A Parallel and Efficient Approach to Large Scale Clone Detection

Hitesh Sajnani, Vaibhav Saini, Cristina Lopes
Donald Bren School of Information and
Computer Science
University of California, Irvine
Irvine, California 92617, USA
{hsajnani, vpsaini, lopes}@uci.edu

***Abstract*—We propose a new token-based approach for large scale code clone detection. It is based on a filtering heuristic which reduces the number of token comparisons. The filtering heuristic is generic and can also be used in conjunction with other token-based approaches. In that context, we demonstrate how it can reduce the index-creation time and memory usage in index-based approaches. We also implement a MapReduce based parallel algorithm that implements the filtering heuristic and scales to thousands of projects.**

In our two separate experiments, we found that: (i) the total number of token comparisons are reduced by a factor of 10, increasing the speed by a factor of 1.5; and (ii) the Speed-up and Scale-up of the parallel implementation appear to be linear on a cluster of 2-32 nodes for 150-2800 projects.

I. Introduction

Code clone detection aims at finding exact or similar pieces of code known as code clones. Several techniques have been proposed for clone detection over many years [1], [2], [3], [4], [5], [6], [7], [8], [9], [10], [11], [12], [13], [14], [15]. These techniques differ in many ways ranging from the type of detection algorithm they use to the source code representation they operate on. Techniques using various representations include Tokens [4], [5], Abstract Syntax Trees [6], [7], [8], Program Dependence Graphs [9], [10], Suffix Trees [6], [11], [4], [12], Text representations [13], [14], Hash representations [15], etc. Each of these different approaches have their own merits and are useful for different use-cases. For example, AST based techniques have high precision, and are useful for refactoring, but may not scale. Moreover, token based techniques have high recall but may yield clones which are not syntactically complete [16]. They are useful where high recall is important. Many of these techniques are proposed to handle use-cases where a fragment of the code or smaller subset of the code clones have to be detected [17], [18], [2]. These techniques are perceived to be useful mostly in software maintenance, as until recently, cloning was considered to be a bad practice. However, this paradigm is changing [19], [20], and researchers have started exploring other use-cases such as license violation detection [12], [21], reverse engineering the product lines [22], [21], finding the provenance of the component [23], plagiarism detection, and code search [24], [25]. While previous research has mainly focused on identifying code fragments on a per-project basis, several of these new use-cases have opened up needs for identifying all the clones in the system or a family of systems. This leads to new research challenges to deal with massive amount of data and computation.

Motivating Scenario: To illustrate the challenges of such new use-cases, consider a retail banking software system maintained by Tata Consultancy Services (TCS), that is in active use by a number of banks (with different codebases). The company decided to form a common codebase for all these banks. The motivation being complexity and the business losses occurring due to duplicated efforts in: (i) delivering common features; and (ii) maintaining these common features separately.

To start with, one of the challenges the team faced, was to detect the blocks of common code across such large codebases. Each codebase runs into many million lines of code that span across few thousand COBOL programs, most of which, are about 100K LOC. Detailed description of the case study is available in [26]. As part of the exploration, the team first ran CloneDR[1], an AST based commercial clone detection tool on ML0000, one of the few thousand programs, consisting of 88K LOC. It took around 8 hours to compute all the clones on a laptop machine[2]. Of course, the tool produced much more than just clones (e.g., information on how to parameterize clones), but there are eight such codebases live at different banks. The aim is to compute all the similar blocks of code across all eight codebases to help the team make decisions about creating the common codebase. The computation is beyond the capacity of any single commodity machine and represents the scale at which the tools need to perform in the industry. This situation at TCS may not be unique and in fact it is likely to represent the state of many companies in the service industry as they are moving away from the green field development model and adopting the packaging development model.

Thus, such new use-cases of large scale clone detection, are likely to increase in the future. They open up new opportunities for the research in clone detection, however, they bring in new scalability and performance challenges as well.

[1]http://www.semdesigns.com/Products/Clone
[2]System config.: T7300(2GHz) processor, 4GB RAM, 160GB storage

Substantial research is also done to scale vertically using high power CPUs and memory added to a single machine and building complex data structures [14], [12]. This approach works well for certain use-cases, but it is bounded by the amount of data that can fit into the memory of a single machine. However, such techniques may not be efficient in scenarios which require computation of all the clones in the system or a family of sytems.

A recent trend is to also investigate algorithms that compute the inverted indexes. Index-based approaches [27], [24], [28] have demonstrated promise for certain large scale use-cases like code search, license violation detection, etc. However, the main challengess are the time taken to create an index, and the amount of memory consumed by the index. The creation of index can take hours even if done in parallel. This time may amortize if the index is reused for multiple searches. However, in scenarios when a company is interested in finding all the clone across multiple systems to migrate towards product lines, or for re-engineering purpose, clone analysis is less frequent, and may not amortize the cost of index creation. Nonetheless, index-based approaches have successfully scaled to ultra-large sytems and repositories, with an opportunity to minimize index creation time and memory footprints.

In general, while computing all the clones in the system or a family of systems, the main challenge is that each code block has to be compared against every other code block. This leads to tremendous number of comparisons. Moreover, in case of large datasets, and for some of the use-cases mentioned above, these comparisons are beyond the capacity of a single machine. In such cases, efficient parallelization of the computation process is the only feasible option. Previously, research in this direction was limited due to the lack of the availability of resources and the cost of setting up the infrastructure. But, the recent developments in the field of cloud computing and the availability of low-cost infrastructure services like Amazon Web Services, Azure, etc., have motivated the research in this area. However, it is important to note that just dividing the input space, and parallelizing the clone detection operation does not solve the problem, because, running tools on projects individually, and then combining the results in the later step would lead to a collection of common clones, but would not identify clones across division boundaries. Also, often, simple distributed approaches consume substantial time in aggregating the information after the clone detection step. For example, Livieri et al. [29], proposed an in-house distributed approach (20 KLOC) which took 2 days to analyze 400 million lines of code with a cluster of 80 machines.

Contributions: In this paper, we propose a new token-based approach to large scale clone detection. It uses a filtering principle to reduce the number of code block comparisons without affecting the recall. The principle exploits the idea that similar code blocks are likely to share rare tokens. The technique is generic and can be used in conjunction with other token-based approaches to increase performance. We also present how this technique can be used to reduce the time and memory usage for index creation in index-based approaches. Towards the end, we demonstrate a MapReduce based distributed and parallel clone detection approach that effectively implements the proposed filtering technique to achieve scalability. We conduct several experiments to (i) demonstrate the effectiveness of the filtering approach to reduce the total time and the number of comparisons done to detect clones; and (ii) measure the Speed-up and Scale-up of the MapReduce based distributed and parallel approach by deploying the solution in Amazon cloud cluster

Outline: In Section 2, we formulate the problem of clone detection and introduce necessary concepts to understand the approach. Subsequently, the paper is divided into three logical sections, describing: (i) Our approach based on the filtering technique (Section 3); (ii) Applicability of the filtering technique on Index-based approaches (Section 4); and (iii) A MapReduce based parallel algorithm for our approach (Section 5). In Section 6, we present experiments to evaluate the effectiveness of the proposed approach, along with experiments to demonstrate the Speed-up and Scale-up of the parallel algorithm. Section 7 covers some more related work on Scalable and Parallel approaches. In Section 7, we conclude with a summary of our contributions.

II. Background

Problem Formulation: We transform the problem of clone detection into a set similarity problem. We represent a project P as a collection of source code blocks $P : \{B_1, ..., B_n\}$. Similarly, a code block B consists of many source terms $B : \{T_1 ..., T_k\}$. The size of a set is the number of elements in the set, and is denoted by $|B|$ for a set B. Since a code block may have repeated source terms, we represent the elements of the of B as a $< term, frequency >$ pair. Here, $frequency$ denotes the number of times a $term$ occured in a code block.

In order to quantitatively infer if the two code blocks are clones, we use a similarity function. The similarity function measures the degree of similarity between two code blocks and returns a non-negative value. The higher the similarity value, the higher the similarity between the code blocks. Thus we can treat pairs of code blocks with similarity value higher than the specified similarity threshold as clones. A *similarity join* between the projects will result in all the pairs of code blocks having similarity value above a given threshold - interproject clones. Similarly, a *self-similarity join* will reveal all the blocks of code which are clones in a project - intraproject clones.

Formally, given two projects $P1$ and $P2$, a similarity function f, and a threshold parameter θ, the aim is to find all the code block pairs $P1.B$ and $P2.B$ $s.t$ $f(P1.B, P2.B) \geq t$. Note that for intra project similarity, $P1 = P2$, i.e, a case of self-similarity join. Similarly, all the clones in the repository can be revealed by doing a self-similarity join on the entire repository.

We use $Overlap$[3] similarity as a similarity function. It is a widely used similarity function when dealing with textual content. Given two code blocks B_1 and B_2, the overlap similarity $O(B_1, B_2)$ is computed as the number of source terms shared by B_1 and B_2. Formally,

$$O(B_1, B_2) = |B_1 \cap B_2| \quad (1)$$

In other way, if θ is specified as 0.8, and $max(|B_1|, |B_2|)$ is t, then B_1 and B_2 should at least share $\lceil \theta \dot{|} t \dot{|} \rceil$ terms to be identified as similar blocks (clones of each other).

Example: Figure 1 represents the two methods from Apache Coocon project. For the ease of representation, Table I shows the mapping of source terms with shorter terms. Using this mapping, the methods can be transformed into following sets:

$$\begin{aligned} B_1 =\{&(T1,1),(T2,1),(T3,2),(T4,1),(T5,2),(T6,2),\\ &(T7,2),(T8,2),(T9,1),(T10,1),(T11,5),(T12,2),\\ &(T13,2),(T14,1),T(15,1),(T16,1)\} \end{aligned}$$

$$\begin{aligned} B_2 =\{&(T1,1),(T2,1),(T17,2),(T4,1),(T5,2),(T6,2),\\ &(T7,2),(T8,2),(T9,1),(T10,1),(T11,5),(T12,2),\\ &(T13,2),(T14,1),T(15,1),(T16,1)\} \end{aligned}$$

The size of each block is 27 terms, out of which, they share 25 terms. Given the Overlap similarity function, and the threshold θ specified as 0.8, the minimum number of terms they should share in order to be identified as similar is 22 ($\lceil \theta \dot{|} t \dot{|} \rceil$). Since they share 25 terms (> 22), they are similar. However, if we change θ to 0.95, the minimum number of shared terms should be 26. Hence, in that case, the two blocks will not be identified as clones.

III. Approach

In this section, we start by presenting a naive approach, and then gradually propose a series of refinements to arrive at the final approach.

A. The Naive Approach

A simple approach is to compare every code block with another, thus bearing a prohibitively $O(n^2)$ time complexity. The slightly optimized version is to exploit symmetry to reduce the number of comparision to half. Nonetheless, the number of comparisons is still large. The problem becomes intractable when clone detection is to be performed on source code repositories consisting of thousands of large projects having millions of code blocks, each of which is to be compared against other. Detecting such similar pairs is challenging because such large amount of data usually does not fit in the main memory of one machine. Hence, it is an algorithmic challenge to perform this comparison in an efficient and scalable way.

[3]The presented approach is very general and can be used with other similarity function like Jaccard, Cosine, etc.

B. Efficiency Improvement Using Filtering

In this section, we present a filtering heuristic that improves the efficiency of the naive approach. Inspired by the arguments in [30], we derive that when two code blocks have a large overlap, even smaller sub-blocks of the those code blocks overlap. Formally, we can state it in the form of the following property:

Property 1: *If blocks B_1 and B_2 consist of t terms each, and if $|B_1 \cap B_2| \geq i$, then any sub-block of B_1 consisting of $t - i + 1$ terms will overlap with block B_2.*

To illustrate, consider a case when all the blocks have a fixed set of terms. For example, consider the following two blocks:

$$B_1 = \{a, b, c, d, e\}$$

$$B_2 = \{f, b, a, d, c\}$$

.

B_1 and B_2 have total of 5 terms ($t = 5$) each, and an overlap of 4 terms ($i = 4$). Let B_{sb1} represent all the sub-blocks of B_1 of size 2 ($t - i + 1$):

$$\begin{aligned} B_{sb1} =\{&\{a,b\},\{a,c\},\{a,d\},\{a,e\},\{b,c\},\{b,d\},\\ &\{b,e\},\{c,d\},\{c,e\},\{d,e\}\} \end{aligned}$$

Then, according to Property 1, all the elements of B_{sb1} will have a non-zero overlap (at least share one term) with B_2.

To understand the implications of this property, consider the above two blocks with 5 terms each, and θ specified as 0.8. That is, the two blocks should share at least $0.8 * 5 = 4$ terms to be considered similar. In this context, if we were to find out if B_1 and B_2 are similar, then, using property 1, we can infer that all the sub-blocks of B_1 should share at least one term with B_2. Thus, if any sub-block of B_1 does not share a term with B_2, we can immediately say that B_1 and B_2 cannot be similar for the given threshold. Interestingly, the above principle suggests that instead of comparing all the terms of B_1 and B_2 against each other, we may compare terms of any sub-block of B_1 to quickly identify cases when B_1 and B_2 are not going to be similar. By filtering out a large subset of B_1, we can often reduce the total number of comparisons by very significant margins.

The above example applies filtering to only block B_1. Naturally, the question is, if and how can we apply it to both the blocks? It turns out that the filtering property can be applied to both the blocks if they are in the same order.

Consider blocks B_1 and B_2 consisting of t terms each in some predefined order. If $|B_1 \cap B_2| \geq i$, then the sub-blocks B_{sb1} and B_{sb2} of B_1 and B_2 respectively, consisting of the first $t - i + 1$ terms, must share at least one term.

[4]The methods are present in file *DOMTransformer.java* in *org.apache.cocoon.transformation*

public	void	characters	char	c	int	start	len	throws	SAXException	this	stack	setOffset	oldOffset	consumer	newOffset	ignorableWhiteSpace
T1	T2	T3	T4	T5	T6	T7	T8	T9	T10	T11	T12	T13	T14	T15	T16	T17

TABLE I: Source Term Mapping

```
public void characters(char c[], int start, int len)
throws SAXException {
    this.stack.setOffset(this.oldOffset);
    this.consumer.characters(c, start, len);
    this.stack.setOffset(this.newOffset);
}
```

```
public void ignorableWhitespace(char c[], int start, int len)
throws SAXException {
    this.stack.setOffset(this.oldOffset);
    this.consumer.ignorableWhitespace(c, start, len);
    this.stack.setOffset(this.newOffset);
}
```

Fig. 1: Two methods from Apache Coocon project [4]

In the previous example, consider the terms in blocks to be alphabetically ordered.

$$B_1 = \{a, b, c, d, e\}$$

$$B_2 = \{a, b, c, d, f\}$$

As mentioned above, since the blocks are in same order, the sub-blocks consisting of the first 2 $(t - i + 1)$ terms must have a non-zero overlap. This is the case above, as both the sub-blocks consist of $\{a, b\}$. Formally, it can be stated in the form of the following property:

Property 2: *If blocks B_1 and B_2 consist of t terms each, which follow an order O, and if $|B_1 \cap B_2| \geq i$, then the sub-block B_{sb1} consisting of the first $t - i + 1$ terms of B_1 will overlap with the sub-block B_{sb2} consisting of first $t-i+1$ terms of B_2.*

This property significantly reduces the total number of comparisons between the code blocks as we do not have to compare all the code blocks always; instead we need to compare them only when their sub-blocks overlap. The sub-blocks consist of the the few prefix terms of the block. The natural question is how many prefix terms should be considered? It is important to note that the length of the prefix of a block depends on the size of the block. Hence, it cannot be same for all the blocks. Moreover, prefix length has implications on the completeness of the results. Hence, if computed incorrectly, it can eliminate blocks which could be similar.

C. Computing the Correct Prefix Length

We can compute the correct prefix length in the following way:
Using property 2, we know that the minimum number of overlapping terms between the two code blocks is:

$$t - i + 1 \tag{2}$$

Now, given the similarity threshold value as θ and the number of terms in the code block (bigger of the two to be compared) as t, the minimum number of overlapping terms can be computed as:

$$\lceil \theta |\dot{t}| \rceil \tag{3}$$

Comparing equation (2) and (3), we have the size of prefix i as:

$$i = |t| - \lceil \theta |\dot{t}| \rceil + 1 \tag{4}$$

Hence, in a code block of t terms, and a similarity threshold value of θ, the prefix size of $|t| - \lceil \theta |\dot{t}| \rceil + 1$ will ensure that no potential clones will be missed.

D. Ordering

As stated in property 2, the approach is applicable only when the blocks are in the same order. Thus, we can improve the effectiveness of the approach by choosing the right ordering. The goal is to pick an ordering that minimizes the number of comparisons among the code blocks. One way is to order the terms by the increasing order of their frequency in the entire code base. In this way, we can eliminate very frequent terms, and thereby expect to minimize the number of comparisons. The basic idea is to eliminate the comparison of frequent terms and instead, check if two code blocks share rare terms.

Term Frequency-Inverse Document Frequency (TF-IDF) is one such measure that captures this intent well. We create a global term list from the subject system or repository based on this measure. Next, we order the terms in the code blocks using this global order. Since high frequency terms will have lower weights, they will appear at the end in the code blocks. Therefore, the sub-blocks will mostly consist of rare terms and hence reducing the subsequent number of comparisons. It is trivial to compute the global term list when dealing with few projects. However, as the number of projects in the corpus increases, the number of terms becomes extremely large. Hence, computation of the global term list is not trivial and expensive. Later in Section V, we will describe an algorithm for how this can be parallelized using the MapReduce framework. Subsequently, we will present a MapReduce based clone detection algorithm which makes use of this computed list.

To summarize, Figure 2 brings everything together and describes the high level algorithm.

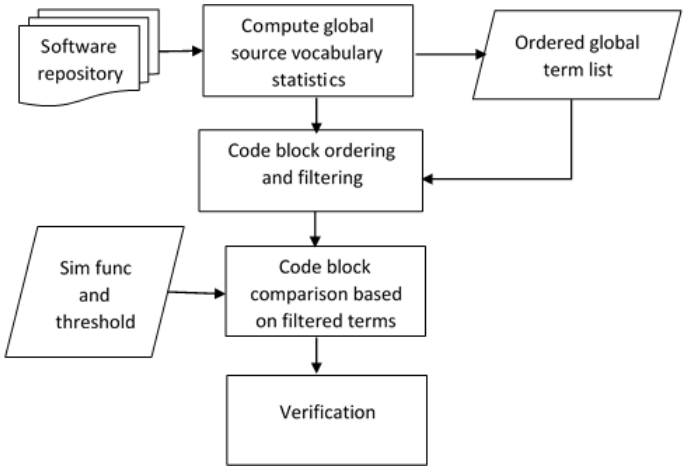

Fig. 2: High-level algorithm for clone detection

IV. Applicability to Index-based approaches

In this section, we highlight some of the limitations of index-based approaches and also describe ways to overcome some of the them using the filtering principle.

The core idea of an index-based approach is to build an inverted index of all the terms (or computed hashes) of all the code blocks in the system. To find clones of a given code block, the same processing, as to building an inverted index, can be applied to the block. The resultant terms or hashes and can then be issued as a query to find all the similar code blocks. The retrieval system itself is encoded with the logic to return only the code blocks which are above the similarity threshold. Although fast, and suitable for some use-cases, there are following limitations to this approach:

1) The index needs to be built upfront before producing any output;
2) The size and time for index creation depends heavily on the code block size chosen for clone detection; and
3) The index size increases with the increase in dataset, and in order to perform well, the index has to be in memory.

We show that the first issue can be alleivated by building the index dynamically while the clones are computed. Later, we refine this approach by using the filtering principle to reduce the index size. The reduced index size eventually reduces the index creation time and the memory usage (issue 2 and 3).

A. Clone Detection Using Dynamic Indexes

Algorithm 1 provides the pseudo-code for dynamically creating the index for clone detection while the clones are computed.

Terminology: B represent a set of code blocks, and I represent a set of inverted indices. Set I consists of t inverted indices, one for each term. (Inverted indices can also be constructed by computing a hash for a group of terms instead of each term, using a sliding window approach). The inverted index I_t for a term t consists of a list of block ids in which the term is present. R is the result set with all the computed clones.

At a higher level, *cloneDetection()* function scans all the code blocks, invokes *detectClones()* (line 5) and later builds the inverted indices incrementally (line 6-8). The *detectClones()* function computes all the clones for the given block. Each term of given block is searched in the inverted indices and the block ids of other blocks where the term is present are retrieved (line 16 & 17). These are potential clones of a given block. We call them candidate blocks. The similarity score for each candidate block is computed and incrementally updated as more terms are processed in the similarity map, $simMap$ (line 18).

```
1  cloneDetection(B, sim, θ)
2  begin
3      I, R ← {}
4      for each code block b1 in B do
5          R ← R ∪ detectClones(b1, sim, I, θ);
6          for each term t in b1 do
7              It ← It ∪ id(b1);
8          end
9      end
10     return R;
11 end
12 detectClones(b1, I, sim, θ)
13 begin
14     simMap ← new HashMap()
15     cloneSet ← {}
16     for each term t in b1 do
17         for each id(b2) in It do
18             simMap[id(b2)] ← sim(b1, b2)
19         end
20     end
21     for each id(b2) in simMap do
22         if id(b2) ≥ θ then
23             cloneSet ← cloneSet ∪ { id(b1),id(b2),
               simMap[id(b2)]} ;
24         end
25     end
26     return cloneSet;
27 end
```

Algorithm 1: Clone detection using dynamic index

Once all the terms of the given block are processed, the algorithm checks the score of each candidate block in the $simMap$, and adds it to the $cloneSet$ if it meets the similarity threshold criteria set by the user (line 21-24). After the $cloneSet$ is populated, the block is indexed in the *cloneDetection()* function (line 6-8). The main benefit of the above approach is that it creates the index as the clones are computed dynamically. Hence, it produces output, even when the index is partial.

B. Reducing the Index size

The above approach although dynamic, like other index-based approaches, still require building the complete inverted

index over the code blocks. In this section, we demonstrate how the filtering technique can be used in conjunction with the dynamic index approach to reduce the amount of information indexed during index creation. The idea is to build smaller indexes of only prefixes of each block. This drastically reduces the overheads such as complete index construction and scanning to calculate similarity score.

Algorithm 2 refines dynamic index approach using the fitlering technique. The core idea is to index just enough terms to ensure that any clone candidate code block is not missed. The algorithm is very similar to Algorithm 1, except here the *while* loop of the *cloneDetect()* function indexes on the subset of each block (line 7-10). We call this index of only prefixes as a partial index.

```
1  cloneDetection(B, sim, θ)
2  begin
3      I, R ← {}
4      for each code block b1 in B do
5          R ← R ∪ detectClones(b1, sim, I, θ);
6          i ← 0
7          while i ≤ (|b1|-⌈ θ |b1| ⌉+1) do
8              I_t ← I_t ∪ id(b1)
               /* Here t is a term at b1[i]   */
9              b1[i] ← 0
10         end
11     end
12     return R;
13 end
14 detectClones(b1, I, sim, t)
15 begin
16     simMap ← new HashMap()
17     cloneSet ← {}
18     for each term t in b1 do
19         for each id(b2) in I_t do
20             simMap[id(b2)] ← sim(b1, b2)
21         end
22     end
23     for each id(b2) ∈ simMap do
24         simVal ← simMap[id(b2)]+sim(b1, b̄2)
           /* Here b̄2 is the unindexed subset
              of b2                          */
25         if simVal ≥ θ then
26             cloneSet ← cloneSet ∪ { id(b1),id(b2),
               simVal}
27         end
28     end
29     return cloneSet;
30 end
```

Algorithm 2: Clone detection using dynamic index and optimized using filtering

As in Algorithm 1, the *detectClones()* function is invoked for each block (line 4 & 5). However, since we store only partial index for each block, the look ups in the inverted indices are are only for the prefixes (line 18 & 19). If a block is identified as a candidate (match in the partial index), the similarity score computed is computed (line 20). Note that this similarity score computation is not complete, as it is computed only for the terms in the partial index. Later similarity score computation is updated for the remaining terms (unindexed terms) (line 24). We use the previously computed score on the partial index (line 20) to avoid recomputing the score completely. Morever, the indexed terms are removed from the code block terms once they are indexed (line 9) so that they are not duplicated in the partial index and the in code blocks. Hence, each term in the code block is stored either in the partial index or in the code block, but not both. This reduces the memory footprint and also speeds up the remaining computation to verify if the candidates are actually clones as the code blocks have fewer terms.

C. Ensuring the Correctness of the Approach

In order to ensure that the approach works correctly, we need to validate that: (i) the similarity scores between candidate blocks are computed correctly; and (ii) all the candidates having similarity scores greater than the threshold value are reported. The similarity score calculation of a block is divided into two parts: (i) similarity score from the indexed terms; and (ii) similarity score from the unindexed terms. The final score is the summation of both these similarity scores (line 24, Algorithm 2). For Overlap similarity,

$$sim(b_1, b_2) = sim(b_1, \overline{b_2}) + sim(b_1, \overline{\overline{b_2}})$$

Here $\overline{b_2}$ and $\overline{\overline{b_2}}$ are indexed and unindexed terms of b_2 respectively. Hence, similarity scores between the candidate blocks are computed correctly.

In order to validate if all candidate blocks are reported, recall that the prefix calculation using equation 4 guarantess that there is no impact on recall. Hence if block b_2 is a candidate for b_1, their indexed prefixes should have at least one term in common (match in the partial index look up). Thus, it follows that, b_2 should be present in the inverted index of the common term (checked at line 19).

D. Example

To illustrate, consider four code blocks where the terms are ordered based on their global occurrence frequency.

$$B1 = [\underline{T1, T2}, T7, T3, T8, ...]$$
$$B2 = [\underline{T4, T5}, T6, T7, T3, ...]$$
$$B3 = [\underline{T5, T6}, T1, T2, T7, ...]$$
$$B4 = [\underline{T6, T1}, T2, T7, T3, ...]$$

The prefix length of each code block is calculated using equation 4. The terms in the prefix set for each code block are underlined. Now, in an index-based approach, instead of indexing all the terms, only the terms in the prefix set (underlined terms) can be indexed. This reduces the index size. Moreover, this also reduces the number of index accesses. For example, consider block $B4$. In order to obtain the candidate blocks, we need to only look at the blocks returned by the inverted index of terms $T6$ and $T1$ (in comparison to all the

terms in a traditional index-based approach). This reduces the index accesses approximately by a factor (1-θ) whenever there is a failed match. Also, note the corresponding reduction in the verification stage as there are fewer candidates generated. This can increase the speed of the retrieval process.

V. Algorithm for Parallelizing the Approach

1) Overview: To recall, in order to apply filtering to code blocks, we have to compute two things: (i) the size of the prefix of each block; and (ii) the ordering of the prefixes such that rare terms are ordered first Prefix calculation for each block can be done using equation 4. However, the ordering of the prefixes needs global term frequency list. Due to big corpus size, this computation is non-trivial. Below we show how to compute this in parallel. After we have the ordered prefixes for each block, we compute keys for each block. The keys are computed using prefix terms. Next, the code blocks are hash partitioned across the network based on the computed keys. The code blocks having the same key are grouped together. Since we also want to compute Type II clones, we compute keys from code blocks (set) instead of the source code directly.

The next subsection describes the MapReduce programming model for parallel computation and describes a parallel algorithm for each of the above mentioned step.

2) Introduction to MapReduce: MapReduce is a distributed computing paradigm. It is inspired by the concepts of functional programming. It is used as a scalable approach for data intensive applications [31], [32]. It is based on two higher order functions: Map and Reduce. As shown in Figure 3, the Map function applies a function to each key-value pair in the input. The input is treated as a list of independent records. The result is an another list of intermediate key-value pairs.

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

The list is grouped by key and used as an input to the reduce function. The Reducer applies another function to its input and produce a list of values as its output.

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

The mapper and the reducer are functional in nature and hence they are side-effect free. For these reasons, they are easily parallelizable.

3) Computing the Global Frequent Term List using MapReduce: The list gets computed using two map reduce phases, which eventually, produce an ordered 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.

Algorithm 3 describes the first phase. Figure 4 describes an example to illustrate the process. The block with Block ID 1 has value "T1 T2 T3", which is given as input to the mapper. The mapper tokenizes each source code block, extracts the source terms out of it, and produces a key-value pair of the form $\langle T1, 1\rangle$, $\langle T2, 1\rangle$, $\langle T3, 1\rangle$, i.e., for each term in the source code block. The reducer computes the total occurrence for each source term and outputs $\langle term, value\rangle$ key-value pairs, where key is the *term* and value is the total frequency for the term across all blocks.

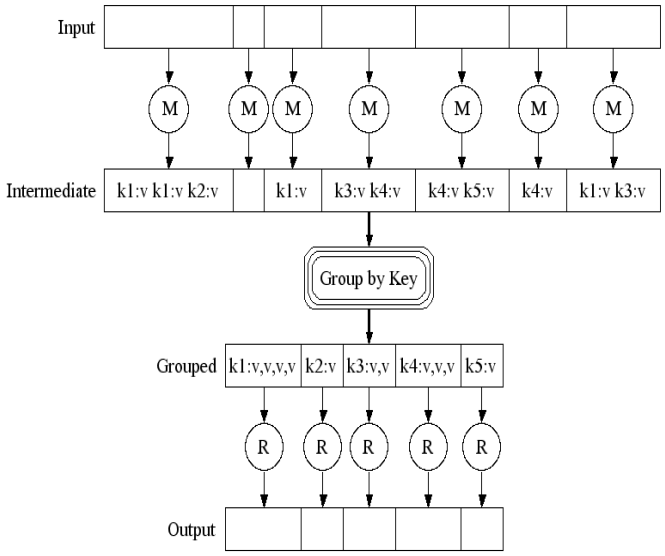

Fig. 3: Data flow in a MapReduce job

```
1  map(key1=null, value1= B:code block)
2  begin
3      extract terms list(t) from B;
4      for each term t in list(t) do
5          output(key2=t, value2=1);
6      end
7  end
8  combine(key2=t, list(value2)=list(1))
9  begin
10     var intermediateCount = sum of 1s;
11     output(key2=t, value2=intermediateCount);
12 end
13 reduce(key2=t, list(value2)=list(intermediateCount))
14 begin
15     count = sum of intermediatecount;
16     output(key3=t, value3=count);
17 end
```

Algorithm 3: Phase 1: Computing the global term frequency list

Phase II uses map reduce to sort these $\langle term, value\rangle$ pairs generated from the phase I. The map function swaps the input keys and values, a combine operation will bring together the pairs with the same key. This sorts the input pairs of the reduce function based on their frequencies. The reducer just outputs the values without keys. Algorithm 4 describes the complete steps in the second phase.

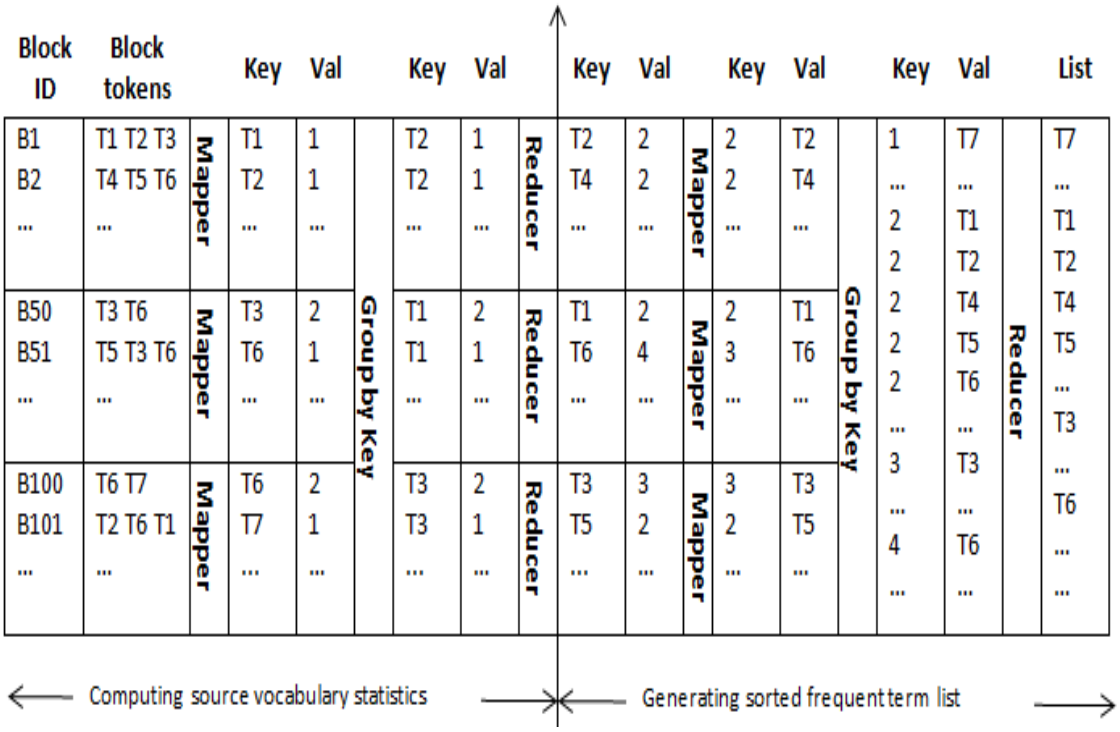

Fig. 4: Computing the global term frequency list

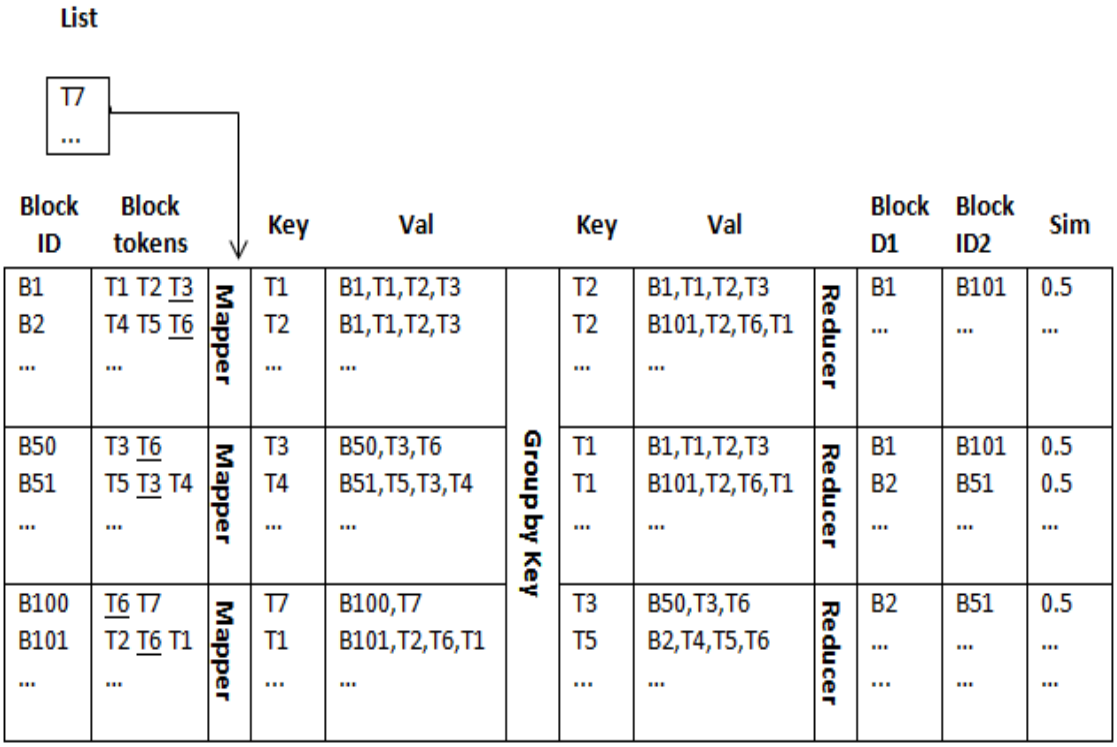

Fig. 5: Detecting the cloned code blocks

```
1  map(key1=t, value1=count)
2  begin
3  |  output(key2=count, value2=t); // Mapper
   |     only swaps the key-value pair
4  end
   /* The combiner does a group by key
      operation to bring pairs with the
      same key together. The reducer just
      splits the key=term as they are
      already in the sorted order.       */
5  reduce(key2=count, list(value2)=list(t))
6  begin
7  |  for each term t in list(t) do
8  |  |  output(key3=t, value3=null);
9  |  end
10 end
```

Algorithm 4: Phase 2: Computing the global term frequency list

4) Detecting the Cloned Code Blocks:: The computed list will act as a reference list for ordering the term in the code blocks. Next, we compute the length of the sub-block using similarity threshold and length of the block. This will enable us to create effective filters which can drastically reduce the number of block pairs (potential clones) whose similarity needs to be verified. For each code block, we define its sub-block of length n as the first n terms in the ordered block terms. If no terms are shared by the sub-blocks, that block can be avoided from further processing.

```
1  map(key1=null,value1=B:code block)
2  begin
3  |  BID ← blockID for code block B; reorder the terms
   |  in B based on term list T;
4  |  compute sub-block size (s) based on heuristic; B_sb
   |  ← sub-block of B of size (s);
5  |  for each term t in B_sb do
6  |  |  output(key2=t, value2=(BID, B));
7  |  end
8  end
9  reduce(key2=t, list(value2)=list(BID, B))
10 begin
11 |  for each (BID_1, B_1) pair in list(value2) do
12 |  |  for each (BID_2, B_2) pair in list(value2) do
13 |  |  |  compute similarity ← similarity(B_1, B_2);
14 |  |  |  if sim ≥ threshold then
15 |  |  |  |  output(k3 =
   |  |  |  |  (BID_1, BID_2, similarity), v3 = null);
16 |  |  |  end
17 |  |  end
18 |  end
19 end
```

Algorithm 5: Phase 3: Detecting cloned code blocks

Figure 5 describes the detection process. The mapper retrieves the original blocks one by one, tokenizes them and reorders the terms based on their frequencies, using the global term list as a reference. Next, it computes the sub-blocks length based on the principle described above. Terms that are not underlined represent sub-block terms. 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 code block multiple times (the number of terms in sub-block). For example, if the code block value is "T1 T2 T3 T4" and the sub-block terms are $T1$, and $T2$, we would output two $\langle key, value \rangle$ pairs, corresponding to the two sub-block terms. The reducer groups together all the values sharing same sub-block terms. In the end, a single reducer, after checking if a pair passes the similarity threshold set by the user, outputs the block pairs as clones with their similarity values. The procedure is described in Algorithm 5.

VI. Experiments

We perform two sets of experiments: (i) to evaluate the effectiveness of the filtering approach; and (ii) to measure the speed-up and scale-up of the parallel approach. Both of these experiments are described in detail below:

A. Experiment 1: Correctness and Performance of Filtering Approach

Experimental Setup: Experiment 1 was done on a laptop running OSX 10.8.5 (12f37), with an Intel Core i7, 2.3 GHz with 4 cores. The machine had 16 GB, 1600 MHz, DDR3 RAM and 500 GB of storage available.

Experiment Details and Results: The goal of this experiment is two-fold:

- To validate the correctness of the approach
- To evaluate the performance of the approach by calculating the decrease in: (i) total number of comparisons done to compute clones; and (ii) the time taken to compute clones.

In order to do that, we had created two tools: (i) NaiveCodeCloneDetector (NCCD); and (ii) FilterCodeCloneDetector(FCCD) to compute all the clones in the project. The NCCD tool does nC_2 (assuming n total blocks) comparisons whereas FCCD uses the filtering technique to reduce the number of comparisons. The granularity of a code block is a method which has more than 25 terms. Similarity function is overlap similarity with threshold θ set to 0.8. We selected five popular projects from Apache Repository as subject systems. The details of the projects including project size and the number of methods is reported in Table II.

To ensure the correctness of the approach, we compared the total number of the clones reported by each tool. Both the tools reported exactly same number of clones for all the projects (column 4, Table II). This finding confirms that the filtering approach does not miss out candidate clones by applying the heuristics. However, it only eliminates those

candidates that cannot be clones. Figure 6 compares the time taken (left) and terms compared (right) by each approach. On an average, FCCD reduces the total token comparisons by 10 times and performs 1.5 times faster than NCCD. Please note that we have not optimized FCCD in any way apart from only applying the filtering logic. Since the reduction in the number of comparisons is so high, we believe that even with minor optimizations, FCCD can perform much faster.

Validation of Clones - Precision & Recall: We manually validated a sample of 88 clone pairs from a total of 206 clone pairs reported for the Coocon project. In terms of precision, we did not find any false positive during manual validation. However, we should note that these clones were detected at a similarity threshold of 0.8. Lowering the threshold might very likely introduce false positives in the reported clones. It was not our intention to calculate recall as it is not practical to be aware of the existence of all the clones in such large systems. However, when comparing the recall of our tool implementing the prefix technique without high recall token based technique, it turned out to be the same (Column 4, Table II). Hence, the prefix technique demonstrated good precision without compromising recall.

Most of the clones we validated, fit into either intentional or unintentional clone category. Figure 7 shows an intentional clone pair, where the methods share all the terms except $startEntity$ and $endEntity$. These methods seem to provide a good abstraction of the starting and ending actions to be performed all the $entities$ in the $lexicalHandlerList$. Figure 8 shows an unintentional clone. The two methods can be easily refactored into one method by modifying the signature of any of these methods to accept an int as a parameter. The function can then be renamed appropriately.

Support for Replicating the Study: We have provided the complete output, input dataset, tools, and detailed steps to replicate the study at http://mondego.ics.uci.edu/projects/clonedetection. The web page has all the 5 subject systems, including their generated input file for clone detection. The output file contains the identified clones. The tool also generates an analysis file which produces information such as time taken to compute clones, number of clones identified, total token comparisons made, etc. The source code and binary of all the set of tools for NCCD, FCCD, and generating the input file from project are made availble along with the detailed instructions for running them.

B. Experiment 2: Speed-up and Scale-up of the Parallel Approach

We conducted Experiment 2 to measure the performance of the approach in terms of two important properties: Speed-up and Scale-up. Speedup holds the data size constant, and grows the size of the cluster. Speed-up is said to be linear, if the speed increases with the same factor as the increase in the cluster size. Scaleup measures the ability to grow both the cluster size and the data size. Scale-up is defined as the ability of an N-times larger cluster to perform an N-times larger job in the same elapsed time as the original system. The granularity of experiment was method code blocks and the similarity threshold parameter was set to 0.8.

In order to measure the Speed-up, we set up a cluster of 2-10 nodes using Amazon Web Services. Each node had the following specification: 1.7 GB memory, 160 GB storage, 1 EC2 compute unit (2 virtual core), and Hadoop 0.20. The dataset consisted of 700 open source java projects (23 MLOC, excluding comments and empty lines). This data set was a subset of the data set we used in our previous study [18]. We computed all the clones in the dataset with different cluster size. The idea was to examine the change in computation time with the increase in cluster size. The Speed-up graph in Figure 9 shows that approach has almost ideal speedup.

For measuring the Scale-up, we set up a cluster of 2-32 nodes with similar node configuration as above. We incrementally increased the number of projects from 150 to 2800 (approx. 81 MLOC) with the same proportion as we increased the number of nodes. The new projects added had a similar size distribution as the previous projects in order to make a fair evaluation. The Scale-up graph in Figure 10 shows the observed and ideal lines. Ideally, there should not be any change in time with the increase in number of projects, because we are also increasing the cluster size. However, there is increase in operational cost with the increase in cluster size - setting up the cluster, starting the mappers, and synchronizing them for the reducers. As observed, the approach showed linear scale up. We speculate that performance gain will improve with larger datasets as HDFS (Hadoop file system) performs better with large files.

Total Cost of the Experiment: We rented a total of 86 amazon small instance ec2 machines (24 to measure Speed-up and 62 to measure the Scale-up) for one hour each. The price of an instance is $0.08 per hour. Thus, we spent a total of $6.88 for a single run of the experiment. Since we ran the experiment twice, the total cost is $13.76.

VII. Related Work

This section describes related research in software clone detection which focuses on scalability and parallelization of code clone detection. A more exhaustive and broad review of code clone detection is presented in [17] and [16].

Livieri et al. [29] proposed DCCFinder[7], a tool that partitions the source code into pieces small enough to be analyzed on a single machine. Different pairs are analyzed on different machines, and the results for individual pairs are composed into a single result for the entire code base. As the number

[5]Reported time is averaged over 5 runs

[6]Hadoop is not included in the graph due to scale proportions. However, results are consistent, and are reported in Table II

[7]As per the interaction with the author, the tool is no longer maintained.

Project	Size (in KLOC)	Total Methods	Clones detected	Time Taken (in seconds)		Token Comparisons (in billions)	
				w/o filtering	w/ filtering	w/o filtering	w/ filtering
Ant	130.42	13527	587	356.43	213.04	6.43	0.62
Cocoon	59.84	5604	206	79.22	47.54	1.34	0.18
Maven	71.35	6468	934	87.12	49.93	1.53	0.16
Lucene	432.89	26627	6920	401.52	293.72	76.65	7.33
Hadoop	660.47	56017	4432	1274.81	1063.16	239.96	19.09

TABLE II: Results [5]

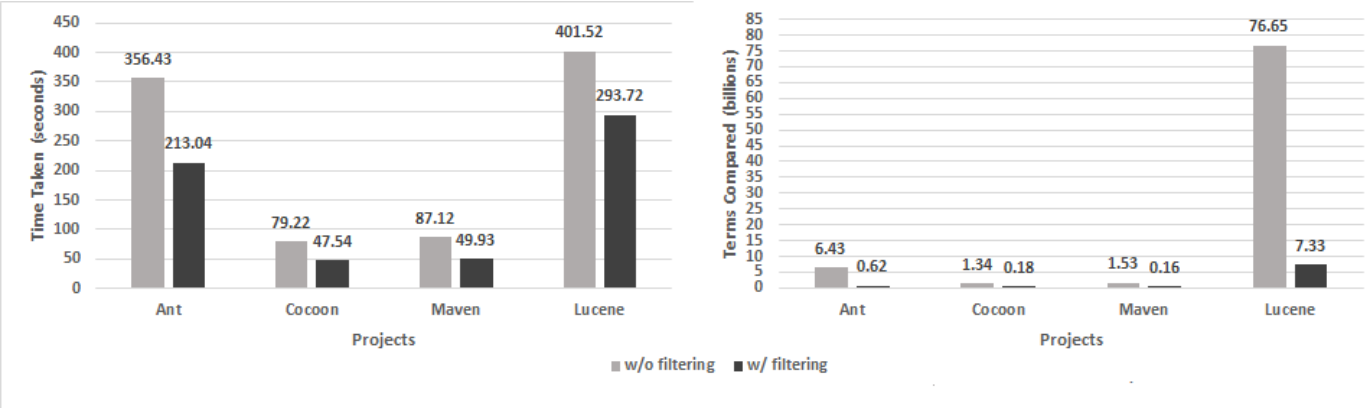

Fig. 6: Comparison: Time Taken and Terms Compared in W/ and W/O Filtering Approach[6]

```
public void startEntity(String name) throws SAXException {
    for(int i=0; i<this.lexicalHandlerList.length; i++)
        if (this.lexicalHandlerList[i] != null)
            this.lexicalHandlerList[i].startEntity(name);
}
```

```
public void endEntity(String name) throws SAXException {
    for(int i=0; i<this.lexicalHandlerList.length; i++)
        if (this.lexicalHandlerList[i] != null)
            this.lexicalHandlerList[i].endEntity(name);
}
```

Fig. 7: Intentional Clones

```
public static String getAuthority(String uri) {
    RE re = new RE(uripattern);
    if (re.match(uri)) {
        return re.getParen(4);
    } else {
        throw new IllegalArgumentException("'" + uri +
                                           "' is not a correct URI");
    }
}
```

```
public static String getPath(String uri) {
    RE re = new RE(uripattern);
    if (re.match(uri)) {
        return re.getParen(5);
    } else {
        throw new IllegalArgumentException("'" + uri +
                                           "' is not a correct URI");
    }
}
```

Fig. 8: Unintentional Clones

of pairs of pieces increases quadratically with system size, the analysis time for large systems exponentially increases the cost of computation time. In comparison to their method our approach can be easily deployed in the cloud and the proposed filtering technique can complement their approach by reducing the total number of comparisons.

Gode and Koschke [33] proposed an incremental clone detection approach based on a generalized suffix-tree. However, generalized suffix-trees are not easily distributed across different machines creating a scalability bottleneck based on the main memory.

Koschke [12] proposed and evaluated another approach to intersystem clone search using suffix trees, which avoids the creation of a costly index. To improve precision, he used decision trees based on metrics to filter the false positives. This approach is efficient when clone detection is between a subject system and a set of other systems. However for use-cases when one needs to find all the clones in the entire corpus, achieving efficient parallelism is non-trivial.

Other large scale clone detection approaches are index based. The idea is to first create an index against which the code of a subject system is compared later. The purpose of the index is to speed up the lookup and identify the code which have chances of being similar. Hummel et al. [28] were the first to propose to create an index to improve performance. In their study they demonstrated that for large systems index

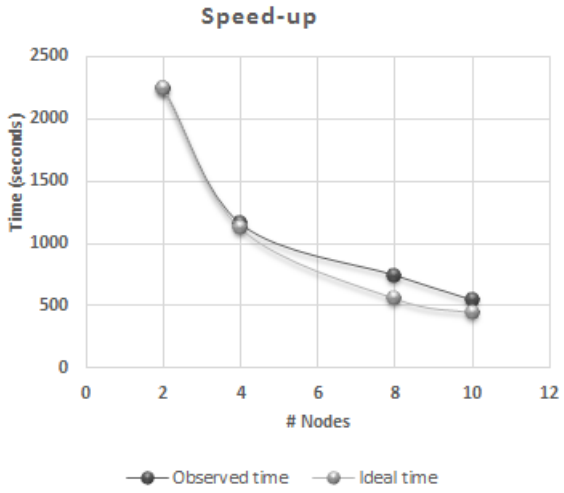

Fig. 9: Performance - Speed-up

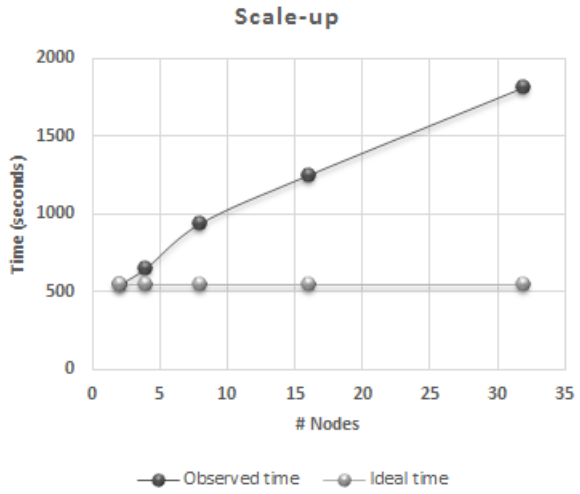

Fig. 10: Performance - Scale-up

based approach outperformed suffix based approach. In their paper, Hummel et al. describe how the index could be queried in parallel using MapReduce. This is a complex approach as all queries must be written as parallel MapReduce programs. In our approach, we demonstrate how the whole process of detection and verification can be parallelized using MapReduce without the need to store index as well.

Keivanloo et al. [24] used a similar approach and demonstrated the usefulness of the index-based approach for code search. They build up a database of hashes for 18,000 Java programs. They create hashes for three consecutive lines of code.

In general, Index-based approaches are scalable and can be distributed as demonstrated by researchers [28], [27], [24]. However, index creation is expensive and time consuming. The idea is to invest upfront in the index creation, and amortize the cost in multiple subsequent searches. Hence, index-based approaches are very suitable for use-cases like code search, but not recommended when clone detection is only a one time activity or not very frequent (e.g., our re-engineering for product lines). Moreover, the index needs to be recreated with any change in the subject systems. Nonetheless, all the index based approaches can be complemented with the filtering heuristic presented above to decrease runtime (index creation and lookup) and memory (index storage) efficiency of the approach.

VIII. Conclusion

We demonstrate a practical and inexpensive approach based on a filtering technique to improve the efficiency of code clone detection.

Our experiments show that the filtering heuristic decreases the number of token comparisons by a factor of 10 and speeds up the clone detection process by a factor of 1.5. Since the heuristic is independent of our approach, it can be adopted to improve the efficiency of any token based clone detection technique. We also present an algorithm based on the filtering heuristic to improve index-based approaches.

Our parallel algorithm efficiently scales to thousands of projects. Moreover, its MapReduce based implementation has inherent advantages like load balancing, data replication, and fault tolerance over any other in-house distributed solutions where these things are to be dealt with explicitly.

Acknowledgment

The authors would like to thank Rares Vernica for his support with the implementation, Dr. Rainer Koschke and Dr. Chanchal Roy for their valuable feedback during the course of this work. This material is based upon the work supported by the National Science Foundation under Grant No. 1018374

References

[1] J. Mayrand, C. Leblanc, and E. Merlo, "Experiment on the automatic detection of function clones in a software system using metrics," in *ICSM*. IEEE, 1996.

[2] Y. Jia, B. Binkley, M. Harman, J. Krinke, and M. Matsushita, "Kclone: A proposed approach to fast and precise clone detection," in *Proceedings of IWSC*, 2009.

[3] T. T. Nguyen, H. A. Nguyen, J. M. Al-Kofahi, N. H. Pham, and T. N. Nguyen, "Scalable and incremental clone detection for evolving software," in *Proceedings of International Conference on Software Maintenance*. IEEE Computer Society, 2009.

[4] T. Kamiya, S. Kusumoto, and K. Inoue, "CCFinder: a multilinguistic token-based code clone detection system for large scale source code," *IEEE Trans. Softw. Eng.*, vol. 28, no. 7, pp. 654–670, 2002.

[5] B. S. Baker, "On finding duplication and near-duplication in large software systems," in *Proceedings of Working Conference on Reverse Engineering*, 1995.

[6] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier, "Clone Detection Using Abstract Syntax Trees," in *Proceedings of the International Conference on Software Maintenance*. IEEE Computer Society, 1998, p. 368.

[7] R. Koschke, R. Falke, and P. Frenzel, "Clone detection using abstract syntax suffix trees," in *Working Conference on Reverse Engineering (WCRE'06)*. IEEE Computer Society, 2006.

[8] L. Jiang, G. Misherghi, Z. Su, and S. Glondu, "Deckard: Scalable and accurate tree-based detection of code clones," in *Proceedings of ICSE*, 2007.

[9] R. Komondoor and S. Horwitz, "Using Slicing to Identify Duplication in Source Code," in *Proceedings of the 8th International Symposium on Static Analysis*. Springer-Verlag, 2001, pp. 40–56.

[10] J. Krinke, "Identifying Similar Code with Program Dependence Graphs," in *Proceedings of the Eighth Working Conference on Reverse Engineering (WCRE'01)*. IEEE Computer Society, 2001, p. 301.

[11] E. Juergens, F. Deissenboeck, and B. Hummel, "Clonedetective a workbench for clone detection research," in *Proceedings of ICSE*, 2009.

[12] R. Koschke, "Large-scale inter-system clone detection using suffix trees," in *Proceedings of CSMR*, 2012, pp. 309–318.

[13] K. S. Muhammad Asaduzzaman, C. K. Roy and M. D. Penta, "Lhdiff: A language-independent hybrid approach for tracking source code lines," in *Proceedings of International Conference on Software Maintenance*. IEEE Computer Society, 2013.

[14] J. Cordy and C. Roy, "The nicad clone detector," in *Proceedings of ICPC*, 2011.

[15] S. Uddin, C. Roy, K. Schneider, and A. Hindle, "On the effectiveness of simhash for detecting near-miss clones in large scale software systems," in *Proceedings of Working Conference on Reverse Engineering*, 2011.

[16] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, and E. Merlo, "Comparison and Evaluation of Clone Detection Tools," *IEEE Trans. Softw. Eng.*, vol. 33, no. 9, pp. 577–591, 2007.

[17] 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.

[18] J. Ossher, H. Sajnani, and C. V. Lopes, "File cloning in open source java projects: The good, the bad, and the ugly." in *ICSM*. IEEE, 2011.

[19] M. Kim, V. Sazawal, D. Notkin, and G. Murphy, "An empirical study of code clone genealogies," in *Proceedings of FSE*, 2005.

[20] J. Harder and R. Tiarks, "A controlled experiment on software clones," in *Proceedings of ICPC*, 2012.

[21] D. German, M. D. Penta, Y. G. eneuc, , and G. Antoniol, "Code siblings: Technical and legal implications of copying code between applications," in *Mining Software Repositories, 2009. MSR '09. 6th IEEE International Working Conference on*, 2009.

[22] A. Hemel and R. Koschke, "Reverse engineering variability in source code using clone detection: A case study for linux variants of consumer electronic devices," in *Proceedings of Working Conference on Reverse Engineering*, 2012, pp. 357–366.

[23] J. Davies, D. German, M. Godfrey, and A. Hindle, "Software Bertillonage: finding the provenance of an entity," in *Proceedings of MSR*, 2011.

[24] I. Keivanloo, J. Rilling, and P. Charland, "Internet-scale real-time code clone search via multi-level indexing," in *Proceedings of WCRE*, 2011.

[25] S. Kawaguchi, T. Yamashina, H. Uwano, K. Fushida, Y. Kamei, M. Nagura, and H. Iida, "Shinobi: A tool for automatic code clone detection in the ide," in *Proceedings of Working Conference on Reverse Engineering*, 2009.

[26] H. Sajnani, R. Naik, and C. Lopes, "Application architecture discovery: Towards domain-driven easily extensible code structure," in *Proceedings of WCRE*, 2011.

[27] I. Keivanloo, J. Rilling, and P. Charland, "Seclone - a hybrid approach to internet-scale real-time code clone search," in *Proceedings of ICPC*, 2011.

[28] B. Hummel, E. Juergens, L. Heinemann, and M. Conradt, "Index-based code clone detection:incremental, distributed, scalable," in *Proceedings of ICSM*, 2010.

[29] 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.

[30] S.Sarawagi and A.Kirpal, "Efficient set joins on similarity predicates," in *Proceedings of SIGMOD*, 2004.

[31] W. Shang, B. Adams, and A. Hassan, "An experience report on scaling tools for mining software repositories using mapreduce," in *Proceedings of ASE*, 2010.

[32] P. Schugerl, "Scalable clone detection using description logic," in *Proceedings of International Workshop on Software Clones*, 2011.

[33] N. Gode and R. Koschke, "Incremental clone detection," in *Proceedings of CSMR*, 2009.