Parallel Code Clone Detection Using MapReduce

Hitesh Sajnani, Joel Ossher, Cristina Lopes
University of California, Irvine
Irvine, USA
hsajnani@uci.edu, jossher@uci.edu, lopes@ics.uci.edu

***Abstract*—Code clone detection is an established topic in software engineering research. Many detection algorithms have been proposed and refined but very few exploit the inherent parallelism present in the problem, making large scale code clone detection difficult. To alleviate this shortcoming, we present a new technique to efficiently perform clone detection using the popular MapReduce paradigm. Preliminary experimental results demonstrates speed-up and scale-up of the proposed approach.**

I. Introduction

Problem Formulation: A project consists of blocks of source code P: {B1, B2, B3, ...} A block of source code consists of various source terms B: {T1, T2, T3, ...}. Given two projects P1 and P2, a similarity function *f*, and a threshold parameter *t*, the aim is to compute and find all block pairs P1.B and P2.B where f(P1.B, P2.B) $\geq$ T

The problem becomes extremely computationally intensive if clone detection is to be performed on source code repositories consisting of thousands of large projects having millions of blocks each of which is to be compared against other. Detecting such similar pairs is challenging because the large amount of data usually do not fit in the main memory of one machine. Attempts are made to address the issue [1], [2]. But we still need more such tools and techniques to perform large scale clone detection efficiently [3]. Recently, the MapReduce [4] paradigm has proven to be helpful for being a scalable parallel shared-nothing data-processing platform for data intensive, similarity search applications [5], [6], [7]. We take inspiration from the previous work to explore the possibility of large-scale code clone computation using MapReduce.

II. The MapReduce Paradigm

MapReduce is a popular paradigm for data-intensive parallel computation in shared-nothing setting. The data is partitioned across the nodes of a cluster and stored in a distributed file system. Data is represented as (key, value) pairs. Each computation is represented using two functions - *Map* and *Reduce*. Map takes one pair of data with a type in one data domain, and returns a list of pairs in a different domain:

$$Map\langle k1, v1\rangle \rightarrow list\langle k2, v2\rangle$$

The Map function is applied in parallel to every item in the input dataset. This produces a list of $\langle k2, v2\rangle$ pairs for each call. After that, the MapReduce framework collects all pairs with the same key from all lists and groups them together, thus creating one group for each one of the different generated keys. The Reduce function is then applied in parallel to each group, which in turn produces a collection of values in the same domain:

$$Reduce\langle k2, list(v2)\rangle \rightarrow list(v3)$$

The returns of all calls are collected as the desired result list. The framework also allows users to have handle on

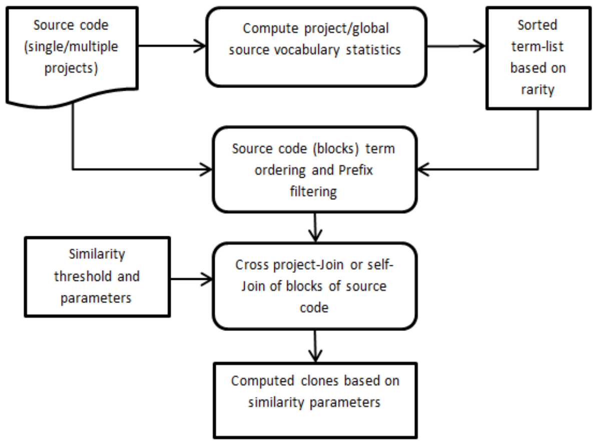

Source code (single/multiple projects)
Compute project/global source vocabulary statistics
Sorted term-list based on rarity
Source code (blocks) term ordering and Prefix filtering
Similarity threshold and parameters
Cross project-Join or self-Join of blocks of source code
Computed clones based on similarity parameters

Fig. 1: High-level algorithm for clone detection

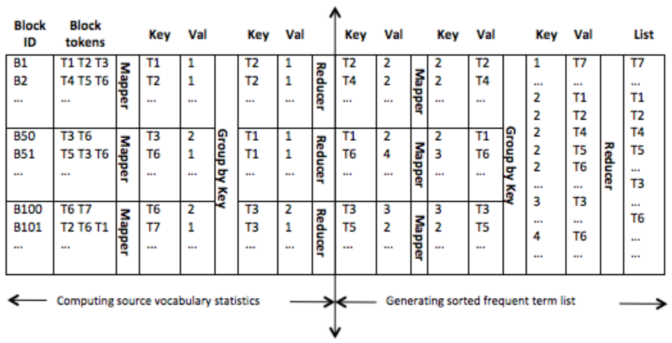
implementation of each MapReduce function and customize hashing and comparison to be used when partioning and sorting the keys.

III. Implementation

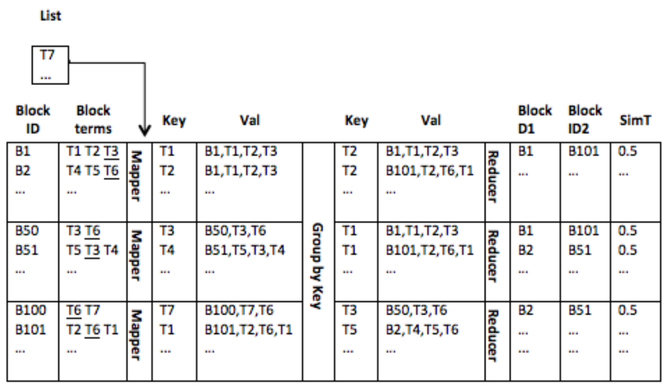
The principle idea of the approach, similar to the one discussed in [7], is to hash partition the source code blocks across the network based on the computed keys and group together source code blocks which have the same key. Since we also want to compute near miss clones, we use signature generated from the source code blocks as partioning keys instead of the source code blocks directly. Figure 1 describes the high level algorithm.

The first step is to compute the project vocabulary statistics to generate good signatures. Figure 2 describes the way the statistics get computed using two map reduce phases which eventually produces sorted list of terms based on rarity. The first phase computes the frequency of each term and the second phase sorts the terms based on their frequencies.

In the first stage, the source code block is given as input to the mapper. The mapper tokenizes each source code block, extract source terms out of it, and produces a key-value pair of the form $\langle term, 1\rangle$ for each term in the source code block. In Figure 2 (a), the block with Block ID 1 has value ”T1 T2 T3”, which is tokenized as source terms ”T1”, ”T2”, and ”T3”. The reducer computes the total occurence for each source term and outputs $\langle sourceterm, value\rangle$ key-value pairs, where key is ”source term” and value is the total frequency for the term across all blocks. Stage II uses map reduce to sort these $\langle sourceterm, value\rangle$ key-value pairs generated from the stage I. The map function exchanges the input keys and values so


Block ID
Block tokens
Key
Val
Mapper
Group by Key
Reducer
List
Computing source vocabulary statistics
Generating sorted frequent term list

(a) Frequent source term list computation



(b) Similar code block detection

Fig. 2: MapReduce phases in detection algorithm

that the input pairs of the reduce function are sorted based on their frequencies. The reducer just outputs the values without keys.

The information computed will enable us to create effective filters which can drastically reduce the number of block pairs (potential clones) whose similarity needs to be verified. We use a technique called prefix filtering [8] which works as follows. The terms in the blocks are ordered based on the sorted list of terms produced above. For each block of source terms, we define its prefix of length n as the first n terms in the ordered block terms. The prefix length is a function of number of source terms in the block, similarity threshold value and also similarity function. The basic idea behind prefix filtering principle is that if two block of source code share rare terms, there is a chance that it might be similar. Since we are doing a global term ordering based on term frequencies, prefix set of a block contains rare terms. If no terms are shared in prefix set, that block can be avoided from further processing.

Figure 2 (b) describes the computation process. The mapper retrieves the original blocks one by one, tokenizes it and reorders the tokens based on their frequencies, using the rarity term list as a reference. Next, it computes the prefix length and extracts the prefix terms. (terms that are not underlined). Considering each source term as a key, we produce a $\langle key, value \rangle$ pair for each of its prefix terms. Hence we project a source code block multiple times (the number of its prefix terms). For example, if the code block value is ”T1 T2 T3 T4” and the prefix terms are ”T1”, and ”T2”, we would output two $\langle key, value \rangle$ pairs, corresponding to the two prefix terms. The reducer groups together all the values sharing same prefix tokens. In the end, a single reducer, for each pair of code block projections applies length and positional filters and checks the pair if it survives. If a pair passes the similarity threshold set by the user, the reducer outputs block pairs as clones with their similarity values.

IV. Preliminary Results

Table I lists the speed up of the proposed approach on 35 projects of apache commons repository. The cluster was setup using Amazon web services with the following node specification: 1.7 GB memory, 160 GB storage, 1 EC2 compute unit (1 virtual core), Hadoop 0.20. We speculate performance gain to improve with larger dataset (order of 100 and 1000 of projects) as HDFS performs better with large files.

TABLE I: Sample speed-up results

Nodes	Jaccard Similarity Threshold	Time (seconds)
4	0.5	436
4	0.8	244
8	0.5	301
8	0.8	188

V. Future Work and Conclusion

We plan to perform experiments on large data sets, using various threshold and similarity functions and collect empirical data to statistically validate our approach. Currently, our granularity of clone detection is blocks which limits our approach from detecting arbitrary subsequences of cloned code. We plan to use sliding window approach while generating blocks to alleviate this shortcoming. In conclusion, research in software clones is very close to industrial application. Among other things, scalability and speed are also major issues of industrial adoption of methods and tools. The proposed approach demonstrates a practical and inexpensive technique using commodity machines for large scale code clone detection by exploiting the inherent parallelism present in the problem. Since the solution is based on top of MapReduce, it comes with advantages like load balancing, data replication, and fault tolerance over any other in house distributed solutions where these things are to be dealt with explicity.

References

[1] S. Livieri, Y. Higo, M. Matsushita, and K. Inoue, “Very-large scale code clone analysis and visualization of open source programs using distributed ccfinder: D-ccfinder,” in *Proceedings of ICSE*, 2007, pp. 106–115.
[2] I. Keivanloo, J. Rilling, and P. Charland, “Internet-scale real-time code clone search via multi-level indexing,” in *Proceedings of WCRE*, 2011.
[3] C. K. Roy, J. R. Cordy, and R. Koschke, “Comparison and evaluation of code clone detection techniques and tools: A qualitative approach,” *Sci. Comput. Program*, pp. 470–495, 2009.
[4] J. Dean and S. Ghemawat, “Mapreduce: simplified data processing on large clusters,” *Commun. ACM*, vol. 51, no. 1, 2004.
[5] C. Xiao, W. Wang, X. Lin, and J. X. Yu, “Efficient similarity joins for near duplicate detection,” in *Proceedings of ICWWW*, 2008.
[6] R. J. Bayardo, Y. Ma, and R. Srikant, “Scaling up all pairs similarity search,” in *Proceedings of ICWWW*, 2007.
[7] R. Vernica, “Efficient processing of set-similarity joins on large clusters,” Ph.D. dissertation, University of California, Irvine, 2011.
[8] S. Chaudhuri, V. Ganti, and R. Kaushik, “A primitive operator for similarity joins in data cleaning,” in *Proceedings of ICDE*, 2006.