Map/Reduce

Sharma Chakravarthy
Information Technology Laboratory
Computer Science and Engineering Department
The University of Texas at Arlington, Arlington, TX 76009
Email: sharma@cse.uta.edu
URL: http://itlab.uta.edu/sharma

1

Tutorial Outline

- ***Hadoop***
- ***Map/reduce***

2

Acknowledgements

- These slides are put together from a variety of sources (both papers and slides/tutorials available on the web)
- Mostly I have tried to: provide my perspective, emphasize aspects that are of interest to this course, and have tried to put forth a consolidated view of Map/Reduce

3

Data Center as a computer [Patterson, cacm 2008]

- Claim: There are dramatic differences between – developing software for millions to use as a service versus distributing software for millions to run on their PCs
  - Availability, dependability
  - Bandwidth (with low latency) to service large number of users
  - Innovation is fast as the software is in their control!
- This has led to distributed data centers
- Microsoft model vs. the new model

4

Data Center as a computer [Patterson, cacm 2008]

- What are the useful programming abstractions for such a large system?
- How do thousands of computers behave differently from a small system?
- What must you do differently to run an abstraction on thousands of computers?
- Google proposed a two-phase primitive:
  - Phase 1: maps a user supplied function onto thousands of computers (mapper)
  - Phase 2: Reduces the returned values from all those instances into a set of results (reducer)

5

Data Center as a computer [Patterson, cacm 2008]

- Map/Reduce was born
- Runs on heterogeneous computers (even different generations)
- Runs on heterogeneous OSs
- Scheduler is dynamic and accommodates above (as compared to batch schedulers of Grid computing
- Failures are handled transparently!
- Hadoop sorted 1 TB in 209 seconds (2009) !!
- Google regenerated its index using Map/Reduce
- Fairly easy to program, easy to understand!

6

Map/Reduce

- Is a programming paradigm
- Is a parallel programming paradigm
  - Multi-threaded (typically shared memory) or
  - Distributed (shared nothing) more complex
- Is derived from functional programming (specification vs. procedural programming) (remember SQL is non-procedural)
- Many a times the question asked is:
  - Is there a difference between Map/Reduce and traditional parallel programming?

7

Data vs. Task parallelism

- Data parallelism means concurrent execution of the same task (code) on multiple computing nodes (cores)
  - Typically on different chunks (subsets) of data
  - Amount of parallelization can be based on input size (and affordability)
  - Load balancing can be done for optimizing response time
  - No communication among the computing nodes (difference from parallel computing)
  - Number of chunks need not be the same as the number of computing nodes

8

Data vs. Task parallelism

- Task parallelism means concurrent execution of the different tasks (code) on multiple computing nodes (cores)
  - Same or different chunks (subsets) of data
  - Amount of parallelization is proportional to the number of independent tasks performed
- Does map/reduce use data parallelism or task parallelism?
  - Data parallelism (Why?)

9

Map/Reduce

- Map/Reduce was developed within Google as a mechanism for processing large amounts of raw data; for example, crawled documents or web request logs.
- This data is so large, it must be distributed across thousands of machines in order to be processed in a reasonable time.
- This distribution implies parallel computing since the same computations are performed on each CPU, but with a different dataset.
- Map/Reduce is an abstraction that allowed Google engineers to perform simple computations while hiding the details of parallelization, data distribution, load balancing and fault tolerance.

10

What is Map/Reduce?

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

11

What is Map/Reduce 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

Note that most/all of these applications are very different from traditional DBMS applications:
very large data sizes
one time computation versus data storage and management
SQL is not a good vehicle/tool for doing this task
Defining schema and loading this data into a DBMS is difficult

12

What is Map/Reduce used for (2)?

- Also used for:
  - Graph mining
  - PageRank calculation
  - Machine learning
  - Shortest path
- Problems are being ported to map/reduce on a daily basis
- Is a popular research area!
- Porting algorithms into map/reduce is not always straightforward!

Note that most/all of these applications are very different from traditional DBMS applications:
very large data sizes
one time computation
versus data storage and management
SQL is not a good vehicle/tool for doing this task
Defining schema and loading this data into a DBMS is difficult

13

What is Map/Reduce used for?

- In research:
  - Analyzing Wikipedia conflicts (PARC)
  - Natural language processing (CMU)
  - Bioinformatics (Maryland)
  - Astronomical image analysis (Washington)
  - Ocean climate simulation (Washington)
  - Graph Mining (UTA)
  - Multilayer Network Analysis (MLN)
  - Storm Identification from rainfall data (UTA)

14

MapReduce Design Goals

1. **Scalability** to large data volumes:
   - Scan 100 TB on 1 node @ 50 MB/s = 23 days
   - Scan on 1000-node cluster = 33 minutes
2. **Cost-efficiency:**
   - Commodity nodes (cheap, but unreliable)
   - Commodity disks (cheap and raid makes them reliable)
   - Commodity network
   - Automatic fault-tolerance (fewer admins)
   - Easy to use (fewer programmers)

15

Map/Reduce

- Automatic parallelization & distribution
- Fault-tolerant
- Provides status and monitoring tools
- Clean abstraction for programmers
- Borrows from functional programming
- Users implement interface of two functions:

map (in_key, in_value) → (int_key, intermediate_value list)

reduce (int_key, intermediate_value list) → (out_key, value list)

In_key, int_key, and out_key need not be same!

16

Typical cluster farm

17

The Basics (5)

- Note that we cannot assume that every problem can be parallelized
- Some problems are easier to parallelize and some are inherently sequential
- So it is important to understand whether a problem can be parallelized and to what extent!
- Cryptography hash-chaining computations are very difficult to parallelize
- Parallelizing I/O is difficult (needs additional technology)

18

Map/Reduce Framework

- Needs to be understood at several levels: Job level
  - Configuring a map/reduce job to run it on desired number of mappers and reducers
  - Whether a single input is processed or multiple inputs
  - Whether a single job (which is a map/reduce pair) is used or they are chained
  - Whether it is an iterative algorithm or one-shot algorithm
  - Other issues, such as reading other files in the mapper or reducer etc.

19

Map/Reduce Framework

- Needs to be understood at several levels: Map/Reduce
  - Code that you write and its understanding as well conformity with the paradigm (we will see this later)
  - Mapper code (one or more)
    - Setup part
    - Main part
    - Cleanup part
  - Same for the reducer code
  - Combiner code (if needed)
  - Writing intermediate results and computations, if any, between iterations, etc.

20

Map/Reduce Framework

- Needs to be understood at several levels: Behind the Scenes (including shuffle)
  - What is done transparently by the Hadoop framework?
  - At the end of mapper code
  - At the end of combiner code(if there is)
  - How are partitions (shards) handled by Hadoop?
  - How are multiple mappers used for the same input?
  - What is partitioning? And who does it?
  - What is sorting needed? And who does it?
  - What is shuffle? Principle behind the shuffle

21

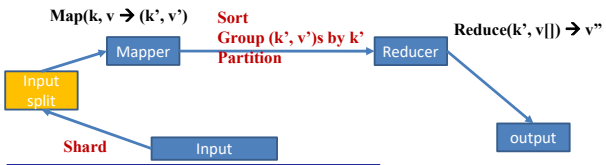
Map/Reduce Framework

- Without clearly understanding all the three levels, you cannot write meaningful map/reduce algorithms
- Please understand each clearly
- The fact that many machine learning algorithms have been successfully translated/mapped to this framework is indicative of its utility
  - Although the paradigm seems very simple
  - User code is typically not very big
  - Some of the heavy lifting is done by the underlying framework (hence need to understand it)

22

MapReduce

- Input: a set of key/value pairs (generated from input data)
- User supplies two functions:
  - map(k,v) → (k1, v1), … → list of (k1,list(v1))
  - reduce(k1, list(v1)) → list of (k2, v2)



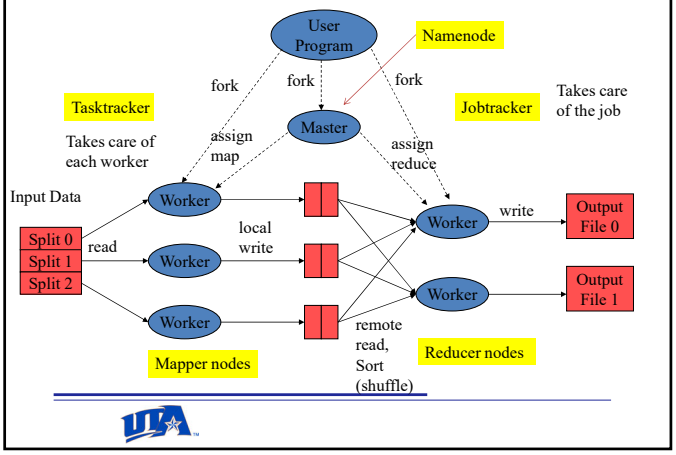
23

MapReduce Programming Model

- Data type: key-value *records*
- Map function:
  (K_{in}, V_{in}) ➔ (K_{out}, V_{out}), … → {<K_{inter}, list(V_{out})>, …}
- Reduce function processes EACH key-valueList
  (K_{inter}, list(V_{inter})) ➔ list(K_{out}, V_{out})

24

Distributed Execution Overview



25

Map/Reduce: job execution

- The MASTER:
  - initializes the data and splits it as specified or based on the number of available WORKERS (# of mappers can be chosen, irrespective of the number of splits)
  - Sends each WORKER its partition/split (task)
  - receives the results from each WORKER (its location and some meta information)
- The WORKER:
  - receives the partition/split (actually its location) from the MASTER
  - performs processing on the partition (with user-supplied code)
  - Notifies the MASTER when completed (with meta info)

26

Map/Reduce: job execution

- Map code/method written by a user using the Map/Reduce library takes as input <key, value> pair and produces a set of intermediate <key, value> pairs.
- Mapper
  groups together all intermediate values associated with the same intermediate key k as <k, value-List>
- Results from mapper is sent to the reducer(s) through the process of shuffling
  - After bringing them to <key, value-list> format
- Reduce code written by a user using the Map/Reduce library takes as input multiple <key, value-list>'s (a partition) and produces the result for that job. Each reducer writes its output separately to HDFS
- If a combiner is defined, it is executed **before shuffle**

27

What happens in the Mapper

- Map code/method takes an input <key, value> pair and produces zero or more intermediate <key, value> pairs
  - Mapper processed **ONE <key, value>** at a time
- Each spill (will be discussed later) is partitioned (using a hash function), a group by is done on the key (i.e., sorted on the key and all key values grouped)
  - All values for the same key are merged creating a value-list
- This is done for each spill as there can be many spills within a Mapper
  - Each spill corresponds to a buffer size (actually 80%)
- Number of partitions above is based on the number of reducers specified (default is 1)
- At this point, the mapper has n partitions for each spill (on the disk) (why?)
- If a combiner is provided, it is executed on each partition of each spill.
- The output of a combiner is also a <key, value-list>
- Same partitions from different spills are merged and sorted on key value.
- These partitions from each mapper is sent to different reducers
  - This is the shuffle

28

What does the framework do in the mapper

- Remember, any computation has to be done in a physical processor (or core). Cannot be done in thin air!
  - partitioning
  - Sorting
  - merging
- Shuffle is NOT a computation! It is shuffling or exchanging data from **all** mappers to **all** reducers over the network!
- Many students think shuffling is sort of magic!
  - It is not!
  - It is a step to send data from different mappers to EACH reducer!
  - Without understanding this, you cannot write reduce code

29

In the Reducer

- Shuffle sends several <key, value-list> to each reducer from all mappers (how does it determine which <key, value-list> to send to who?)
- A reducer reads the **same numbered partition** from each mapper over the network!
- Each reducer merges the value-list of the same key coming from different mappers to create a single **global <key, value-list> for each key in the partition**
  - Now, they are sorted on key
- The reduce function, also written by the user, acts upon each <key, value-list> in that partition
  - It operates on one <key, value-list> at a time and produces the desired result and writes to HDFS

30

Testing Your understanding

- Are the keys in <key, value-list> generated by the mappers ordered before applying the combiner?
  - Yes (Why?)
- Are the keys in <key, value-list> generated by the mappers ordered after applying the combiner?
  - They are already ordered(Why?)
- Are the keys in <key, value-list> from each mapper ordered when it comes to the reducer?
  - Yes (Why?)
- Are the values in the <key, value-list> produced by the mapper ordered?
  - No (why?)
- Are the values in each <key, value-list> processed by the reducer code ordered?
  - No (why?)

31

NFS

- Let us understand the network file system (NFS) before going into HDFS
- Provides remote access to a single logical volume stored on a single machine
- Makes portions of its local file system visible to external clients
- Clients can mount the remote file system and make it look like a local drive

  \+ transparency

  \- Size, because it is on one single machine

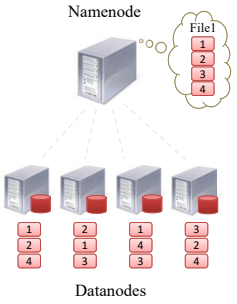
  \- Single point of failure

  \- bottleneck

32

Hadoop Distributed File System (HDFS)

- Files split into 64MB *blocks* *(Access uses 128MB by default)*
- Blocks replicated across several *datanodes* (usually 3)
- Single *namenode* stores metadata (file names, block locations, etc.)
- Optimized for large files, sequential reads



33

HDFS

- HDFS can store large amounts of data (TB to PB)
- Is spread across several machines
- Much larger file sizes than NFS
- stores data reliably. If individual machines in the cluster malfunction, data is still be available.
- Provides fast, scalable access to this information. It is possible to serve more clients by simply adding more machines to the cluster.
- Integrates well with Hadoop MapReduce, allowing data to be read and computed upon locally when possible.

34

HDFS (2)

- HDFS is **not** a general-purpose NFS
- Applications that use HDFS are assumed to perform long sequential streaming reads from files.
- HDFS is optimized to provide streaming read performance; this comes at the expense of random seek times to arbitrary positions in files.
- Data will be written to the HDFS once and then read several times; updates to files after they have already been closed are not supported.
- An extension to Hadoop provides support for appending new data to the ends of files.

35

HDFS (4)

- Due to the large size of files, and the sequential nature of reads, the system does not provide a mechanism for local caching of data. The overhead of caching is great enough that data should simply be re-read from HDFS source.
- Individual machines are assumed to fail on a frequent basis, both permanently and intermittently.
- The cluster must be able to withstand the complete failure of several machines, possibly many happening at the same time (e.g., if a rack fails all together).
- While performance may degrade proportional to the number of machines lost, the system as a whole should not become overly slow, nor should information be lost. Data replication strategies combat this problem.

36

HDFS details

- HDFS is a distributed, scalable, and portable file system written in Java for the Hadoop framework.
- Each node in a Hadoop instance can be a DataNode; a cluster of DataNodes form the HDFS cluster.
- Each DataNode serves up blocks of data over the network using a block protocol specific to HDFS.
- The file system uses the TCP/IP layer for communication; clients use RPC to communicate between each other.

37

HDFS

- HDFS stores large files (an ideal file size is a multiple of 64MB[9]), across multiple machines.
- It achieves reliability by replicating the data across multiple hosts, and hence does not require RAID storage on hosts.
- With the default replication value of 3, data is stored on three nodes: two on the same rack, and one on a different rack.
- Data nodes can talk to each other to rebalance data, to move copies around, and to keep the replication of data high.

38

Data Flow

- Input, final output are stored on a distributed file sy stem (e.g., HDFS)
- Scheduler tries to schedule map tasks "close" to physical storage location of input data
- Intermediate results are stored on local FS of map and reduce workers
- Output is often input to another map reduce job
- Not all problems solved with one round of map /reduce computation!

39

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
- How does this compare with recovery of DBMS?
  - Similarities and differences?

40

Map/Reduce Execution Overview

- Map invocations are distributed across multiple machines by automatically partitioning the input data into a set of S spl its/shards. The input shards can be processed in parallel on different machines.
  - Each shard becomes a mapper task
  - One mapper processes one shard at a time
  - Number of mapper nodes need not be the same as the number of splits
- Reduce invocations are distributed by partitioning the **intermediate key space** into R disjoint groups using a partitioning function (e.g., hash(key) mod R).
- Number of reducers can be specified.
- By writing your own partition function, you can determine the number of partitions

41

Testing Your understanding

- What happens if the number of splits are more than the number of mappers?
  - Splits are processed using the same mapper nodes until all splits are processed!
- What happens if the number of splits are less than the number of mappers?
  - Some mapper nodes are unused (waste of resources!)
- When do we get maximum parallelism for mapper processing?
  - Splits = number of mapper nodes
  - However, a mapper node is capable of processing several mapper tasks. There is a formula associated with it.
- If number of reducers are given, what is its effect on the partitioner?
  - Number of partitions created is equal to the number of reducers
- If number of custom partitions is greater than the reducers specified, what happens?
  - I believe partition numbers greater than the number of reducers are NOT processed!
- Do the **reducers have to wait** for all mapper tasks to complete?
  - Yes. User reduce code cannot start until all mapper tasks are done
  - However, shuffle can overlapped with mapper task to improve response time (user can set this parameter)

42

Testing Your understanding

- Can the number of reducer nodes be more than the number of mapper nodes?
  - Yes (under what circumstances)
  - Space of mapper output keys can be larger than mapper input keys! So more reducers may be needed to process them!
- How does one determine the number of reducers?
  - Based on the key space produced by the mappers!
- Are there situations where more than one reducer cannot be used?
  - Yes! Depends on what you are computing
  - If you are computing average of entire input, they cannot be computed on (key) partitions. Only one partition need to be used
  - However, by computing the numerator and denominator separately, multiple reducers can be used
    - But the final result need to be computed outside (after the job)

43

Testing Your understanding

- What about the hash function? How complex is it?
  - A simple mod function is used as default
- Do all mappers need to use the **SAME hash function**?
  - **Absolutely**! Otherwise, same keys from different mappers will end up in different reducers messing up the whole thing
  - You need to keep this mind if you are specifying your own partition function
  - You also need to make sure number of reducers specified matches the number of partitions generated by the custom partitioner

44

Clarity on Map/Reduce

- User can specify the number of splits (using size) or it is done by the system using default size (64 or 128MB)
- Number of mapper nodes can be specified. A node corresponds to a processor (physical resource). Each node can perform ONE mapper task at a time or in case of cores may be able to perform multiple mapper tasks on the same node using cores
- Number of Reducer nodes can be specified. Hadoop will partition the output of mappers and send it to corresponding reducers. A reducer task is run on a reducer node. But each reducer task processes a number of <key, value-list>s
- Number of mapper and reducer nodes can be determined independently. But is usually not ad hoc. They are determined based on the amount of computation to be done and dividing the computation among mappers/reducers (response time)
  - Or for experiments to understand speed up etc.

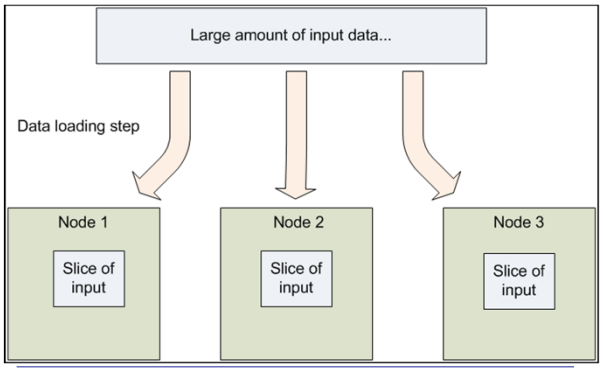
45

Clarity on Map/Reduce (2)

- Mapper method/function written by the user is executed by each instantiation of mapper task. It processes *one* <key1, value1> pair at a time and produces *one or more* <keyi, valuei> pairs
- Optionally, a combiner method/function written by the user is executed AFTER each input split is processed completely by the mapper method/function as part of the mapper task
- All the <key, value> pairs are partitioned, sorted on the key and grouped on the key into <key, value-list> pairs as input to the combiner. We will later see that this input is similar to the input to the reducer with a subtle (local vs. global) difference!
- The output of the combiner is converted to a set of <key, value-list> and partitioned on the key before shuffling
- These <key, value-list> are shuffled to one or more reducers, and they are merged in each reducer BEFORE applying the reduce method/function

46

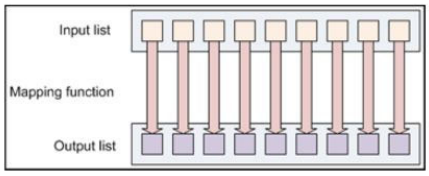
Data distribution across nodes



47

Mapping Lists

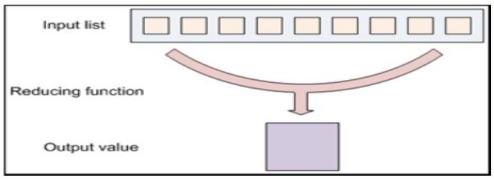
- A list of data elements are provided, one at a time, to a function called mapper, which transforms each element individually to one or more output elements



48

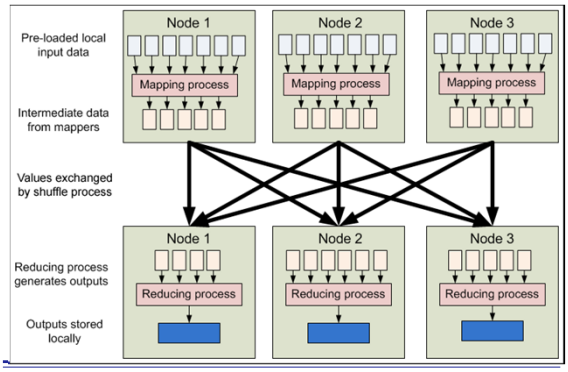
Reducing List

- Reducing lets you aggregate values together. A reducer function receives an iterator of input values from an input list. It them combines these values together, returning an output value

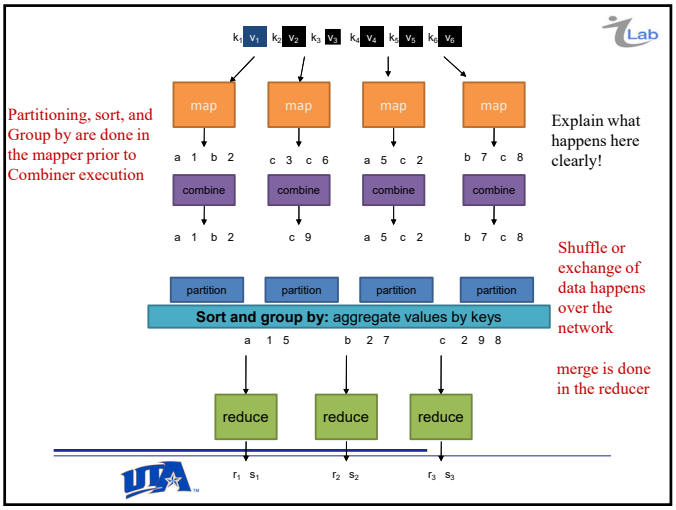


49

Shuffling in Map/Reduce (high level view/diagram)



50



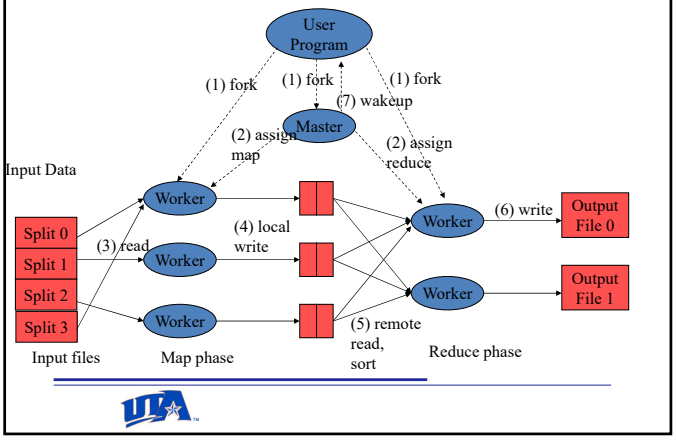
51

Coordination

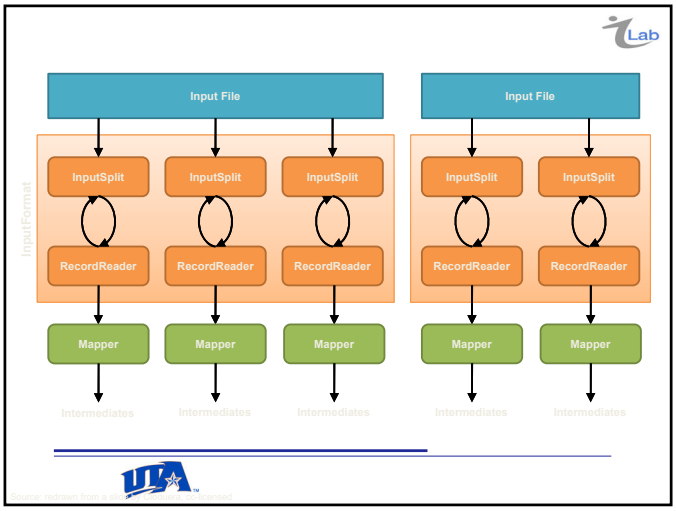
- Master data structures
  - Task status: (idle, in-progress, completed)
  - Idle tasks get scheduled as workers become available
  - When a map task completes, it sends the master the location and sizes of its R intermediate files, one for each reducer
  - Master pushes this info to reducers
- Master pings workers periodically to detect failures
- Static load balancer: allocates processes to processors at run time while taking no account of current network load.

52

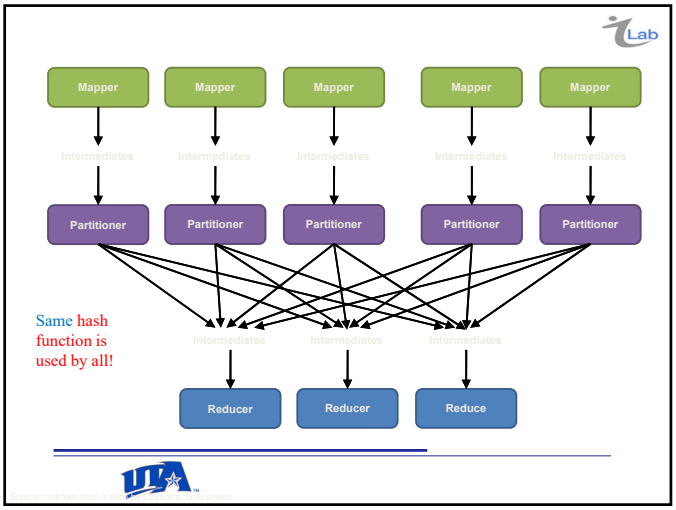
Distributed Execution Overview



53



54



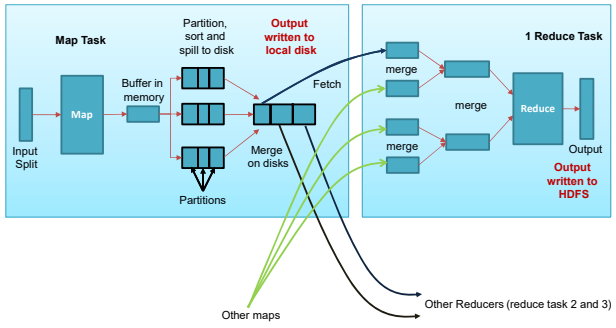
55

Behind the scenes: Partition, Sort and group by in Mapper

- Probably the most complex aspect of this paradigm
- Also, transparent to the user
- Difficult to write combiner or reducer code without understanding this!
- Map side
  - Map outputs are buffered in memory in a circular buffer (ring buffer). Default size 100MB
  - When buffer reaches threshold (80%), contents are "spilled" to disk
  - Each "spill" is partitioned (on key) and group by is applied (sorted and values in each partition are merged into value-list) before writing each partition as separate files on disk
  - How many files are there for each spill?

56

Partition, sort, merge, and shuffle



57

Level2: Partition, Sort and group by in Hadoop (2)

- Once all the spills are processed, each partition file from each spill is merged to create a <key, value-list> for EACH key
- This is processed by the combiner, if there is one
- If not, this is shuffled to reducers!
- Now, you can see why a combiner is very similar to a reducer, EXCEPT, it works on the output of a SINGLE mapper tas
  - hence, does local reduce, where as a reducer does a global reduce
- Whereas a reducer works on the global <key, value-list> as SAME <key, value-list> comes from DIFFERENT mappers
- Hence, it must merge them BEFORE applying the reduce function!

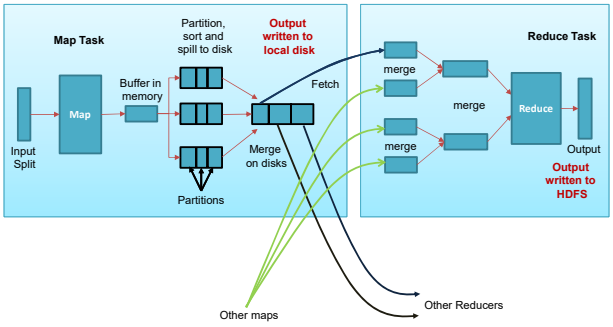
58

Behind the scenes: Partition, Sort and group by in Hadoop

- Reducer side
  - Each mapper sends the SAME partition (read multiple <key, value-list>) to each reducer.
  - They are sorted on key! Values are NEVER processed by Hadoop!
  - If 3 partitions come to a reducer from 3 different mappers each containing a number of <key, value-list>
    - Each <key, value-list> with the SAME key, but from different mappers need to be MERGED into one <key, value-list> for processing by the reduce code!
    - These merged <key, value-list> is processed by the user reduce code
    - Reducer outputs <key, value-list>

59

Hadoop MapReduce Architecture



60

Partition and shuffle

- After the initial map tasks have completed, the map nodes may still be performing several more map tasks each. But they also begin exchanging the intermediate outputs from the map tasks to where they are required by the reducers.
- This process of moving map outputs to the reducers can overlap wit other map tasks and what the reducer does transparently (considered part of shuffling)
- A different subset of the intermediate key space is assigned to each reduce node; these subsets (known as "partitions") are the inputs to the reduce tasks.
- Each map task may emit (key, value) pairs to any partition; all values for the same key are always reduced together regardless of which mapper is its origin. Therefore, the map nodes must all agree on where to send the different pieces of the intermediate data.

61

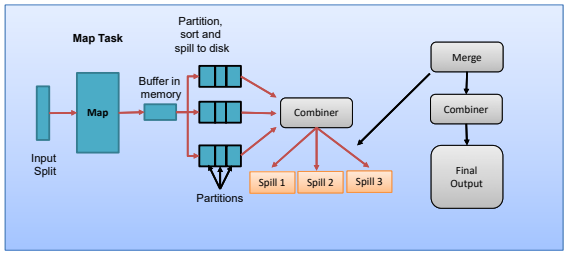
Partition and shuffle

- The *Partitioner* class determines which partition a given (key, value) pair will go to. The default partitioner computes a hash value for the key based on the number of reducers.
- **Merge:** Each reducer is responsible for merging the values associated with several intermediate keys. The set of intermediate keys on a single node is automatically merged by Hadoop before they are presented to the user written Reducer code
- Reducer can receive shuffle (data collection from different mappers) even if all mapper tasks are not complete! This % can be user specified. (why is this good?)
- However, merge and reduce cannot happen until all mapper tasks are done and shuffle is complete! (why is this?)

62

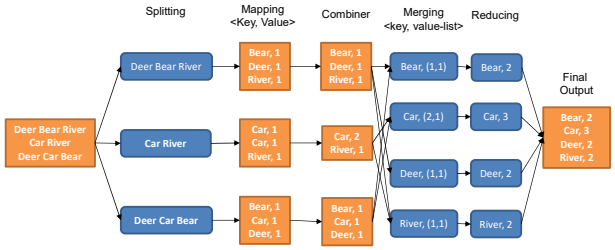
Use of Combiner

- Combiner works as a mini reducer



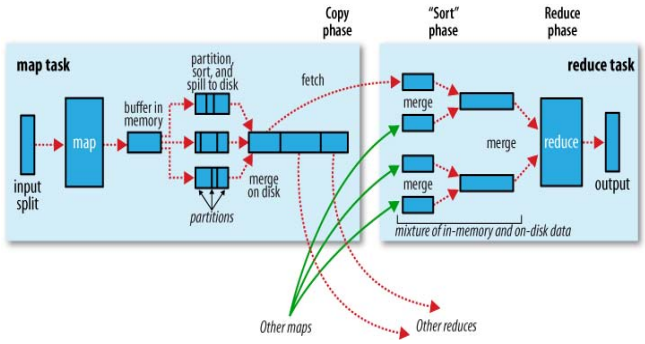
63

MapReduce Example (not a good one!)



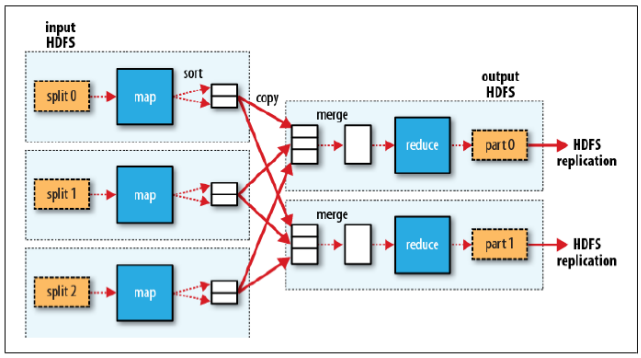
64

Partition, shuffle, sort



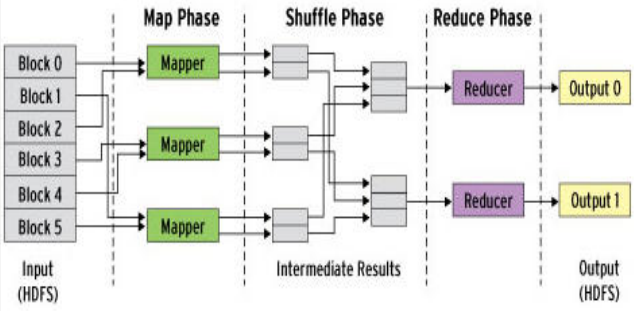
65

Partition, shuffle, sort



66

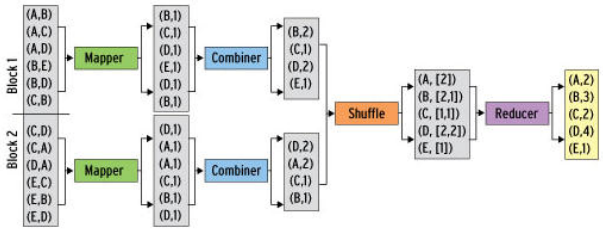
Map tasks and mapper nodes



67

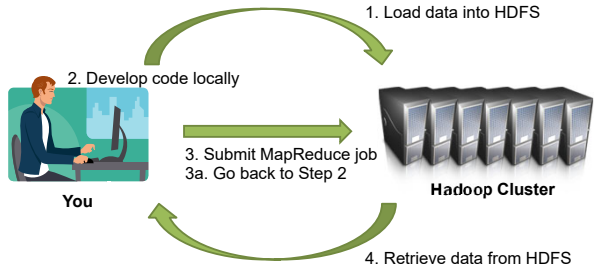
A not so good example

- Counting letters



68

Hadoop Workflow



69

Mapper

- The Mapper performs the interesting **user-defined** work of the first phase of the Map/Reduce program.
- Given a key and a value, the map() method emits (key, value) pair(s) which are forwarded to the Reducers.
- A new instance of Mapper is instantiated in a separate Java process for each map task (InputSplit) that makes up part of the total job input.
- The individual mappers are intentionally not provided with a mechanism to communicate with one another in any way. This allows the reliability of each map task to be governed solely by the reliability of the local machine.
- The map() method receives two parameters in addition to the key and the value: output collector and reporter objects

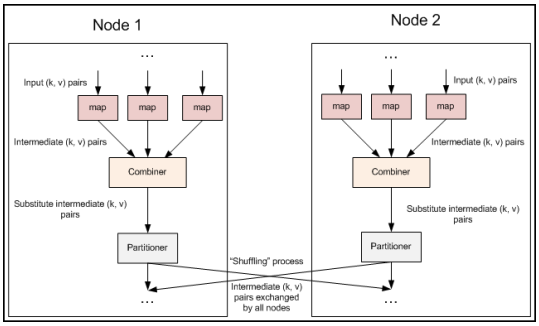
70

Reduce

- A Reducer instance is created for each reduce task. This is an instance of **user-provided** code that performs the second important phase of job-specific work.
- For each key in the partition assigned to a Reducer, the Reducer's reduce() method is called once. This receives a key as well as an iterator over all the values associated with the key. The values associated with a key are returned by the iterator in an undefined order.
- The Reducer also receives as parameters *OutputCollector* and *Reporter* objects; they are used in the same manner as in the map() method.

71

Additional Map/Reduce functionality



72

Fault tolerance

- This fault tolerance underscores the need for program execution to be side-effect free.
- If Mappers and Reducers had individual identities and communicated with one another or the outside world, then restarting a task would require the other nodes to communicate with the new instances of the map and reduce tasks, and the re-executed tasks would need to reestablish their intermediate state (remember cascading rollbacks or aborts)
- This process is notoriously complicated and error-prone in the general case.
- Map/Reduce simplifies this problem drastically by eliminating task identities or the ability for task partitions to communicate with one another. An individual task sees only its own direct inputs and knows only its own outputs, to make this failure and restart process clean and dependable.

73

MR Fault tolerance and DBMS Recovery

- This fault tolerance underscores the need for program execution to be side-effect free.
- This requirement is also needed/used in the recovery of a DBMS using logs.
- If a transaction were to communicate with outside (i.e., outside of reading and writing from disks, and with others), recovery becomes very complicated and may not even be feasible.
- DBMS recovery aims at restoring the state of the DBMS to a consistent state so that transactions aborted can be re-executed from a consistent state
- It also requires that each transaction leaves the DBMS in a consistent state if it completes!
- ACID property (which is much stronger than what is used in MR) is guaranteed in a DBMS

74

Multiple input jobs

- Suppose you want to do a equi-join of two relations using map/reduce
- This requires two inputs to be processed by different mappers and sent to common set of reducers by all the mappers
- Further, you will have to keep track of which tuple/list came from which input.
  - This is done through tagging.

75

Chaining jobs

- Not every problem can be solved with a Map/Reduce job, but fewer still are those which can be solved with a single Map/Reduce job. Many problems can be solved with Map/Reduce, by writing several Map/Reduce jobs which run in sequence to accomplish a goal:
- Map1 -> Reduce1 -> Map2 -> Reduce2 -> Map3...
- You can easily chain jobs together in this fashion by writing multiple driver methods, one for each job. Call the first driver method, which uses JobClient.runJob() to run the job and wait for it to complete. When that job has completed, then call the next driver method, which creates a new JobConf object referring to different instances of *Mapper* and *Reducer*, etc.

76

Chaining examples

- Suppose you want to compute
  - Single Source Shortest Path
  - Page Rank
  - Graph substructures
- The above problems cannot be done in one iteration.
- This means several map/reduce pairs have to be chained to solve the problem!

77

Chaining jobs

- The first job in the chain should write its output to a path which is then used as the input path for the second job. This process can be repeated for as many jobs are necessary to arrive at a complete solution to the problem.
- Many problems which at first seem impossible in Map/Reduce can be accomplished by dividing one job into two or more.
- Hadoop provides another mechanism for managing batches of jobs with dependencies between jobs. Rather than submit a JobConf to the JobClient's runJob() or submitJob() methods, org.apache.hadoop.mapred.jobcontrol.Job objects can be created to represent each job;
- Dependencies can be accommodated

78

Summary

- Additional features such as pipes and streaming are available in Hadoop.
- If you are familiar with C++ or Java, it is not very difficult to understand the basic concept and use it
- Of course, if you want to use advanced features, you need to learn them
- Much easier than using a DBMS for some jobs where the data is in free format; will discuss more of this later!
- https://hadoop.apache.org/
- https://hadoop.apache.org/docs/r3.2.2/hadoop-mapreduce-client/hadoop-mapreduce-client-core/MapReduceTutorial.html
- https://hadoop.apache.org/docs/r3.2.2/hadoop-project-dist/hadoop-hdfs/HdfsUserGuide.html

79

Questions !

80