Modifications to MapReduce

Evguenia (Elmi) Eflov

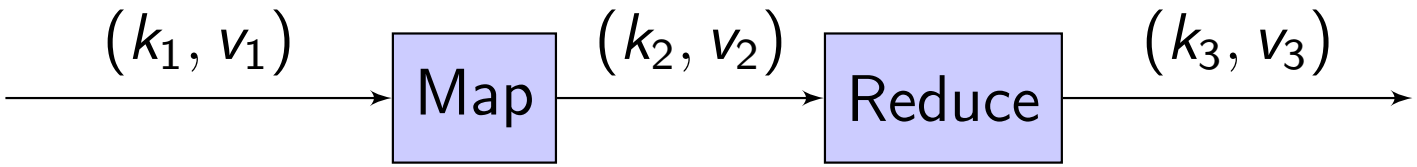
March 29, 2010

Presentation Outline

MapReduce

- MapReduce is a framework that allows its user to implement tasks that fit into the MapReduce model fast, by specifying two functions, Map and Reduce
- MapReduce framework handles distributing work required between machines in the cluster, communication between nodes, scheduling and error handling
- Input data is accepted in the form of files, though input from other sources, for example, a database, can be consumed

MapReduce Details



- *Map* maps each input (k_1, v_1) pair to 0 or more (k_2, v_2) pairs
- *Reduce* processes all values v_2 for a given key k_2 to produce 0 or more output (k_3, v_3) pairs

Optional Elements of MapReduce

In addition to the required functions, *Map* and *Reduce*, MapReduce allows the user to specify the following optional functions

- *Combiner* function to process output of *Map* locally, before transferring it to the other machines - in most cases, *Combiner* is similar to *Reduce* in effect
- *Partitioner* function that determines how the output of *Map* is partitioned for processing by *Reduce*

Problem

- Processing done by *Map* function can be expensive and generate a large amount of intermediate data
- There may be two or more tasks that require output of the same *Map* function, but processed by different *Reduce* functions that operate on the same key, or the key of one is a subset of the key of the other, for example k_1 and k_1, k_2
- Such sets of tasks could benefit from ability to share and reuse common parts of the data and processing

What is Currently Possible with MapReduce?

There are three possible solutions to the problem using existing MapReduce implementations

- Implementing each task as a separate MapReduce task
- Splitting the tasks into a common processing step and task-specific processing steps, and executing each such step as a separate MapReduce task
- Implementing a *Reduce* function that combines both (or all of the) *Reduce* functions required for the tasks

Separate Tasks Approach

This is not the optimal approach if

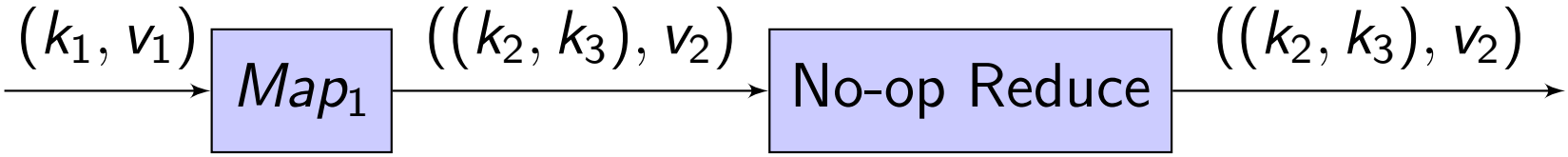
- Common processing required by the tasks is time consuming
- Intermediate results generated by *Map* are large
- There is a large number of unique keys in the *Map* output

Common Processing as a Separate Step Approach

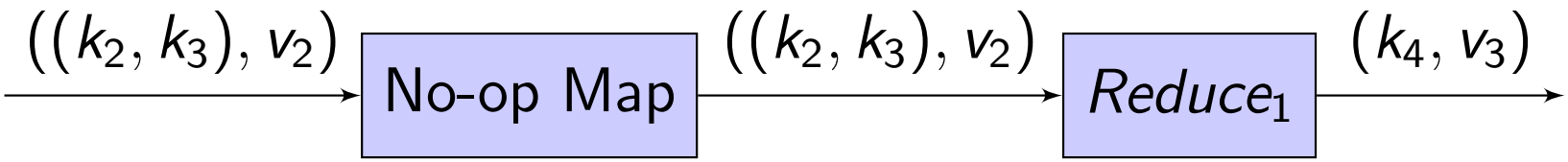
- This approach requires a separate *Map* and *Reduce* implementation for the common processing
- The result of the first step is sorted on the maximal key of all the *Reduce* functions
- Subsequent steps accept output of the first step as input, using either no-op *Map* functions or *Map* that move part of the key into the value
- *Reduce* functions of the subsequent steps perform the required processing

Example of Separating Common Processing

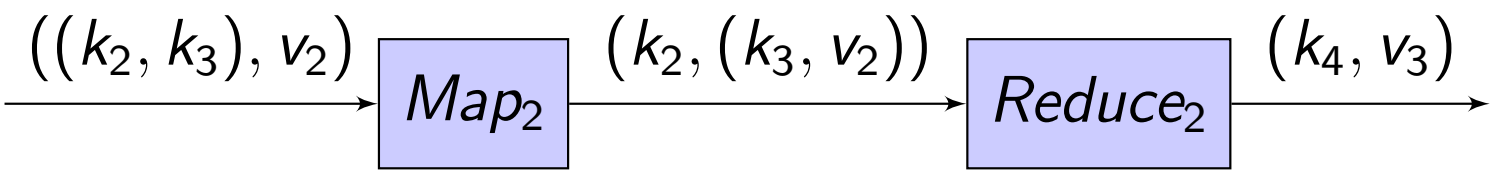
- Common first step



- $Reduce_1$



- $Reduce_2$



Notes About the Example

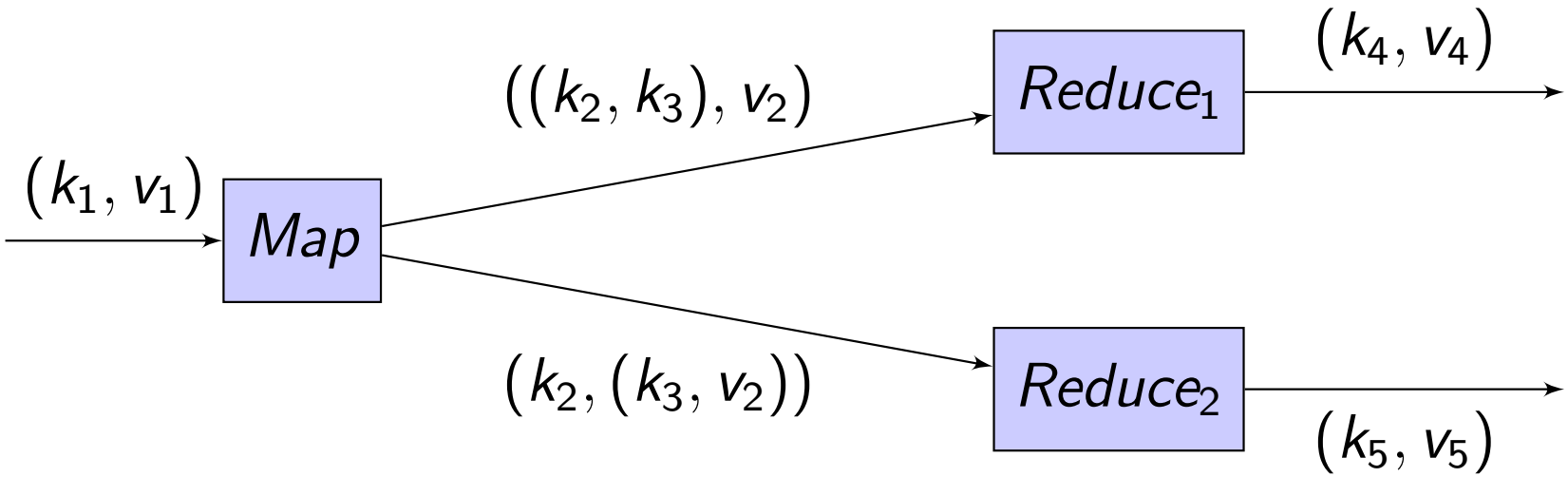
- Note that after the common step the result is sorted on (k_2, k_3), therefore, neither of the following steps requires additional sorting of the inputs (though partitioning is still require)
- If the output size of the common step is large, reading and writing of intermediate data for the *Reduce* steps may constitute a significant portion of execution time for the process
- This approach might perform better than the previous approach if initial processing is time consuming, or intermediate results contain a large number of unique keys

Combining Reduce Functions

- This approach avoids repetition of any part of the processing
- Combined *Reduce* function is required to do its own sorting and partitioning of data for a key if different keys were required for the original *Reduce* functions, since partitioning by the minimal shared key will be provided by MapReduce
- Output generated by this approach has to combine the results of all the *Reduce* functions into a single record - this may not always be convenient
- Output then needs to be processed to separate the results of different *Reduce* functions - this may not always be convenient

Extension to MapReduce

Another way to handle the motivating problem is to allow the same output of *Map* function to be processed by more than one *Reduce* (and possibly *Combiner*) function



Details

- The idea of this proposal is to take advantage of the best part of the Combined Reducer approach while avoiding its disadvantages
- This approach requires modifying MapReduce as follows:
  - Enable the user to specify the subset of common key a *Reduce* function should use as a key (if different from the complete key)
  - Enable the system to run two different *Reduce* tasks on the same intermediate input file
  - Enable the system to produce two output file sets in parallel

Implementation

- Amazon EC2 Linux-based instances with Hadoop
- Freebase data dump will be used as sample data, possibly followed by Wikipedia data dump - both are public data sets available on AWS
- Modifications will be made to the MapReduce code to implement the proposed changes

Future Work

The project idea can be generalized to accommodate not only multiple *Reduce* tasks, but also to use multiple *Map* tasks to supply data for a single *Reduce* task.
This can be useful if data from multiple sources - web pages, news feeds, databases, plain text files - needs to be aggregated and/or processed in the same way to generate a single result set

Questions