CompSci 516
Database Systems

Lecture 11
Map-Reduce
and
Spark

Instructor: Sudeepa Roy

Announcements

- Practice midterm posted on sakai
  - First prepare and then attempt!
- Midterm next Thursday 10/11 in class
  - Closed book/notes, no electronic devices
  - Everything until and including today's lecture (Lecture 11) included
- HW2 to be published soon
  - First run your code on local machine to ensure that it is correct, then on AWS

Reading Material

- Recommended (optional) readings:
  - Chapter 2 (Sections 1,2,3) of Mining of Massive Datasets, by Rajaraman and Ullman: http://i.stanford.edu/~ullman/mmds.html
  - Original Google MR paper by Jeff Dean and Sanjay Ghemawat, OSDI' 04: http://research.google.com/archive/mapreduce.html
  - "Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing" (see course website) – by Matei Zaharia et al. - 2012

Acknowledgement:
Some of the following slides have been borrowed from Prof. Shivnath Babu, Prof. Dan Suciu, Prajakta Kalmegh, and Junghoon Kang

Map Reduce

Big Data

it cannot be stored in one machine → store the data sets on multiple machines → Google File System (Will learn distributed DBMS later)

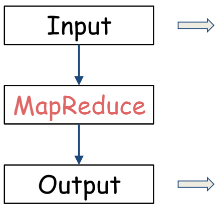
it cannot be processed in one machine → parallelize computation on multiple machines → MapReduce (Today!)

Ack: Slide by Junghoon Kang

The Map-Reduce Framework

- Google published MapReduce paper in OSDI 2004, a year after the Google File System paper
- A high level programming paradigm
  - allows many important data-oriented processes to be written simply
- processes large data by:
  - applying a function to each logical record in the input (map)
  - categorize and combine the intermediate results into summary values (reduce)

Where does Google use MapReduce?



- crawled documents
- web request logs

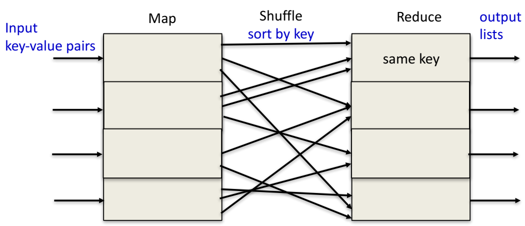
- inverted indices
- graph structure of web documents
- summaries of the number of pages crawled per host
- the set of most frequent queries in a day

Ack: Slide by Junghoon Kang

Storage Model

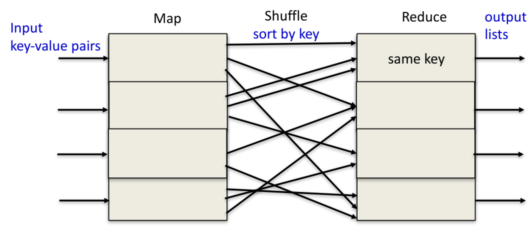
- Data is stored in large files (TB, PB)
  - e.g. market-basket data (more when we do data mining)
  - or web data
- Files are divided into chunks
  - typically many MB (64 MB)
  - sometimes each chunk is replicated for fault tolerance (later in distributed DBMS)

Map-Reduce Steps



- Input is typically (key, value) pairs
  - but could be objects of any type
- Map and Reduce are performed by a number of processes
  - physically located in some processors

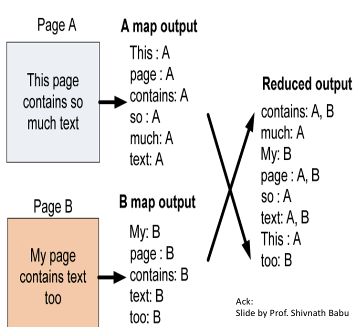
Map-Reduce Steps



1. Read Data
2. Map – extract some info of interest in (key, value) form
3. Shuffle and sort
   - send same keys to the same reduce process
4. Reduce
   - operate on the values of the same key
   - e.g. transform, aggregate, summarize, filter
5. Output the results (key, final-result)

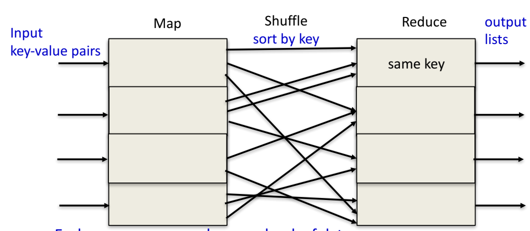
Simple Example: Map-Reduce

- Word counting
- Inverted indexes



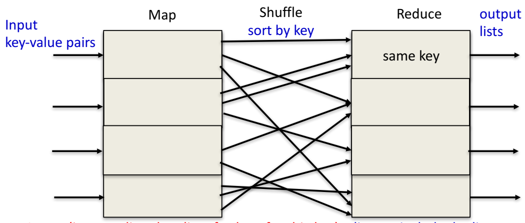
Ack:
Slide by Prof. Shivnath Babu

Map Function



- Each map process works on a chunk of data
- Input: (input-key, value)
- Output: (intermediate-key, value) -- may not be the same as input key value
- Example: list all doc ids containing a word
  - output of map (word, docid) – emits each such pair
  - word is key, docid is value
  - duplicate elimination can be done at the reduce phase

Reduce Function



- Input: (intermediate-key, list-of-values-for-this-key) – list can include duplicates
  - each map process can leave its output in the local disk, reduce process can retrieve its portion
- Output: (output-key, final-value)
- Example: list all doc ids containing a word
  - output will be a list of (word, [doc-id1, doc-id5, ….])
  - if the count is needed, reduce counts #docs, output will be a list of (word, count)

Example Problem: Map Reduce

Explain how the query will be executed in MapReduce

- **SELECT a, max(b) as topb**
- **FROM R**
- **WHERE a > 0**
- **GROUP BY a**

Specify the computation performed in the map and the reduce functions

Map

SELECT a, max(b) as topb
FROM R
WHERE a > 0
GROUP BY a

- Each map task
  - Scans a block of R
  - Calls the map function for each tuple
  - The map function applies the selection predicate to the tuple
  - For each tuple satisfying the selection, it outputs a record with key = a and value = b

•When each map task scans multiple relations, it needs to output something like
key = a and value = ('R', b)
which has the relation name 'R'

Shuffle

SELECT a, max(b) as topb
FROM R
WHERE a > 0
GROUP BY a

- The MapReduce engine reshuffles the output of the map phase and groups it on the intermediate key, i.e. the attribute a

•Note that the programmer has to write only the map and reduce functions, the shuffle phase is done by the MapReduce engine (although the programmer can rewrite the partition function), but you should still mention this in your answers

Reduce

SELECT a, max(b) as topb
FROM R
WHERE a > 0
GROUP BY a

- Each reduce task
  - computes the aggregate value **max(b) = topb** for each group (i.e. ***a***) assigned to it (by calling the reduce function)
  - outputs the final results: **(a, topb)**

A local combiner can be used to compute local max before data gets reshuffled (in the map tasks)

• Multiple aggregates can be output by the reduce phase like
key = a and value = (sum(b), min(b)) etc.

• Sometimes a second (third etc) level of Map-Reduce phase might be needed

More Terminology

however, there is no uniform terminology across systems

- A Map-Reduce "Job"
  - e.g. count the words in all docs
  - complex queries can have multiple MR jobs
- Map or Reduce "Tasks"
  - A group of map or reduce "functions"
  - scheduled on a single "worker"
- Worker
  - a process that executes one task at a time
  - one per processor, so 4-8 per machine
- A master controller
  - divides the data into chunks
  - assigns different processors to execute the map function on each chunk
  - other/same processors execute the reduce functions on the outputs of the map functions

Ack: Slide by Prof. Dan Suciu

Why is Map-Reduce Popular?

- Distributed computation before MapReduce
  - how to divide the workload among multiple machines?
  - how to distribute data and program to other machines?
  - how to schedule tasks?
  - what happens if a task fails while running?
  - … and … and ...
- Distributed computation after MapReduce
  - how to write Map function?
  - how to write Reduce function?
- Developers' tasks made easy

Ack: Slide by Junghoon Kang

Handling Fault Tolerance in MR

- Although the probability of a machine failure is low, the probability of a machine failing among thousands of machines is common
- Worker Failure
  - The master sends heartbeat to each worker node
  - If a worker node fails, the master reschedules the tasks handled by the worker
- Master Failure
  - The whole MapReduce job gets restarted through a different master

Ack: Slide by Junghoon Kang

Other aspects of MapReduce

- Locality
  - The input data is managed by GFS
  - Choose the cluster of MapReduce machines such that those machines contain the input data on their local disk
  - We can conserve network bandwidth
- Task granularity
  - It is preferable to have the number of tasks to be multiples of worker nodes
  - Smaller the partition size, faster failover and better granularity in load balance, but it incurs more overhead
  - Need a balance
- Backup Tasks
  - In order to cope with a "straggler", the master schedules backup executions of the remaining in-progress tasks

Ack: Slide by Junghoon Kang

Apache Hadoop

- Apache Hadoop has an open-source version of GFS and MapReduce
  - GFS -> HDFS (Hadoop File System)
  - Google MapReduce -> Hadoop MapReduce
- You can download the software and implement your own MapReduce applications

Ack: Slide by Junghoon Kang

Map Reduce Pros and Cons

- MapReduce is good for off-line batch jobs on large data sets
- MapReduce is not good for iterative jobs due to high I/O overhead as each iteration needs to read/write data from/to GFS
- MapReduce is bad for jobs on small datasets and jobs that require low-latency response

Ack: Slide by Junghoon Kang

Spark

See the RDD paper from the course website

What is Spark?

Distributed in-memory large scale data processing engine!

- Not a modified version of Hadoop
- Separate, *fast*, MapReduce-like engine
  - In-memory data storage for very fast iterative queries
  - General execution graphs and powerful optimizations
  - ~~Up to 40x faster than Hadoop~~
  - Up to 100x faster (2-10x on disk)
- Compatible with Hadoop's storage APIs
  - Can read/write to any Hadoop-supported system, including HDFS, HBase, SequenceFiles, etc

Ack: Slide by Prajakta Kalmegh

Applications (Big Data Analysis)

- In-memory analytics & anomaly detection (Conviva)
- Interactive queries on data streams (Quantifind)
- Exploratory log analysis (Foursquare)
- Traffic estimation w/ GPS data (Mobile Millennium)
- Twitter spam classification (Monarch)
- . . .

Ack: Slide by Prajakta Kalmegh

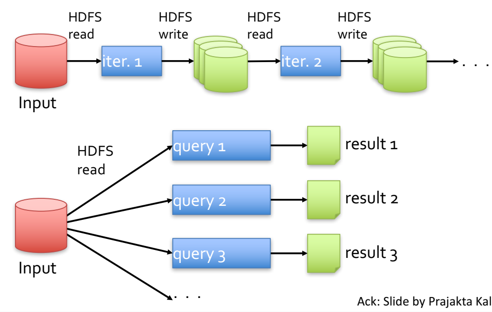
Why a New Programming Model?

- MapReduce greatly simplified big data analysis
- But as soon as it got popular, users wanted more:
  - More complex, multi-stage iterative applications (graph algorithms, machine learning)
  - More interactive ad-hoc queries
  - More real-time online processing
- All three of these apps require fast data sharing across parallel jobs

NOTE: What were the workarounds in MR world? Ysmart[1], Stubby[2], PTF[3], Haloop [4], Twister [5]

Ack: Slide by Prajakta Kalmegh

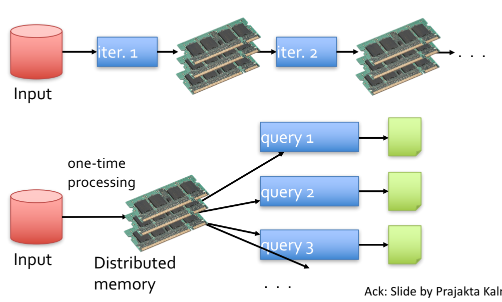
Data Sharing in MapReduce



Ack: Slide by Prajakta Kalmegh

Slow due to replication, serialization, and disk IO

Data Sharing in Spark



Ack: Slide by Prajakta Kalmegh

10-100× faster than network and disk

RDD: Spark Programming Model

- Key idea: *Resilient Distributed Datasets (RDDs)*
  - Distributed collections of objects that can be cached in memory or stored on disk across cluster nodes
  - Manipulated through various parallel operators
  - Automatically rebuilt on failure (How? Use Lineage)

Ack: Slide by Prajakta Kalmegh

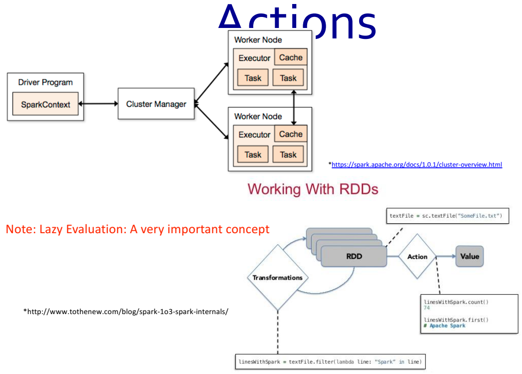
Additional Slides on Spark (Optional Reading)

Ack: The following slides are by Prajakta Kalmegh

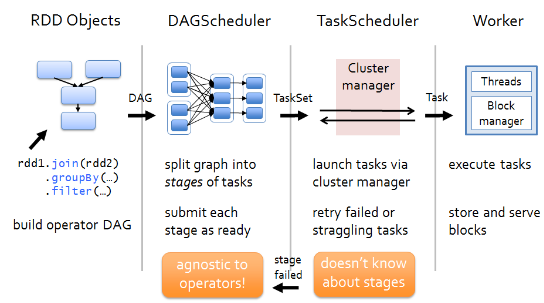
More on RDDs

- **Transformations:** Created through deterministic operations on either
  - data in stable storage or
  - other RDDs
- **Lineage:** RDD has enough information about how it was derived from other datasets
- **Immutable:** RDD is a read-only, partitioned collection of records
  - Checkpointing of RDDs with long lineage chains can be done in the background.
  - Mitigating stragglers: We can use backup tasks to recompute transformations on RDDs
- **Persistence level:** Users can choose a *re-use* storage strategy (caching in memory, storing the RDD only on disk or replicating it across machines; also chose a persistence priority for data spills)
- **Partitioning:** Users can ask that an RDD's elements be partitioned across machines based on a key in each record

RDD Transformations and Actions



DAG of RDDs

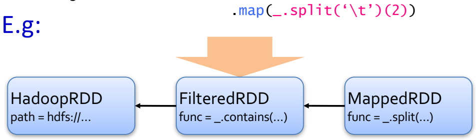


*https://trongkhoanguyenblog.wordpress.com/2014/11/27/understand-rdd-operations-transformations-and-actions/

Fault Tolerance

- RDDs track the series of transformations used to build them (their *lineage*) to recompute lost data
- E.g:

```
messages = textFile(...).filter(_.contains("error"))
                        .map(_.split('\t')(2))
```



Tradeoff:
Low Computation cost (cache more RDDs)
VS High memory cost (not much work for GC)

Representing RDDs

- Graph-based representation. Five components :

Operation	Meaning
partitions()	Return a list of Partition objects
preferredLocations(*p*)	List nodes where partition *p* can be accessed faster due to data locality
dependencies()	Return a list of dependencies
iterator(*p, parentIters*)	Compute the elements of partition *p* given iterators for its parent partitions
partitioner()	Return metadata specifying whether the RDD is hash/range partitioned

Computation function

helps in partitions based optimization

Table 3: Interface used to represent RDDs in Spark.

Representing RDDs (Dependencies)

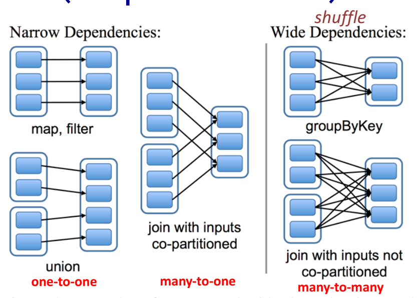


Figure 4: Examples of narrow and wide dependencies. Each box is an RDD, with partitions shown as shaded rectangles.

37

Representing RDDs (An example)

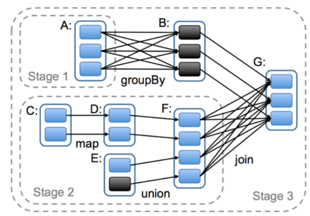


Figure 5: Example of how Spark computes job stages. Boxes with solid outlines are RDDs. Partitions are shaded rectangles, in black if they are already in memory. To run an action on RDD G, we build build stages at wide dependencies and pipeline narrow transformations inside each stage. In this case, stage 1's output RDD is already in RAM, so we run stage 2 and then 3.

38

Advantages of the RDD

Aspect	RDDs	Distr. Shared Mem.
Reads	Coarse- or fine-grained	Fine-grained
Writes	Coarse-grained	Fine-grained
Consistency	Trivial (immutable)	Up to app / runtime
Fault recovery	Fine-grained and low-overhead using lineage	Requires checkpoints and program rollback
Straggler mitigation	Possible using backup tasks	Difficult
Work placement	Automatic based on data locality	Up to app (runtimes aim for transparency)
Behavior if not enough RAM	Similar to existing data flow systems	Poor performance (swapping?)

Table 1: Comparison of RDDs with distributed shared memory.

39

Checkpoint!

- Data Sharing in Spark and Some Applications
- RDD Definition, Model, Representation, Advantages

Other Engine Features: Implementation

- Not covered in details
- Some **Summary**:
  - Spark local vs Spark Standalone vs Spark cluster (Resource sharing handled by Yarn/Mesos)
  - *Job Scheduling:* DAGScheduler vs TaskScheduler (Fair vs FIFO at task granularity)
  - *Memory Management:* serialized in-memory(fastest) VS deserialized in-memory VS on-disk persistent
  - *Support for Checkpointing:* Tradeoff between using lineage for recomputing partitions VS checkpointing partitions on stable storage
  - *Interpreter Integration:* Ship external instances of variables referenced in a closure along with the closure class to worker nodes in order to give them access to these variables