MapReduce using Hadoop

The problem with single system parallelism is that

1. Dividing the work into equal size pieces is not always easy and obvious
2. Combining results from independent processes introduces overhead
3. Processing capacity is limited to one system

Why Mapreduce?

MapReduce has developed itself to be the fundamental way to process extremely large data. It has a simple programming model, which is a functional model for large scale data processing.

It's main benefits are

1. It exploits a large set of commodity computers
2. Highly distributed
3. High availability

Design

MapReduce mainly consists of 3 pipelines

1. Map
    a. Accepts key, value pairs and emits a key, value pair
    b. (K_{in},V_{in}) -> $list(K_{inter},V_{inter})$
    c. Sorts all key value pairs before sending it to the reducer, done by using combiners and secondary sorting
2. Reduce
    a. Accepts intermediate key value pairs and emits an output key value pair
    b. $(K_{inter}, list(V_{inter}))$ -> $list(K_{out},V_{out})$
    c. Merges all intermediate values associated with the same key.
    d. This runs after all map tasks are finished to accumulate all the data points for the same key globally, and not just for intermediate map tasks.
3. Partition function
    a. Decides to which reducer an intermediate key value pair must go
    b. hash(key) mod R is the usual partition

Example programs

Sorting

1. Map : identity
2. Reduce : identity
3. Partition function : if $k_1 < k_2$ then $p(K_1) < p(K_2)$

Searching

1. Map : if line matches pattern output line
2. Reduce : identity

Word counting

1. Map : Send each word as word,1
2. Reduce : Group repeated words and add their values
3. Partition : Based on A-M and N-Z or some partition to distribute work

Flow of the program

1. User submits the job
2. The input data is arranged in splits and Hadoop provisions a set of MapReduce tasks, which run on these splits, usually one map task per split
3. The tasks return output to the user after the job is run.

The split size is proportional to the amount of parallelism required.

The advantages of a small split size are

1. Large # of splits
2. Increased parallelism
3. Increased load balancing

Disadvantages

1. Overhead of managing splits and map task creation begins to dominate execution time.

Usually a split size equal to the block size is chosen.

This is good because all the data required for the map is on the same node, decreasing inter-node data transfer. If this split size is not equal to the block size, data transfer impacts performance.

This aligns with the principle of “Move compute to Data”, which is desirable when the data is large.

The map outputs are written to the local disk and not the HDFS. This is because the map output is temporary and can be discarded after reduce.

Failure control is done when a node running a map task fails, by automatically running the map task on a different node.

The record boundary problem of having incomplete data is solved by Hadoop by keeping a pointer to the location of the next data block, and if needed, data can be read till the correct byte offset to complete that incomplete data and handling the overcount occurring as a result.

Data locality is maintained for Mappers in Hadoop where

1. Map tasks run on node where the data resides
2. If all nodes containing the data are busy, use an available node on the same rack to avoid long network transfer
3. If no such nodes are available, use node on another rack, causing inter-rack transfer.

However, reduce tasks don’t have the advantage of locality so the sorted mapper outputs need to be transferred across the network after merging. The output is stored on the HDFS for reliability, one on the local node and other replicas on off-rack nodes, consuming network bandwidth.

Working of MapReduce

A high level working of an MR job is given as

1. Pre-loaded local input data is sent to a mapping process.
2. Mapping process sends mapped data to a shuffler, that shuffles the values among the reducer nodes
3. Reducer processes generate outputs and store the outputs locally.

A closer look at the map task gives

1. Setup map task
2. Run map() for each record
3. Store map output in memory buffer
    a. Buffer size is defaulted to 100 MB. In case of overflow, the output is spilled to disk
    b. Spill file to buffer is sorted before writing
    c. Combiner (if exists) runs after sort

4. Partition keys to different files
5. Move map outputs to reducers

A combiner is used to combine multiple map outputs before doing a reduce, which does an internal reduce job local to the mapper. The combiner thus runs between the mapper and the shuffler, usually in the mapper itself.

Shuffle and Sort

The process by which the map output is sorted and sent to the reducer is a shuffle.

Each map task has a circular memory buffer that it writes outputs to. The buffer is 100 MB by default. When the contents of the buffer reaches a certain threshold, a background thread starts spilling the contents to disk. If the buffer is filled, the map blocks until the spill is completed.

Before writing to disk, the thread first divides the data into partitions corresponding to the reducers that they will be ultimately sent to. Within each partition, the background thread performs an in-memory sort by key, and the combiner runs on the output of this sort.

Each time the buffer reaches a spill threshold, a new spill file is created. Once the map is done, the spill files are merged into a single partitioned and sorted output file. Combiners may be repeatedly run on spill files, without affecting the final result.

As many map tasks can take time to finish, the reducer starts copying its partition of the files from the mapper. (The outputs from the mapper are deleted only upon job completion). As the copies accumulate on disk, a background thread starts merging them to save time. Once all the map outputs are copied, a sort phase or merge phase occurs, merging the map outputs.

The number of spills need to be minimised for optimality. Optimality can also be achieved if the intermediate data on the reducer side can exist in memory, reducing the size of the reducer.

Matrix Vector Multiplication using MR

Pages in WWW are represented as a directed graph, usually stored as a sparse matrix on HDFS. Each non-zero entry is a separate record, in the format <row_number, col_number, value>.

To multiply a nxn matrix M with an n element vector v

1. v fits in memory:
    a. v is shared by all mappers, where each mapper outputs (i, $m_{ij}v_j$)
    b. reducer sums it.
2. v does not fit into main memory
    a. Partition M and v into stripes, M vertically and v horizontally
    b. Use the same MR algorithm defined above for each splice

PageRank as an Eigenvector problem using MR

Assume each page has

1. an importance I
2. n links to other pages

And the page distributes its importance I among the n links equally.

First, we build a graph of links.

Assume if a node had n outgoing links, users are equally likely to take any link, so importance is divided among all links.

Let A be the importance matrix of all links between all the pages. Let x denote the importance of each page as a single column matrix. Here, x is an eigenvector of A, and we solve to find x as the pagerank of all pages.

To solve this, we can use the iterative approach

1. i = number of times we have looped
2. x_i = value of x on i^{th} iteration
3. Calculate $Ax_i - x_i$ as the error term
4. Derive x_{i+1} as Ax_i
5. Loop over 3 and 4 until the error term is small.