MapReduce Frameworks: Comparing Hadoop and HPCC

Work in Progress

Fabian Fier, Eva Höfer, Johann-Christoph Freytag

Humboldt-Universität zu Berlin, Institut für Informatik,
Unter den Linden 6, 10099 Berlin, Germany
{fabian.fier,eva.hoefer,freytag}@informatik.hu-berlin.de

Abstract. MapReduce and Hadoop are often used synonymously. For optimal runtime performance, Hadoop users have to consider various implementation details and configuration parameters. When conducting performance experiments with Hadoop on different algorithms, it is hard to choose a set of such implementation optimizations and configuration options which is fair to all algorithms. By fair we mean default configurations and automatic optimizations provided by the execution system which ideally do not require manual intervention. HPCC is a promising alternative open source implementation of MapReduce. We show that HPCC provides sensible default configuration values allowing for fairer experimental comparisons. On the other hand, we show that HPCC users still have to consider implementing optimizations known from Hadoop.

1 Introduction

In our research, we use MapReduce and its implementation Hadoop to experimentally compare the runtime of various scalable algorithms. Along these experiments, we identified practical issues that make a fair comparison and experimental reproducability hard. Hadoop offers many configuration parameters that influence the runtimes of programs such as the number of Reducers or whether data compression is to be used between Map and Reduce. Furthermore, there are numerous possible optimizations in Hadoop programs, such as custom datatypes and very efficient byte-wise comparators. It is possible to "tweak" every implementation with a certain set of options and implementation optimizations. The same set of options and optimizations can lead to poor execution times for the implementation of different algorithms [1]. The literature barely discusses these configuration parameters and implementation details, although they are crucial for the validity of experimental results.

Another promising open source MapReduce implementation is HPCC (High Performance Computing Cluster) from LexisNexis [2]. Unlike Hadoop, HPCC hides many configuration details from the user. We are interested in how HPCC might replace Hadoop in our context and whether it allows for a fairer comparison when implementing MapReduce algorithms in it. As a running example,

we describe a common MapReduce-based textual similarity join algorithm. We introduce our implementation of this algorithm in Hadoop and its pitfalls concerning system configuration and code details. We compare this implementation to our corresponding implementation in HPCC and discuss our findings.

The paper is structured as follows. Section 2 describes HPCC and ECL. Section 3 contains the textual similarity join problem and the MapReduce-based approach we use as a running example. Section 4 introduces and compares our implementations of the algorithm in Hadoop and HPCC. The last section sums up our findings and gives an outlook on future research on this topic.

2 HPCC and ECL

HPCC is an open source parallel distributed system for compute- and data-intensive computations [2]. It contains a distributed file system. The main user interface of HPCC is ECL (Enterprise Control Language). ECL follows a dataflow-oriented programming paradigm and has declarative components. The following example code[1] computes a word count:

Define record structure "WordLayout" consisting of string "word":

```
WordLayout := RECORD
  STRING word;
END;
```

Read given dataset into the variable "wordsDS", apply WordLayout:

```
wordsDS := DATASET([{'HPCC'}, .., {'ANALYTICS'} ], WordLayout);
```

Define record structure "WordCountLayout" consisting of "word" and "count". Note the count is defined by COUNT(GROUP) which implies that this record structure is to be applied to a grouped dataset:

```
WordCountLayout := RECORD
  wordsDS.word;
  wordCount := COUNT(GROUP);
END;
```

Apply WordCountLayout on dataset wordsDS and group by "word":

```
wordCountTable := TABLE(wordsDS, WordCountLayout, word);
```

ECL allows to incorporate user-defined first-order functions written in C++ or Java [2]. These functions can be called from ECL functions. The semantics of the Map operator is represented in the ECL function PROJECT. It applies a user-defined function to each record in a given dataset. Reduce semantics can be emulated by partitioning and distributing the data with DISTRIBUTE, sorting it locally with SORT, and running a user-defined function on each group with ROLLUP. Although HPCC is not originally designed for the MapReduce programming paradigm, it is straightforward to adapt a MapReduce program to ECL. Thus, we regard HPCC as an alternative implementation of Hadoop.

[1] Adapted from https://aws.hpccsystems.com/aws/code-samples/

3 Textual Similarity Join

This section describes the textual similarity join problem and outlines the algorithmic approach from Vernica et al. [3] to compute it. We subsequently use this algorithm as an example to compare Hadoop to HPCC.

The textual all-pairs similarity join is a common operation that detects similar pairs of objects. Objects can either be strings, sets, or multisets. The similarity is defined by similarity functions such as Cosine or Jaccard similarity. Applications of this join are near-duplicate removal, document clustering, or plagiarism detection. Without loss of generality, we assume a self-join on sets.

Definition 1 (Similarity Join). *Given a collection of sets S, a similarity function sim, and a user-defined threshold δ, a similarity join finds all pairs with a similarity above the threshold:* $\{\langle s_1, s_2\rangle | sim(s_1, s_2) \geq \delta, s_1 \in S, s_2 \in S, s_1 \neq s_2\}$.

A naive approach of computing this join is to compare each possible pair of objects. Due to its quadratic complexity, it is not feasible even for small datasets. More advanced approaches use a filter-and-verification framework. The framework consists of two steps. The first step computes candidate pairs, which are a superset of the result set. Due to the use of filters, the candidate set is much smaller than the cross product (assuming that a majority of pairs of objects of S is not similar). The second step computes the actual similarity for each candidate pair to verify if its similarity is above δ.

A prominent filter-and-verification MapReduce-based algorithm for set similarity joins is the VernicaJoin [3]. The main filtering idea is to only compare short prefixes of two objects to generate candidate pairs. Given a similarity function, a threshold δ and an object length $|s|$, we can compute a prefix length. It can be shown that two objects can only be similar if they have an overlap of at least 1 in their prefixes. One optimization is to sort the words in the objects by their global frequency in ascending order. This assures that the prefixes only contain the least frequent words which reduces the number of candidate pairs.

For our experimental comparison of similarity join algorithms, we adapted VernicaJoin to use already integer-tokenized input (instead of raw string input) and to output ID pairs of similar objects (instead of string pairs). These changes enable us to compare this algorithm to others. In the following, we describe our implementations of this adapted algorithm.

4 Comparison of Implementations

In this section, we introduce our implementations of the previously described algorithm in Hadoop and HPCC. We discuss the most runtime-relevant details concerning implementation and configuration. Due to space restrictions, we refer to the upcoming full version of this paper for experimental results.

The implementations consist of three steps. In the first step, we compute the global token frequency. In the second step, we sort the tokens in each object by this frequency and replicate each object for each token in its prefix. We group

all objects by their prefix tokens and verify for each pair in this group if it meets the threshold δ. The third step removes duplicates.

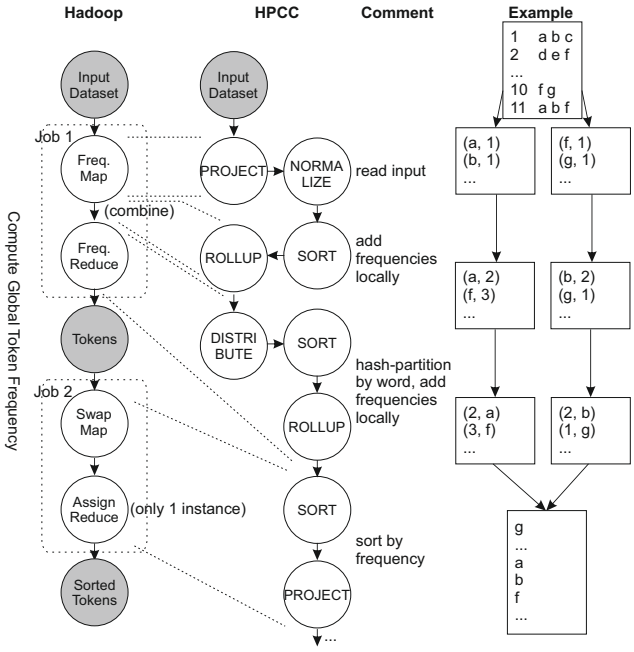

Fig. 1. Hadoop and HPCC Dataflows (First Step).

Figure 1 shows the dataflows in Hadoop and HPCC for the first step. Consider the example in the right column of the figure. The example assumes a parallelization degree of 2. The input has the form $\langle recordId, \langle tokenId1, .., N\rangle\rangle$. For each token in each object, we create a new record $\langle tokenId, 1\rangle$. The 1 represents the initial count of the token. In the following step, we add the token count for each partition. In the last two steps, we order the tokens according to their frequency.

Note that we use a *Combine* in Hadoop Job 1, which computes the token counts locally on the Map side. This significantly reduces network traffic and shuffle costs at the the Reduce side. We apply the same idea in HPCC by using ROLLUP and SORT in local mode. As in Hadoop, it is important to apply this concept. The ECL compiler does not automatically insert such an optimization.

In Hadoop, the *number of Map instances* is dependant on the HDFS block size by default. The default block size is either 64 or 128 MB. If the number of data blocks is smaller than the number of available Map instances, only a subset of the available Map instances is used by default. If in addition the first-order function is compute-intensive, this can lead to a longer runtime compared to executing Map on all available Map instances. This issue might be solved for

example by manually changing the input split size parameter of Hadoop. The distributed file system of HPCC splits the data at object borders and evenly distributes it amongst the available compute nodes. If the subsequent operator can operate on independant data chunks, it is executed on each data split.

In Hadoop, we manually set the *number of Reduce instances*. The default number is 1. If it is set too low, the computing nodes are under-utilized. If it is set too high, resources like main memory or network get overloaded. An optimal value is usually application- or even data-dependant. HPCC handles this parallelization implicitly.

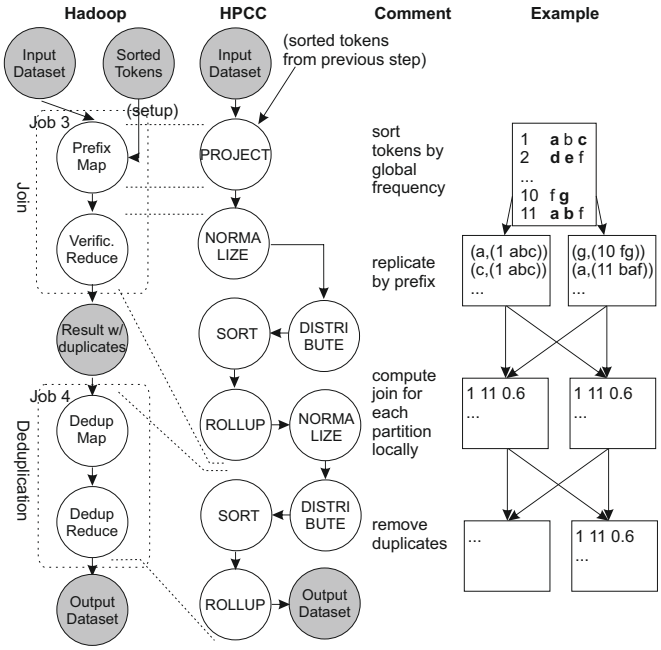

Fig. 2. Hadoop and HPCC Dataflows (Second and Third Step).

Figure 2 shows the dataflows for the second and third step of the implementations. Consider the data example in the right column. We read the input, sort the tokens by their frequency in ascending order, and replicate each object for each token in the prefix. We illustrate the prefix with bold numbers in the input. Since this replication can be computed on independent data partitions, we use two boxes for the resulting partitions. We group all records with the same tokens and compute their pairwise similarity. If two records share more than one common token in the prefix, the similarity of this pair is computed more than once. In the last step, we deduplicate the result.

The Map in Hadoop Job 3 uses a setup function, which initially reads the word frequencies from the first step. It sorts the words in each object according

to their global frequency and computes the prefix length. For each word in the prefix, it outputs the key-value pair $\langle\langle word, objectLength\rangle, object\rangle$. Note that the key consists of two integers, the word and the length. As proposed by Vernica et al. [3], we use a combined key consisting of the word and the length of the containing record. The word in the key partitions the data. The length is used additionally for local sort on the Reduce side. The Reducer retrieves the objects ordered by length. Since it performs a local nested-loop, we can *prune locally buffered objects* which cannot be similar anymore to all subsequent objects due to their length difference. We implemented a custom partitioner, sorter and grouper for this. This approach has an impact on the runtime. If the number of locally buffered objects exceeds memory boundaries, the computation becomes slow. In ECL, we implement the same approach by sorting the records by length within each partition. For each partition, we run a user-defined Java function that computes the similarity join locally. As in Hadoop, the user-defined function is stateful and can cause memory overflow.

5 Summary, Future Work

We were interested in how HPCC might be an alternative to Hadoop as an execution platform to allow for a fairer comparison when implementing MapReduce algorithms. Using the VernicaJoin to implement a textual similarity join, we showed that a complex MapReduce algorithm can be adapted to HPCC in a straightforward way. HPCC takes away some configuration details from the user like the parallelization degree (number of Map and Reduce instances). However, ECL still requires its users to carefully partition data so that intermediate buffers do not get overloaded. It is also necessary to explicitly implement optimizations such as local Combines. We plan to investigate further the influence of memory configuration on runtime. Especially in Hadoop, it is usually not clear to the user how its memory-related parameters impact performance. Furthermore, we plan to adapt this textual similarity join approach to use even more native ECL functions rather than user-defined "black box" code. This opens optimization possibilities which can potentially be integrated into the HPCC system.

Acknowledgements. This work was supported by the Humboldt Elsevier Advanced Data and Text (HEADT) Center.

References

1. Babu, S.: Towards Automatic Optimization of MapReduce Programs. In Proceedings of the 1st ACM symposium on Cloud computing. ACM (2010)
2. Middleton, A. M., Bayliss, D. A., and Halliday, G.: ECL/HPCC: A Unified Approach to Big Data. In: Furth, B. and Escalante, A.: Handbook of Data Intensive Computing, pp. 59–107. Springer, New York (2011)
3. R. Vernica, M. J. Carey, and C. Li. Efficient parallel set-similarity joins using mapreduce. In Proceedings of the 2010 ACM SIGMOD International Conference on Management of data, pages 495506. ACM (2010).