Let’s see how MapReduce works.

MapReduce

- Proposed by Google in research paper
  - Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. OSDI'04: Sixth Symposium on Operating System Design and Implementation, San Francisco, CA, December, 2004
- MapReduce implementations such as Hadoop differ in details, but main principles are the same

Overview

- MapReduce = programming model and associated implementation for processing large data sets
- Programmer essentially just specifies two (sequential) functions: map and reduce
- Program execution is automatically parallelized on large clusters of commodity PCs
  - MapReduce could be implemented on different architectures, but Google proposed it for clusters

Overview

- Clever abstraction that is a good fit for many real-world problems
- Programmer focuses on algorithm itself
- Runtime system takes care of all messy details
  - Partitioning of input data
  - Scheduling program execution
  - Handling machine failures
  - Managing inter-machine communication

Programming Model

- Transforms set of input key-value pairs to set of output values (notice small modification compared to paper)
- Map: (k1, v1) $\rightarrow$ list (k2, v2)
- MapReduce library groups all intermediate pairs with same key together
- Reduce: (k2, list (v2)) $\rightarrow$ list (k3, v3)
  - Usually zero or one output value per group
  - Intermediate values supplied via iterator (to handle lists that do not fit in memory)

Example: Word Count

- Insight: can count each document in parallel, then aggregate counts
- Final aggregation has to happen in Reduce
  - Need count per word, hence use word itself as intermediate key (k2)
  - Intermediate counts are the intermediate values (v2)
- Parallel counting can happen in Map
  - For each document, output set of pairs, each being a word in the document and its frequency of occurrence in the document
  - Alternative: output (word, 1) for each word encountered

Word Count in MapReduce

Count number of occurrences of each word in a document collection:

```
map(String key, String value):
  // key: document name
  // value: document contents
  for each word w in value:
    EmitIntermediate(w, 1);
```

```
reduce(String key, Iterator values):
  // key: a word
  // values: a list of counts
  int result = 0;
  for each v in values:
    result += v;
  Emit(key, result);
```

Almost all the coding needed
(need also MapReduce specification object with names of input and output files, and optional tuning parameters)

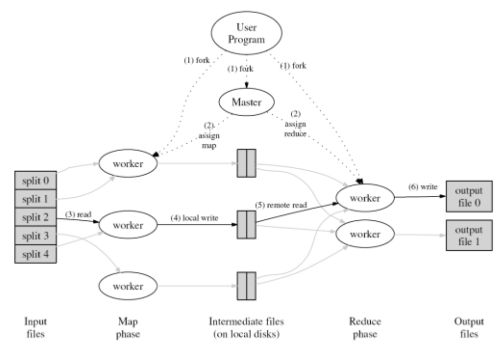
Execution Overview

- Data is stored in files
  - Files are partitioned into smaller splits, typically 64MB
  - Splits are stored (usually also replicated) on different cluster machines
- Master node controls program execution and keeps track of progress
  - Does not participate in data processing
- Some workers will execute the Map function, let's call them mappers
- Some workers will execute the Reduce function, let's call them reducers

Execution Overview

- Master assigns map and reduce tasks to workers, taking data location into account
- Mapper reads an assigned file split and writes intermediate key-value pairs to local disk
- Mapper informs master about result locations, who in turn informs the reducers
- Reducers pull data from appropriate mapper disk location
- After map phase is completed, reducers sort their data by key
- For each key, the Reduce function is executed and output is appended to final output file
- When all reduce tasks are completed, master wakes up user program

Execution Overview



Master Data Structures

- Master keeps track of status of each map and reduce task and who is working on it
  - Idle, in-progress, or completed
- Master stores location and size of output of each completed map task
  - Pushes information incrementally to workers with in-progress reduce tasks

Handling Mapper Failures

- Master pings every worker periodically
- Workers who do not respond in time are marked as failed
- Mapper's in-progress and completed tasks are reset to idle state
  - Can be assigned to other mapper
  - Completed tasks are re-executed because result is stored on mapper's local disk
- Reducers are notified about mapper failure, so that they can read the data from the replacement mapper

Handling Reducer Failures

- Failed reducers identified through ping as well
- Reducer's in-progress tasks are reset to idle state
  - Can be assigned to other reducer
  - No need to restart completed reduce tasks, because result is written to distributed file system

Handling Master Failure

- Failure unlikely, because it is just a single machine
- Can simply abort MapReduce computation
  - Users re-submit aborted jobs when new master process is up
- Alternative: master writes periodic checkpoints of its data structures so that it can be re-started from checkpointed state

Semantics with Failures

- If map and reduce are deterministic, then output is identical to non-faulting sequential execution
  - For non-deterministic operators, different reduce tasks might see output of different map executions
- Relies on atomic commit of map and reduce outputs
  - In-progress task writes output to private temp file
  - Mapper: on completion, send names of all temp files to master (master ignores if task already complete)
  - Reducer: on completion, *atomically* rename temp file to final output file (needs to be supported by distributed file system)

Practical Considerations

- Conserve network bandwidth ("Locality optimization")
  - Schedule map task on machine that already has a copy of the split, or one "nearby"
- Create backup tasks to deal with machines that take unusually long for the last in-progress tasks ("stragglers")

Refinements

- User-defined partitioning functions for reduce tasks
  - Default: assign key K to reduce task *hash(K) mod R*
  - Use *hash(Hostname(urlkey)) mod R* to have URLs from same host in same output file
  - We will see others in future lectures
- Combiner function to reduce mapper output size
  - Pre-aggregation at mapper for reduce functions that are commutative and associative
  - Often (almost) same code as for reduce function

Careful With Combiners

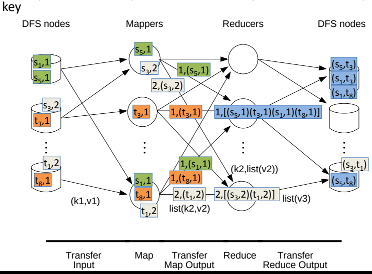
- Consider Word Count, but assume we only want words with count > 10
  - Reducer computes total word count, only outputs if greater than 10
  - Combiner = Reducer? No. Combiner should not filter based on its local count!
- Consider computing average of a set of numbers
  - Reducer should output average
  - Combiner has to output (sum, count) pairs to allow correct computation in reducer

Equi-Join in MapReduce

- Given two data sets S=(s_1,s_2,…) and T=(t_1,t_2,…) of integers, find all pairs (s_i,t_j) where s_i.A=t_j.A
- Can combine the s_i and t_j only in Reduce
  - To ensure that the right tuples end up in the same Reduce invocation, use join attribute A as intermediate key (k2)
- Map needs to output (s.A, s) for each S-tuple s (similar for T-tuples)
  - Also adds a flag indicating if the tuple is from S or T

Equi-Join in MapReduce

- Join condition: S.A=T.A
- Map(s) = (s.A, (s, "S")); Map(t) = (t.A, (t, "T"))
- Reduce computes Cartesian product of set of S-tuples and set of T-tuples with same key



Theta-Joins in MapReduce?

- Equi-join algorithm has problems when data is skewed
- What about non-equi joins?
  - Inequality (S.A<T.A): map just forwards T-tuples, but replicates S-tuples for all larger T.A values as keys
    - Not practical
- Need different approach: discussed in future lecture