Goel, Ajay: Nu Id (001897443)

INFO 7250
Engineering Big-Data Systems
Summer Full 2019
Assignment 1

Map Reduce: Simplified Data Processing on Large Clusters

Introduction:

- MapReduce is a programming model and an associated implementation for processing and generating large data sets. A map function that processes a key/value pair to generate a set of intermediate key/value pairs, and a reduce function that merges all intermediate values associated with the same intermediate key.
- Programs written in this functional style are automatically parallelized and executed on a large cluster of commodity machines. Functions:
    - details of partitioning the input data, scheduling the program's execution across a set of machines, handling machine failures, and managing the required inter-machine communication.

Programming Model:

- Set of input key/value pairs and produces a set of output key value pairs. Map takes an input pair and produces a set of intermediate key value pairs. The MapReduce library groups together all intermediate values associated with the same intermediate key I and passes them to the Reduce function.
- The Reduce function accepts an intermediate key I and a set of values for that key. It merges together these values to form a possibly smaller set of values.

Examples for MapReduce:

- Distributed Grep
- Count of URL Access Frequency
- Reverse Web-Link graph
- Term-Vector per Host
- Inverted Index
- Distributed Sort

The implementation of the MapReduce interface are possible. The right choice depends on the environment.

The following sequence of actions occurs when a MapReduce function is called:

- The MapReduce library in the user program splits the input into M pieces of typically 16 megabytes to 64 megabytes per piece.
- One of the copies of the program is special, the master, rest are workers that are assigned work by the master. There are map tasks and reduce tasks to assign. The master picks idle workers and assigns each one a map task or a reduce task.
- A worker who is assigned a map task reads the contents of the corresponding input split. It parses key/value pairs out of the input data and passes each pair to the user defined Map function.
- The buffered pairs are written to local disk, partitioned into R regions by the partitioning function. The locations of these buffered pairs on the local disk are passed back to the master, who is responsible for forwarding these locations to the reduce workers.
- Reduce worker is notified by the master about these locations, it uses remote procedure calls to read the buffered data from the local disks of the map workers. When a reduce worker has read all intermediate

  data, it sorts it by the intermediate keys so that all occurrences of the same key are grouped together.

- The reduce worker iterates over the sorted intermediate data and for each unique intermediate key encountered, it passes the key and the corresponding set of intermediate values to the user's Reduce function. The output of the Reduce function is appended to output for this reduce partition.
- When all map tasks and reduce tasks have been completed, the master wakes up the user program.

Master Data Structures
The master is the conduit through which the location of intermediate regions is propagated from map tasks
to reduce tasks. Therefore, for each completed map task, the master stores the locations and sizes of the R intermediate regions produced by the map task. Updates to this location and size information are received as map tasks are completed. The information is pushed incrementally to workers that have in-progress reduce tasks.

Fault Tolerance:
- **Worker Failure:** The master pings every worker periodically. If no response is received from a worker in a certain amount of time, the master marks the worker as failed.
- **Master Failure:** If the master task dies, a new copy can be started from the last checkpointed state. Current implementation aborts the MapReduce computation if the master fails.
- **Locality:** Network bandwidth is a relatively scarce resource in the computing environment. Conserve network bandwidth by taking advantage of the fact that the input data is stored on the local disks of the machines that make up cluster. When running large MapReduce operations on a significant fraction of the workers in a cluster, most input data is read locally and consumes no network bandwidth.
- **Backup Tasks:** The cluster scheduling system may have scheduled other tasks on the machine, causing it to execute the MapReduce code more slowly due to competition for CPU, memory, local disk, or network bandwidth.

Refinements:
- **Partition Function**
- **Ordering Guarantees**
- **Combiner Function**
- **Input and Output Types**
- **Side – effects**
- **Skipping Bad Records**
- **Local Execution**
- **Status Information**
- **Counters**

Performance: It is done by running it on a large cluster of machines. One computation search through approximately one terabyte of data looking for a particular pattern. The other computation sorts approximately one terabyte of data.\
- **Cluster Configuration**
- **Grep:** The grep program scans through 10^{10} 100-byte records, searching for a relatively rare three-character pattern (the pattern occurs in 92,337 records). The input is split into approximately 64MB pieces (M = 15000), and the entireoutput is placed in one _le (R = 1).
- **Sort:** The sort program sorts 10^{10} 100-byte records. This program is modeled after the Tera Sort benchmark.

Large Scale Indexing:
- The indexing system takes as input a large set of documents that have been retrieved by our crawling system, stored as a set of GFS. The raw contents for these documents are more than 20 terabytes of data. The indexing process runs as a sequence of five to ten MapReduce operations.

- The indexing code is simpler, smaller, and easier to understand, because the code that deals with fault tolerance, distribution and parallelization is hidden within the MapReduce library.
- The performance of the MapReduce library is good enough that we can keep conceptually unrelated computations separate, instead of mixing them together to avoid extra passes over the data.
- The indexing process has become much easier to operate, because most of the problems caused by machine failures, slow machines, and networking hiccups are dealt with automatically by the MapReduce library without operator intervention. It is easy to improve the performance of the indexing process by adding new machines to the indexing cluster.

Many systems have provided restricted programming models and used the restrictions to parallelize the computation automatically. The MapReduce implementation relies on an in-house cluster management system that is responsible for distributing and running user tasks on a large collection of shared machines.

Conclusion:

The MapReduce programming model has been successfully used at Google for many different purposes:

- The model is easy to use, even for programmers without experience with parallel and distributed systems, since it hides the details of parallelization, fault-tolerance, locality optimization and load balancing.
- Large variety of problems are easily expressible as MapReduce computations.
- Third, we have developed an implementation of MapReduce that scales to large clusters of machines comprising thousands of machines.
- The implementation makes client use of these machine resources and therefore is suitable for use on many of the large computational problems encountered at Google.
- The programming model makes it easy to parallelize and distribute computations and to make such computations fault-tolerant. Network bandwidth is a scarce resource. A number of optimizations in system are therefore targeted at reducing the amount of data sent across the network: the locality optimization allows us to read data from local disks and writing a single copy of the intermediate data to local disk saves network bandwidth.
- Redundant execution can be used to reduce the impact of slow machines, and to handle machine failures and data loss.