The Map Reduce Framework

CompSci 590.03
Instructor: Ashwin Machanavajjhala
This Class

• Map Reduce Programming Framework

• Map Reduce System Implementation
MAP REDUCE FRAMEWORK
Map-Reduce

map \ (k1,v1) \ \rightarrow \ list(k2,v2); 
reduce \ (k2,\ list(v2)) \ \rightarrow \ list(k3,v3).
Running Example: Word Count

• Input: A set of documents D, each containing a list of words.

• Output: \(<w, c>\), where c is the number of times word w appears across all documents.
Map Task

- Divides the large file into small chunks.
- One *mapper* is assigned to each chunk (and chunks are typically distributed across machines).

- A UDF (user defined function) is applied to each (key, value) pair in the chunk.
- The UDF creates zero, one or more new (key, value) pairs for every input record.
Word Count

- `<docid, {list of words}> → {list of <word, 1>}

The mapper takes a document d and creates n key value pairs, one for each word in the document.

The output key is the word

The output value is 1 (count of each appearance of a word)
Reduce Task

- A reduce task aggregates the keys from different mappers.

- A *reducer* takes all the key value pairs sharing the same key and applies a reduce function to the set of values to generate one output key value pair.

- Shuffle Phase:Reducers are distributed across many machines by hashing key values to different machines.
Word Count

- \(<\text{word}, \{\text{list of 1s}\}> \rightarrow \langle \text{word}, \text{count} \rangle\)

*In this case, the reducer just adds up the count values in each of the tuples with the same key.*
Combiner

• For certain types of reduce functions (commutative and associative), one can decrease the communication cost by running the reduce function within the mappers.
  – SUM, COUNT, MAX, MIN ...

• `<docid, {list of words}> → <word, c>`

  where c is the number of times the word appears in the mapper.
Distributed Grep

• Map:
  <lineid, string> → <lineid, string>  //if string matches pattern

• Reduce: Identity function.
2.3.9 Matrix Multiplication

If $M$ is a matrix with element $m_{ij}$ in row $i$ and column $j$, and $N$ is a matrix with element $n_{jk}$ in row $j$ and column $k$, then the product $P = MN$ is the matrix $P$ with element $p_{ik}$ in row $i$ and column $k$, where

$$p_{ik} = \sum_j m_{ij} n_{jk}$$

It is required that the number of columns of $M$ equals the number of rows of $N$, so these moves over $j$ make sense.

We can think of a matrix as a relation with three attributes: the row number, the column number, and the value in that row and column. Thus, we could view matrix $M$ as a relation $M(I,J,V)$, with tuples $(i, j, m_{ij})$, and we could view matrix $N$ as a relation $N(J, K, W)$, with tuples $(j, k, n_{jk})$. As large matrices are often sparse (mostly 0's), and since we can omit the tuples from a matrix element that are 0, this relational representation is often a very good one for a large matrix. However, it is possible that $i$, $j$, and $k$ are implicit in the position of a matrix element in the file that represents it, rather than written explicitly with the element itself. In that case, the Map function will have to obey designed to construct the $I$, $J$, and $K$ components of tuples from the position of the data.

The product $MN$ is almost a natural join followed by grouping and aggregation. That is, the natural join of $M(I,J,V)$ and $N(J, K, W)$, having only attribute $J$ in common, would produce tuples $(i, j, k, v, w)$ from a tuple $(i, j, v)$ in $M$ and tuple $(j, k, w)$ in $N$. This five-component tuple represents the pair of matrix elements $(m_{ij}, n_{jk})$. What we want instead is the product of these elements, that is, the four-component tuple $(i, j, k, v \times w)$, because that represents the product $m_{ij}n_{jk}$.

Once we have this relation as the result of one map-reduce operation, we can perform grouping and aggregation, with $I$ and $K$ as the grouping attributes and the sum of $V \times W$ as the aggregation. That is, we can implement matrix multiplication as the cascade of two map-reduce operations, as follows. First:

The Map Function: For each matrix element $m_{ij}$, produce the key-value pair $(j, (M,i,m_{ij}))$. Likewise, for each matrix element $n_{jk}$, produce the key-value pair $(j, (N,k,n_{jk}))$. Note that $M$ and $N$ in the values are not the matrices themselves. Rather they are names of the matrices or (as we mentioned for the similar Map function used for natural join) better, a bit indicating whether the element comes from $M$ or $N$.

The Reduce Function: For each key $j$, examine its list of associated values. For each value that comes from $M$, say $(M,i,m_{ij})$, and each value that comes from $N$, say $(N,k,n_{jk})$, the result.

$\sum_j m_{ij} n_{jk}$
Matrix Multiplication

- **Map:**

  \[
  \langle (i,j), m_{ij} \rangle \rightarrow \langle j, (M, i, m_{ij}) \rangle \\
  \langle (j,k), n_{jk} \rangle \rightarrow \langle j, (N, k, n_{jk}) \rangle
  \]

- **Reduce:**

  \[
  \langle j, \{(M, l, m_{ij}), (N, k, n_{jk}) \ldots\} \rangle \rightarrow \{ \langle (i,k), \sum m_{ij}n_{jk} \rangle \}
  \]
MAP REDUCE SYSTEM
Inverted Index: The map function parses each document, and emits a sequence of \( \langle \text{word}, \text{document ID} \rangle \) pairs. The reduce function accepts all pairs for a given word, sorts the corresponding document IDs and emits a \( \langle \text{word}, \text{list (document ID)} \rangle \) pair. The set of all output pairs forms a simple inverted index. It is easy to augment this computation to keep track of word positions.

Distributed Sort: The map function extracts the key from each record, and emits a \( \langle \text{key}, \text{record} \rangle \) pair. The reduce function emits all pairs unchanged. This computation depends on the partitioning facilities described in Section 4.1 and the ordering properties described in Section 4.2.

3 Implementation
Many different implementations of the MapReduce interface are possible. The right choice depends on the environment. For example, one implementation may be suitable for a small shared-memory machine, another for a large NUMA multi-processor, and yet another for an even larger collection of networked machines.

This section describes an implementation targeted to the computing environment in wide use at Google: large clusters of commodity PCs connected together with switched Ethernet [4]. In our environment:

1. Machines are typically dual-processor x86 processors running Linux, with 2-4 GB of memory per machine.
2. Commodity networking hardware is used – typically either 100 megabits/second or 1 gigabit/second at the machine level, but averaging considerably less in overall bisection bandwidth.
3. A cluster consists of hundreds or thousands of machines, and therefore machine failures are common.
4. Storage is provided by inexpensive IDE disks attached directly to individual machines. A distributed file system [8] developed in-house is used to manage the data stored on these disks. The file system uses replication to provide availability and reliability on top of unreliable hardware.
5. Users submit jobs to a scheduling system. Each job consists of a set of tasks, and is mapped by the scheduler to a set of available machines within a cluster.

3.1 Execution Overview
The Map invocations are distributed across multiple machines by automatically partitioning the input data.
System

• Workers/machines are typically commodity hardware
  – Arranged on racks
  – Connected by gigabit ethernets

• Storage is inexpensive hard disks

• Since there are thousands of machines in a cluster, failures are to be expected.
Fault Tolerance

• Map reduce runs over a distributed file system (GFS or HDFS) which ensure availability by replicating the data (e.g., 3x).

• Master (or job tracker) pings each worker periodically.

• If a mapper or a reducer fails midway, then the task is restarted.

• Map (or reduce) phase waits for all the mappers (or reducers) to complete successfully.
Map Execution

• Data is divided into M splits, and one work is assigned to each split

• Worker applies map function to each record in the split.

• Resulting key-value pairs are stored into disk divided into R partitions.
Reduce Execution

• Keys output by the map phase are divided into R partitions using a *partition function*
  – E.g., hash(key) mod R

• The $i^{th}$ reduce worker reads the $i^{th}$ partition output by each map using remote procedure calls

• Data is sorted based on the keys so that all occurrences of the same key are close to each other.

• Reducer iterates over the sorted data and passes all records from the same key to the reduce function.