

Data Intensive Computing in the Cloud

MAP/REDUCE

1

Map/Reduce

- Key-Value Data Store
- Programming model
- Examples
- Execution model
- Criticism

2

Overview

- New systems have emerged to address requirements of data management in the cloud
 - so-called “NoSQL” data stores
 - scalable SQL databases
- **Horizontal and Vertical Scaling**
 - shared nothing
 - replicating and partitioning data over thousands of servers
 - distribute “simple operation” workload over thousands of servers
- **Simple Operations**
 - key lookups
 - read and writes of one or a small number of records
 - **no** complex queries or joins

3

Defining “NoSQL”

- No agreed upon definition
 - “not only SQL”
 - “not relational”
 - ...
- Six key features
 1. ability to scale simple operation throughput over many servers
 2. ability to replicate and distribute (partition) data over many servers
 3. simple call level interface or protocol (in contrast to a SQL binding)
 4. weaker concurrency model than ACID transactions of most relational (SQL) database systems
 5. efficient use of distributed indexes and RAM for data storage
 6. ability to dynamically add new attributes to data records

Based on: “Scalable SQL and NoSQL Data Stores” by R. Cattell, 2010

4

Key/Value Data Model

k_1	v_1
k_2	v_2
k_3	v_3
⋮	
k_n	v_n

- Interface
 - `put(key, value)`
 - `get(key): value`

- Data storage
 - values (data) are stored based on programmer-defined keys
 - system is agnostic as to the structure (semantics) of the value
- Queries are expressed in terms of keys
- Indexes are defined over keys
 - some systems support secondary indexes over (part of) the value

5

Motivation

- Background and Requirements
 - computations are conceptually straightforward
 - input data is (very) large
 - distribution over hundreds or thousands of nodes
- Programming model for processing of large data sets
 - abstraction to express simple computations
 - hide details of parallelization, data distribution, fault-tolerance, and load-balancing

6

Programming Model

- Inspired by primitives from functional programming languages such as Lisp, Scheme, and Haskell
- Input and output are sets of key/value pairs
- Programmer specifies two functions
 - **map** $(k_1, v_1) \rightarrow \text{list}(k_2, v_2)$
 - **reduce** $(k_2, \text{list}(v_2)) \rightarrow \text{list}(v_2)$
- Key and value domains
 - input keys and values are drawn from a different domain than intermediate and output keys and values
 - intermediate keys and values are drawn from the same domain as output keys and values

7

Map Function

- User-defined function
 - processes input key/value pair
 - produces a set of *intermediate* key/value pairs
- Map function I/O
 - **input**: read from GFS file (chunk)
 - **output**: written to intermediate file on local disk
- Map/reduce library
 - executes map function
 - groups together all intermediate values with the same key
 - “passes” these values to reduce functions
- Effect of map function
 - processes and partitions input data
 - builds distributed map (transparent to user)
 - similar to “group by” operation in SQL

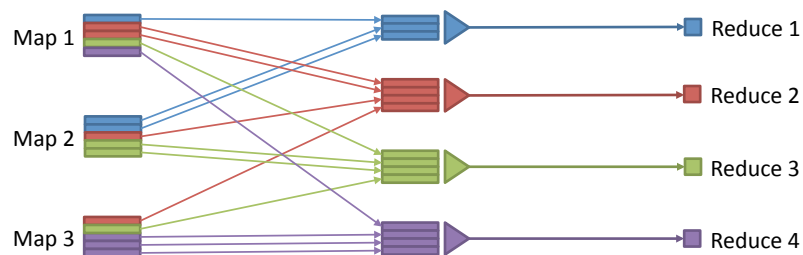
8

Reduce Function

- User-defined function
 - accepts *one* intermediate key and a set of values for that key
 - merges these values together to form a (possibly) smaller set
 - typically, zero or one output value is generated per invocation
- Reduce function I/O
 - **input:** read from intermediate files using remote reads on local files of corresponding mapper nodes
 - **output:** each reducer writes its output as a file back to GFS
- Effect of reduce function
 - similar to aggregation operation in SQL

9

Map/Reduce Interaction



- Map functions create a user-defined “index” from source data
- Reduce functions compute grouped aggregates based on index
- Flexible framework
 - users can cast raw original data in any model that they need
 - wide range of tasks can be expressed in this simple framework

10

Example: Looking Up Friends on Social Networks

- Facebook has a list of friends (bidirectional relationship)
- This list can be seen when visiting a user's profile
- How many friends do two people (users) have in common
- Can pre-compute results and store if list does not change often
- Person \rightarrow [List of Friends]

A \rightarrow B C

B \rightarrow A C

C \rightarrow A B

Each line an argument to mapper

For every friend in the list of friends, mapper outputs (K,V) pair

11

Looking Up Friends on Social Networks (contd)

For map(A \rightarrow B C):

(A,B) \rightarrow B C

(A,C) \rightarrow B C

For map(B \rightarrow A C):

(A,B) \rightarrow A C

(B,C) \rightarrow A C

For map(C \rightarrow A B):

(A,C) \rightarrow A B

(B,C) \rightarrow A B

Group these by their keys to get:

(A,B) \rightarrow (A C)(B C)

(A,C) \rightarrow (A B)(B C)

(B,C) \rightarrow (A B)(A C)

Reduce per line by intersect lists per key:

(A,B) \rightarrow (C)

(A,C) \rightarrow (B)

(B,C) \rightarrow (A)

E.g. when C visits B's profile can look-up (B,C) for friends in common

12

Other Examples

- Distributed “grep”
 - **goal:** find positions of a pattern in a set of files
 - **map:** (File, String) → list(Integer, String), emits a <line#, line> pair for every line that matches the pattern
 - **reduce:** identity function that simply outputs intermediate values
- Count of URL access frequency
 - **goal:** analyze Web logs and count page requests
 - **map:** (URL, String) → list(URL, Integer), emits <URL, 1> for every occurrence of a URL
 - **reduce:** (URL, list(Integer)) → list(Integer), sums the occurrences of each URL
- Workload of first example is in map function, whereas it is on the reduce in the second example

13

Execution Overview

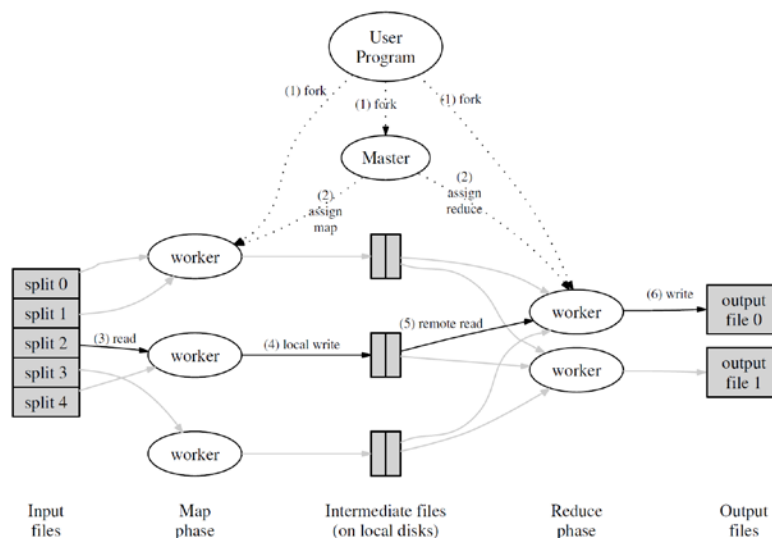


Figure Credit: “MapReduce: Simplified Data Processing on Large Clusters” by J. Dean and S. Ghemawat, 2004

14

Execution Overview

1. Map/reduce library splits input files into M pieces and then starts copies of the program on a cluster of machines
2. One copy is the master, the rest are workers; master assigns M map and R reduce tasks to idle workers
3. Map worker reads its input split, parses out key/value pairs and passes them to user-defined map function
4. Buffered pairs are written to local disk, partitioned into R regions; location of pairs passed back to master
5. Reduce worker is notified by master with pair locations; uses RPC to read intermediate data from local disk of map workers and sorts it by intermediate key to group tuples by key
6. Reduce worker iterates over sorted data and for each unique key, it invokes user-defined reduce function; result appended to reduce partition
7. Master wakes up user program after all map and reduce tasks have been completed

15

Master Data Structures

- Information about all map and reduce task
 - **worker state:** idle, in-progress, or completed
 - **identity** of the worker machine (for non-idle tasks)
- Intermediate file regions
 - propagates intermediate file locations from map to reduce tasks
 - stores locations and sizes of the R intermediate file regions produced by each map task
 - updates to this location and size information are received as map tasks are completed
 - information pushed incrementally to workers that have in-progress reduce tasks

16

Fault Tolerance

- Worker failure
 - master pings workers periodically; assumes failure if no response
 - completed/in-progress map and in-progress reduce tasks on failed worker are rescheduled on a different worker node
 - dependency between map and reduce tasks
- Master failure
 - checkpoints of master data structure
 - can recover after failure of master but progress can halt
- Failure semantics
 - if user-defined functions are *deterministic*, execution with faults produces the same result as execution without faults
 - rely on atomic commits of map and reduce tasks

17

Other Implementation Aspects

- Locality
 - network bandwidth is scarce resource
 - move computation close to data
 - master takes GFS metadata into consideration (location of replicas)
- Task granularity
 - master makes $O(M + R)$ scheduling decisions
 - master stores $O(M * R)$ states in memory
 - M is typically larger than R
- Backup Tasks
 - “stragglers” are a common cause for suboptimal performance
 - as a map/reduce computation comes close to completion, master assigns the same task to multiple workers

18

Map/Reduce Criticism

- “Why not use a parallel DBMS instead?”
 - map/reduce is a “giant step backwards”
 - no schema, no indexes, no high-level language
 - not novel at all
 - does not provide features of traditional DBMS
 - incompatible with DBMS tools
- Performance comparison of approaches to large-scale data analysis
 - Pavlo et al. “A Comparison of Approaches to Large-Scale Data Analysis”, Proc. Intl. Conf. on Management of Data (SIGMOD), 2009
 - parallel DBMS (Vertica and DBMS-X) vs. map/reduce (Hadoop)
 - original map/reduce task: “grep” from Google paper
 - typical database tasks: selection, aggregation, join, UDF
 - 100-node cluster

19

References

- J. Dean and S. Ghemawat: **MapReduce: Simplified Data Processing on Large Clusters**. *Proc. Symp. on Operating Systems Design & Implementation (OSDI)*, pp. 137-149, 2004.
- A. Pavlo, E. Paulson, A. Rasin, D. J. Abadi, D. J. DeWitt, S. Madden, and M. Stonebraker: **A Comparison of Approaches to Large-Scale Data Analysis**. *Proc. Intl. Conf. on Management of Data (SIGMOD)*, pp. 165-178, 2009.
- S. Krenzel: **MapReduce: Finding Friends**, 2010.
- Y. Bu, B. Howe, M. Balazinska, and M. D. Ernst: **HaLoop: Efficient Iterative Data Processing on Large Clusters**. *Proc. Intl. Conf. on Very Large Data Bases (VLDB)*, pp. 285-296, 2010.

20

That's All Folks!



*"It was much nicer before people started
storing all their data in the Cloud."*

21