MapReduce Jobs

- Tend to be very short, code-wise
  - IdentityReducer is very common
- "Utility" jobs can be composed
- Represent a *data flow*, more so than a procedure

MR Algorithm Design Principles

- Think data, not flow!
- Decompose the problem into modules connected by data flow, not algorithmic flow.
- Design MR job(s) for each module.
- Link jobs with (key, value) pairs.
- Key and value can potentially be anything !!

Sorting

Inputs:

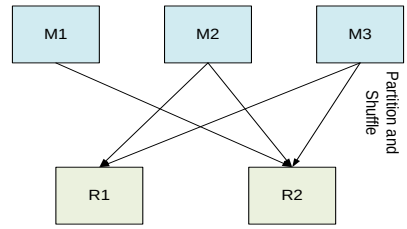
- A file of values to sort, one value per line.
- Mapper key is file ID, line number
- Mapper value is the contents of the line
- This can be easily generalized into sorting multiple files. (Exercise)

Sort Algorithm

- Takes advantage of reducer properties: (key, value) pairs are processed in order by key; reducers are themselves ordered
- Mapper: Identity function for value
  (k, v) → (v, _)
- Reducer: Identity function (k', _) -> (k', "")

Sort: The Trick

- (key, value) pairs from mappers are sent to a particular reducer based on hash(key)
- Must pick the hash function for your data such that $k_1 < k_2 \Rightarrow hash(k_1) < hash(k_2)$



Searching

- Given a set of files containing lines of text and a search pattern to find.
- Determine the files that matches the pattern.
- Can be easily generalized into determining (file, [l1, l2, …]), i.e. all lines that match the pattern. (Exercise)
- Search pattern sent as special parameter

Search Algorithm

- Mapper:
  - Given (fileID, some text) and “pattern”, if “text” matches “pattern” output (filename, _)
- Reducer:
  - Identity function

Search: An Optimization

- Once a file is found to be interesting, we only need to mark it that way once
- Use *Combiner* function to fold redundant (filename, _) pairs into a single one
  - Reduces network I/O

Inverted Index

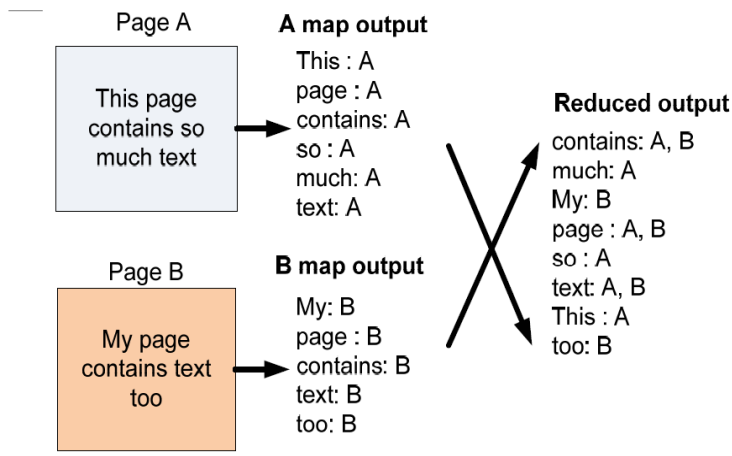
- Given a set of document(text) files.
- For each word, determine the docs in which the word appears. (Boolean)
- For each word, determine the docs and positions where the word appears. (Exercise)

Inverted Index Algorithm

- Mapper key is file ID
- Mapper value is the contents of the file.

- Mapper: For each word in (fileID, words), map to (word, fileID)
- Reducer: For each word, output (word, [f1, f2, …])

Indexing: Data Flow



TF-IDF

- Term Frequency – Inverse Document Frequency
  - Relevant to text processing
  - Common web analysis algorithm

The Algorithm, Formally

$$\mathrm{tf_i} = \frac{n_i}{\sum_k n_k}$$

$$\mathrm{idf_i} = \log \frac{|D|}{|\{d : t_i \in d\}|}$$

$$\mathrm{tfidf} = \mathrm{tf} \cdot \mathrm{idf}$$

- $|D|$: total number of documents in the corpus
- $|\{d : t_i \in d\}|$: number of documents where the term t_i appears (that is $n_i \neq 0$).

Information We Need

- Number of times term X appears in a given document
- Number of terms in each document
- Number of documents X appears in
- Total number of documents

Job 1: Word Frequency in Doc

- Mapper
  - Input: (docname, contents)
  - Output: ((word, docname), 1)
- Reducer
  - Sums counts for word in document
  - Outputs ((word, docname), *n*)
- Combiner is the same as Reducer

Job 2: Word Counts For Docs

- Mapper
  - Input: ((word, docname), *n*)
  - Output: (docname, (word, *n)*)
- Reducer
  - Sums frequency of individual *n*'s in same doc
  - Feeds original data through
  - Outputs ((word, docname), (*n*, *N)*)

Job 3: Word Frequency In Corpus

- Mapper
  - Input: ((word, docname), (*n, N)*)
  - Output: (word, (docname, *n*, *N*, 1))
- Reducer
  - Sums counts for word in corpus
  - Outputs ((word, docname), (*n*, *N*, *m*))

Job 4: Calculate TF-IDF

- Mapper
  - Input: ((word, docname), (*n*, *N*, *m*))
  - Assume D is known (or, easy MR to find it, exercise!)
  - Output ((word, docname), TF*IDF)
- Reducer
  - Just the identity function

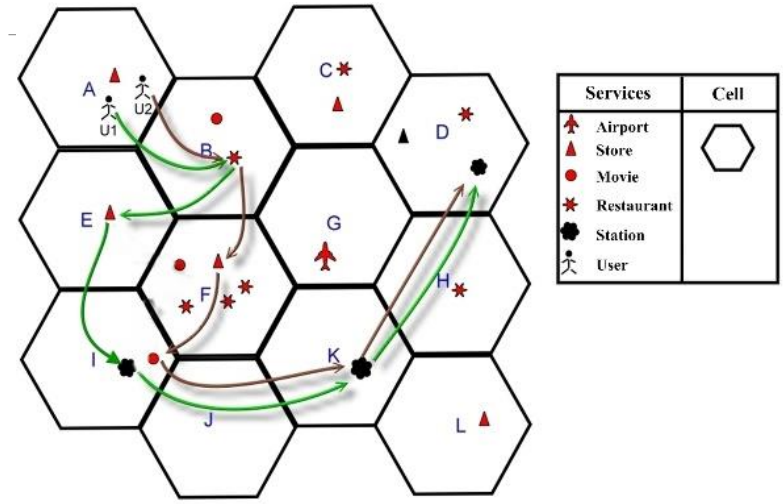
Working At Scale

- Buffering (doc, *n*, *N*) counts while summing 1's into *m* may not fit in memory
  - How many documents does the word "the" occur in?
- Possible solutions
  - Ignore very-high-frequency words (AKA stop words)
  - Write out intermediate data to a file
  - Use another MR pass

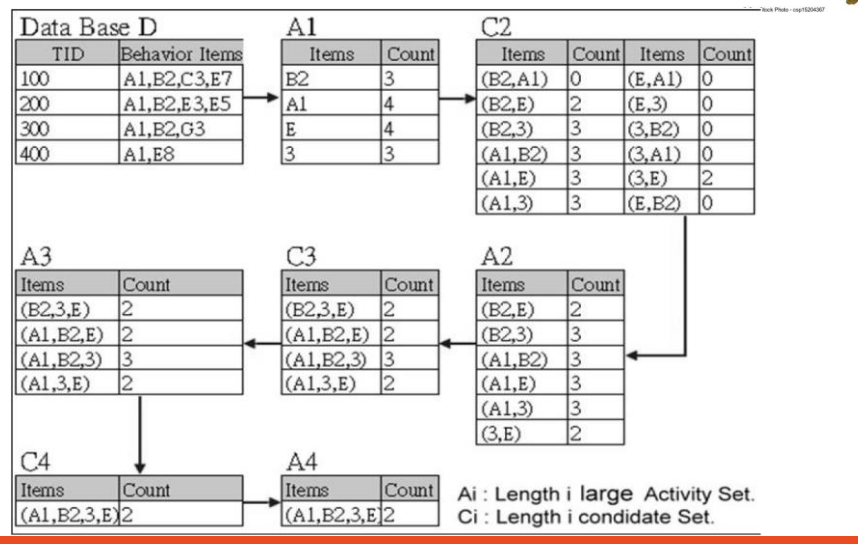
Final Thoughts on TF-IDF

- Several small jobs add up to full algorithm
- Lots of code reuse possible
  - Stock classes exist for aggregation, identity
- Jobs 3 and 4 can really be done at once in same reducer, saving a write/read cycle
- Very easy to handle medium-large scale, but must take care to ensure flat memory usage for largest scale

Sequential Activity Mining in Mobile Environments



Sequential Activity Mining



Data Base D

TID	Behavior Items
100	A1,B2,C3,E7
200	A1,B2,E3,E5
300	A1,B2,G3
400	A1,E8

A1

Items	Count
B2	3
A1	4
E	4
3	3

C2

Items	Count	Items	Count
(B2,A1)	0	(E,A1)	0
(B2,E)	2	(E,3)	0
(B2,3)	3	(3,B2)	0
(A1,B2)	3	(3,A1)	0
(A1,E)	3	(3,E)	2
(A1,3)	3	(E,B2)	0

A2

Items	Count
(B2,E)	2
(B2,3)	3
(A1,B2)	3
(A1,E)	3
(A1,3)	3
(3,E)	2

C3

Items	Count
(B2,3,E)	2
(A1,B2,E)	2
(A1,B2,3)	3
(A1,3,E)	2

A3

Items	Count
(B2,3,E)	2
(A1,B2,E)	2
(A1,B2,3)	3
(A1,3,E)	2

C4

Items	Count
(A1,B2,3,E)	2

A4

Items	Count
(A1,B2,3,E)	2

Ai : Length i large Activity Set.
Ci : Length i condidate Set.

Activity Mining V1

- Direct conversion from the original algorithm.
- Job1: Large-1 activity set generation.
  - Mapper: given (TID, Behavior_Items), generate all possibnle (item, 1).
  - Reducer: for each item, sum the count and emit all ((item), n) for items with n >= support.

Activity Mining V1

- Job2: Large-2 ~ n activity set generation.
- Mapper: Given a (activityset, count) pair (A, n), emit (prefix of A except the last item, last item).
- Reducer: Given (prefix, [m1, m2, …]), emit all possible ((prefix, mi, mj), _) as candidates.
- Job3: Given each candidate pattern, count frequency in TDB and keep only those with enough support. (Exercise)

Activity Mining V1

- The final result is the union of the output of all reducers that generate (ActivitySet, count)
- Optimization: (Exercise)
  - Use combiner to compute local sums.
  - Use more than one reducer for candidate generation.
  - Use efficient MR for TDB scanning.

Activity Mining V2

- One pass MapReduce algorithm
- Mapper: Given (TID, Behavior_Items), generate all possible (pattern, 1).
- Reducer: Sum the count for the same pattern and emit (pattern, n) if n >= support.
- That's it !!
- Optimization: (Exercise)
  - Parallelize the mapper?
  - Stop counting whenever n>= support?

Assignment 1

- Test run the word count example. (No need to turn in anything.)
- Implement the Sorting algorithm.
- Implement the Searching algorithm.
- Implement the TF-IDF computation algorithm.
- Implement the Activity Mining algorithms. (optional)
- Due date: three weeks!