MapReduce

Daniele Licari

November 20, 2010

based on Dr. Nicola Tonellotto's lesson

A typical one application model takes an input to processing an output. what happen if a function should process a very large input to produce a very large output? In this case the function could be a bottleneck.

Divide & Conquer model offers a common approach to solve these kind of problems. This model is based on breaking one large problem into several smaller problems easier; the solutions to the sub-problems are then combined to give a solution to the original problem. However there are a lot of problems in this model, e.g:

- How do we split the input?
- How do we distribute the input splits to application? (the application can be running on several resources, e.g. GRID, cluster, and the data could be partitioned on the machines)
- How do we collect the output splits?
- How do we aggregate the output?
- How do we coordinate the work?
- What if input splits > num workers?
- What if workers need to share input/output splits?
- What if a worker dies?
- What if we have a new input?

How do we set the workers number? How do we between workers share data? How is program coordinated? How is program synchronized? How do we implement all these stuff? How do we solve these problems? the answer is MapReduce. MapReduce is a programming model and an associated implementation for processing and generating large data sets.

The big ideas behind MapReduce are:

Scale out, not up. A large number of commodity low-end servers is preferred over a small number of high-end servers. The costs of high-end servers do not scale linearly with own efficiency. E.g. The article "Scaling Up vs. Scaling Out: Hidden Costs"[1] provides a comparison a Server "HP ProLiant DL785 G5"(32 CPU, RAM 512 GB, Disk 4TB) with cost closer to $100,000[2] with 83 low-end servers(each with 2.83 GHz quad-core CPU , RAM 8 GB, Disk 2x500 GB SATA) assuming a fixed spend of $100,000

	Scaling Up	Scaling Out
Cores	32	332
RAM	512 GB	664 GB
Disk	4 TB	41.5 TB

On the other hand, we have power consumption problems and rack space isn't free.

Move processing to the data. When nodes need to larger data volumes(e.g. hundreds of GB) the network is a bottleneck. MapReduce assumes an architecture where processors and storage are co-located, so data access is fast since it is local(data locality).

Process data sequentially, avoid random access. Datasets are too large to fit in memory and must be held on disk. As a result, it is desirable to avoid random data access, and instead organize computations so that data is processed sequentially, so we have low seek time.

Seamless scalability. Algorithms that should be used must be scalable in terms of data and resources.

Right level of abstraction, hide implementation details from applications development. MapReduce operates only at the higher level: the programmer thinks in tems of function of key and value pair, and the data flow is implicit. (if you want to hurt yourself there are the MPI API)

MapReduce PROGRAMMING MODEL.

Before showing MapReduce Paradigm, we see a example of typical data problem that we can resolve with MapReduce.

[1]http://www.codinghorror.com/blog/2009/06/scaling-up-vs-scaling-out-hidden-costs.html

[2]on date of article.

E.g. Assuming we have a text file uncompressed size of 1 TB, which must be processed by a function that generates as output the number of occurrences of each word in the input file. The output will be defined by a set of key-value pairs (binary-binary string string), each pair will contain, respectively, word and an associated count of occurrence. The binary string to encode any kind of data type.

MapReduce Paradigm is divided into rounds, each round has three phases:

1. Map: maps input key/value pairs to a set of intermediate key/value pairs. Maps are the individual tasks that transform input records into intermediate records. The transformed intermediate records do not need to be of the same type as the input records. A given input pair may map to zero or many output pairs.

2. Shuffle and Sort (occur simultaneously): while map-outputs are being fetched they are merged. The fetched map output pairs are merged constructing pairs of (key; list(values)) based on the same key. The newly structured pairs of key/list are propagated to the user defined reduce function

3. Reduce: reduces a set of intermediate values which share a key to a smaller set of values.

In a nutshell,

Map: ∀(k,v) produces a new key-value pair (k',v' in a other semantic domain ,intermediate results).

Shuffle and Sort: ∀(k',v') aggregate in a multiset U.

Reduce: to reduce U and returns a new multiset U' (new key/value pair in other semantic domain, final result).

Applications typically implement the Mapper and Reducer interfaces to provide the map and reduce methods. These form the core of the job.
method Map(key k, value v)→EMIT(key k',value v')
method Reduce(key k, value v)→Emit(key k',value [v',v_2',v_3'...])
All values with the same key are reduced together

In the example above, where we have a very large collection of input files and we want to know how many occurrences there are for each term in the documents. In this case, what is key field? the document is a string then a typical key could be a document identifiers. Below a pseudo-code for the word count algorithm in MapReduce.

```
1: class Mapper
2:  method Map(docid a, doc d)
3:   for all term t ϵ doc d do
4:    Emit(term t, count 1)
```

The mapper emits an intermediate key-value pair for each word in a document.

```
1: class Reducer
2:  method Reduce(term t, counts [$c_1$,$c_2$, ...])
3:   sum ← 0
4:   for all count c $\epsilon$ counts [$c_1$,$c_2$, ...] do
5:    sum ← sum + c
6:   Emit(term t, count sum)
```

The reducer sums up all counts for each word.

Formal definition[3]

Definition 1. A mapper is a (possibly randomized) function that takes as input one ordered (*key; value*) pair of binary strings. As output the mapper produces a nite multiset of new (*key; value*) pairs. It is important that the mapper operates on one (*key; value*) pair at a time.

Definition 2. A reducer is a (possibly randomized) function that takes as input a binary string k which is the key, and a sequence of values (v1, v2, ...) which are also binary strings. As output, the reducer produces a multiset of pairs of binary strings (k; $v_{\mathrm{k};1}$); (k;$v_{\mathrm{k};2}$); (k; $v_{\mathrm{k};3}$); ... The key in the output tuples is identical to the key in the input tuple.

A map reduce program consists of a sequence (μ_1,ρ_1,μ_2,ρ_2,..., μ_i,ρ_i) of mappers and reducers.

The input is a multiset of (*key; value*) pairs denoted by U_0.

To execute the program on input U_0:

For r = 1, 2,...,R do:

1. Execute Map: Feed each pair (k; v) in U_{r-1} to mapper μ_r, and run it. The mapper will generate a sequence of tuples, (k_1; v_1); (k_2; v_2),... Let $U_r^{'}$ be the multiset of (key; value) pairs output by μ_r, that is, $U_r^{'}$= $\bigcup_{(k;v)\epsilon U_{r-1}} \mu_r(k;\ v)$.

2. Shuffle: For each k, let $V_{k;v}$ be the multiset of values v_i such that $(k;\ v_i)\epsilon U_r^{'}$. The underlying MapReduce implementation constructs the multisets $V_{k;v}$ from $U_r^{'}$.

[3]H. Karlo, S. Suriy, S. Vassilvitskiiz, “A Model of Computation for MapReduce”, http://www.siam.org/proceedings/soda/2010/SODA10_076_karloffh.pdf

3. Execute Reduce: For each k, feed k and some arbitrary permutation of $(k;v)$ to a separate instance of reducer ρ_r, and run it. The reducer will generate a sequence of tuples $(k_1;\ v'_1)$; $(k_2;\ v'_2)$ ... Let U_r be the multiset of (*key; value*) pairs output by ρ_r, that is, $U_r = \bigcup_k \rho_r(k, V_{k;v})$

MapReduce Example

Consider the following example, we want to compute the k-th frequency moment of a large data (multi)-set.

Let x be the input string of length n, and denote by xi the ith symbol in x. We can represent the input x as a sequence of n pairs(i, x_i) (key- value pairs).

1. Thus we can defined the first map as follows:$\mu_1(i, x_i) = (x_i, i)$. Every tuple (eg, $(1, x_1)$,$(2, x_2)$...) is mapped to a pair with the symbol as the key, and the position as the value.

2. After the aggregation by the key, we can proceed to collapse that list into a single number and we can defined the first reducer as follow: $\rho_1(x_i,[v,v_2,v_3,..,v_m])=(x_i, m^k)$.

3. Now we want to sum the number of remaining pairs, the second map as follows: $\mu_2(x_i, v) = (*, v)$.

4. Finally all of the pairs now have the same key, thus they will all be mapped to the same reducer: e.g. for sum ρ_1(*,$[v,v_2,v_3,..,v_m])=(*, m^k)$, with m$^k=\sum_i v_i$.

References

[1] N. Tonellotto, *"Map Reduce"*, Intenal Slides, 2010.

[2] J. Lin, C. Dyer, *"Data-Intensive Text Processing with MapReduce"*, Ed. Graeme Hirst. Morgan and Claypool Publishers. 2010.

[3] J. Dean, S. Ghemawat, "MapReduce: Simplied Data Processing on Large Clusters", Proceedings of the 6th conference on Symposium on Opearting Systems, 2010.

[4] Apache Hadoop project, *"Map/Reduce Tutorial"*, Hadoop project website , november 2010.

[5] J Atwood, "*Scaling Up vs. Scaling Out: Hidden Costs*", Coding Horror blog website, Jun 2010.

[6] H, Karloff, S. Suri, S. Vassilvitskii, "*A model of Computation for. MapReduce*". Symposium on Discrete Algorithms (SODA), 2010