Paper Review –3

MapReduce

Title: MapReduce: Simplified Data Processing on Large Clusters

Summary

The paper presented here provides the design, implementation and evaluation of MapReduce which is a programming model and Implementation for processing and generating large data sets in a distributed environment. The paper describes the programming model and the functions *map* and *reduce* that is responsible for the computation and result of a particular problem. Since MapReduce is a programming model, an implementation of this model can be done by any one by understanding the needs and design of the distributed system. Author provides the implementation details of the Google implementation of MapReduce used for their google products. It also covers various Fault tolerance techniques provided within the model and also refinements added on top of the basic functionality. The paper concludes with performance comparison using two famous problems ‘grep’ and ‘sort’ across a large data set and publishes the results even in case of failure.

Good Aspects

A need for parallelization of tasks was a necessity when coming up with the design of MapReduce and Google seems to have approached it in a way that it solves problems for themselves and also in general. A classic and famous implementation of MapReduce is also Hadoop which shows how the programming model is designed to handle any ones needs. MapReduce programming model works on cluster of commodity machines with not too much bandwidth requirements hence makes it feasible.

Review

Following details will cover all aspect of the paper providing crucial and also interesting aspects of the paper that appealed the reader.

Technical Details

- **Programming Model**
  - **Input –** Set of Key/Value Pairs.
  - **Output –** Set of Key/Value Pairs.
  - **Map Function –** Takes an input pair and produces a set of Intermediate Key/Value Pairs.
  - **Reduce Function –** Accepts an Intermediate key 'I' and set of values associated with the Key '**I'** and merges these to produce a reduced set of Value/Values.
  - Reduce function receive input as Iterator to help deal with lists that are too big for the memory.
  - Types
    - Map (k1,v1) -> list(k2,v2)
    - Reduce (k2, list(v2)) ->list(v2)

- **Example**

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

//Iterator will have values from all the documents for each word
reduce (String key, Iterator values):
   //key: a word
   //values: a list of counts
   int result=0;
   for each v in values:
      result+= ParseInt(v);
   Emit(AsString(result);
```

- **Scenarios of Usage**
  - **Distributed Grep –**
    - Map function – Throw a line that matches the pattern.
    - Reduce function – Copy the line to the output.
  - **Count of URL Frequency**
    - Map function – For each URL , output <URL,1>
    - Reduce function – For each URL , add all contents of list

- **Reverse Web Link Graph**
  - Map function – outputs a **Target** url for every **Source** url
  - Reduce function – concatenates list of all **Source** url's for a **Target** url.
- **Term Vector per Host**
  - Map function – outputs a <hostname, term vector> pair for each document. Term is a pair of <word,frequency>.
  - Reduce function – passed all per-document term vectors for a given host and then emits a final <hostname, term vector>.
- **Inverted Index**
  - Map function – outputs a sequence of <word, documentID>
  - Reduce function – combines and outputs <word, documentID list>
- **Distributed Sort**
  - Map function – outputs a <key,record> pair
  - Reduce function – emits all pairs.

- **Implementation**
  - **Google Requirements**
    - Dual Core x86 Processors running Linux with 2-4 Gb of memory.
    - Networking hardware either 100 megabits/second or 1 gigabits/second.
    - A cluster containing 100 to 1000 of machines.
    - Storage by inexpensive IDE Disks.
    - Jobs scheduled using a run time system.
  - **Initial Working**
    - Map invocations are distributed by splitting files into a set of 'M' splits.
    - Each of these split is parallel processed by machines in a cluster.
    - Reduce invocations are distributed by partitioning the intermediate key into 'R' pieces using a partitioning function *hash(key) mod R*.
  - **Execution Sequence**
    - Map Reduce library splits the input files into sizes of 16Mb to 64Mb. It starts many copies of the program on different machines.
    - One of the copies is special and its called **Master**. The rest are called **workers** and are assigned work by the master. Master picks up an idle worker and assigns a map or reduce task.
    - A worker who is assigned a map task reads all the contents of the corresponding input split file and outputs the intermediate key pairs into its memory.

- Periodically all the buffered pairs are flushed to a local disk and partitioned into 'R' Regions. Once this is done, the master is notified of the location in disk of the R Regions.
- The master now assigns the reduce task to an idle worker and once its notified, does RPC to the machine to read the buffered data from the local disk. It sorts all occurrences for each key and may apply external sort depending on size.
- The reduce workers iterates through all the intermediate key value pairs and gives this as input to the reduce function. The output of this reduce function is appended to a final output file.
- Once all the reduce and map workers are done, master wakes up the program and the *MapReduce* function returns to the user.

- **Data Structures**
  - For each map and reduce task, master stores the state {*idle, in-progress or completed*}
  - Master also stores identity of each worker machine
  - Master stores locations and sizes of R intermediate file regions produced by the map task.

- **Fault Tolerance**
  - **Worker Failure**
    - The master pings the worker's periodically to determine if they are still alive.
    - Based on the response it marks the state for each of these workers.
    - Completed map tasks that have failed execution due to various reasons are rescheduled by master on different worker.
    - When a failed task is re executed by any worker, then this is notified to all reduce program workers so that they know where the new location is as all map job outputs are stored in local memory.

  - **Master Failure**
    - The implementation can keep track of states of the data structures present in the master by storing checkpoints.
    - A new master can be assigned with a given checkpoint to restore the work how ever since it's a single machine, the chances of failure is less.
    - In googles implementation, the error is returned to client and the client can re start the job.

  - **Semantics in the Presence of failures**

- Both the user supplied Map and Reduce functions are deterministic.
- The implementation of Map and Reduce is such that, it gives the user the feel of a sequential computation.
- By using the Atomic rename functionality, its guaranteed that every one of the 'R' reduce jobs will produce a unique output file.

- **Network Bandwidth**
  - Network bandwidth is conserved by map jobs that produce output into the local machine disks and into blocks of 64MB each using GFS.
  - GFS distributed the input data initially by storing duplicate of copies across 3 machines. Hence in the case of failure, the other map workers can be scheduled to perform the job rather than split the file again and dispatching over the network.

- **Granularity**
  - Map phase is divided into M pieces and Reduce phase is divided into R pieces.
  - M and R must be greater than number of worker machines.
  - Practical bounds on scheduling decisions is O(M+R)
  - Bounds on memory usage is O(M*R) where each bit is used to track the state of machines
  - M can be chosen between 16MB to 64MB.
- **Back up tasks**
  - One common problem is a 'straggler', a machine that takes time to computer the last job that is map or reduce.
  - This could be of various reasons like bad disk sectors.
  - The problem increases when more tasks are scheduled on that machine and resource contention begins to arise that leads to more delays.
  - A mechanism is used to reduce the problem of stragglers by creating back up jobs on different machines and which every machine reports the result that is Primary execution or back up execution, will be consider the final and the rest are just stopped.
  - In this way, if the straggler is stuck, then the other machines can as well compute the result.

- **Refinements**
  Extra refinements were added, in addition to the basic functionality of Map and Reduce
  - **Partitioning Function**
    - A good enough partitioning function is *hash(key) mod R*

- Al though users can write their own partitioning function where like for example, a host that contains all URLS can be mapped to one R file by using a hash function *hash(hostname(url)) mod R*.

- **Combiner Function**
  - User's have flexibility in providing a combiner function since there can be duplicates of a record like word frequencies.
  - Hence a combiner can be used and is executed on the same machine that performs the map task.
  - The output of a combiner function is written to an intermediate file that will be sent to a reduce task.

- **Input and Output types**
  - A commonly supported format is text, how ever user has flexibility to implement own reader based on the type of file.
  - Such a reader can read data from database or memory.

- **Skipping Bad Records**
  - Some times its possible for a buggy Map Reduce function to crash and this can also occur in a 3rd party library where there is no way to make the change to the code.
  - For such cases, its possible to skip the record by master when it learns about the sequence that caused the crash, so that on failure this sequence will not be executed by any of the workers.
  - When such a crash occurs, a signal installed will invoke the handler and send a UDP packet containing the sequence number back to the master.
  - The master will store this information to avoid re execution on failure.

- **Local Execution**
  - MapReduce library has support for local execution to help in development phase.

- **Status Information**
  - Master exports a web page that contains the status of the jobs.
  - It runs an internal HTTP server to publish this web page.
  - It also contains information of which map and reduce jobs have failed.

- **Performance**
  - Performance here is measured on two programs, one that searches for a pattern in one terabyte of data and the other computation sorts approximately one terabyte of data.

- **Cluster Configuration** – Cluster consisting of 1800 machines and each machine has two 2GHZ Xeon Intel Xeon processors with hyper threading, 4GB of RAM, two 160 GB HDD and Gigabit Network Interface.
- **Grep** – Scans for 10^{10} 100 byte records and the input is split with M= (15000) and the entire output is placed in one file (R=1). The rate gradually picks up and peaks up to 30GB/S when the maximum amount of workers is performing the map function. Once the map jobs are done, the rate slows down and goes to 0. The entire operation took 150 seconds and some initial propagation delay for code duplication.
- **Sort** – Sorts 10^{10} 100 byte records. Three phases where in first phase the Map function provides a 10-byte key for ever line in the file. This is output as an intermediate file to the second phase called shuffling. In Shuffling the data is partitioned based on keys are two replicas are created for reliability and availability. The overall time for sort took 850 seconds including the overhead for locality optimization.

- **Real World Applications of MapReduce at Google**

  - To solve large scale Machine Learning Problems.
  - Clustering problems for Google News and Froogle Products.
  - Extraction of data used to produce reports of the popular queries for example Google Zeitgeist.
  - To perform Large Scale Graph Computations.
  - Extraction of properties of Web Pages like locality to implement localized search.

Conclusion

From the paper we understand that the needs for Google to build a scalable and parallel processing programming model is successful. Its ease of use, features like fault tolerance, locality optimization and load balancing make it quite powerful. Also a large variety of real world problems can be expressed in a map reduce model. It is also widely scalable across clusters of machines.