# Index-Based Code Clone Detection: Incremental, Distributed, Scalable

Benjamin Hummel Elmar Juergens Lars Heinemann
Technische Universität München, Germany
{hummelb,juergens,heineman}@in.tum.de

Michael Conradt
Google Germany GmbH
conradt@google.com

***Abstract*—Although numerous different clone detection approaches have been proposed to date, not a single one is both incremental and scalable to very large code bases. They thus cannot provide real-time cloning information for clone management of very large systems. We present a novel, index-based clone detection algorithm for type 1 and 2 clones that is both incremental and scalable. It enables a new generation of clone management tools that provide real-time cloning information for very large software. We report on several case studies that show both its suitability for real-time clone detection and its scalability: on 42 MLOC of Eclipse code, average time to retrieve all clones for a file was below 1 second; on 100 machines, detection of all clones in 73 MLOC was completed in 36 minutes.**

## I. Introduction

Research in software maintenance has shown that many programs contain a significant amount of duplicated (cloned) code. Such cloned code can be problematic for two reasons: (1) multiple, possibly unnecessary, duplicates of code increase maintenance costs [1], [2] and, (2) inconsistent changes to cloned code can create faults and, hence, lead to incorrect program behavior [3], [4].

In response, a large number of clone detection approaches have been proposed that can uncover duplication in large scale software systems [1], [2], [5], [6], [7], [8], [9], [10], [11], [12], [13], [14], [15], [16], [17], [18]. Their results are employed by clone visualization and management tools [19], [20], [21], [22], [23], [24], [25] to alleviate the negative consequences of cloning during maintenance of software that contains duplication. Clone management tools rely on accurate cloning information to indicate cloning relationships in the IDE while developers maintain code. To remain useful, cloning information must be updated continuously as the software system under development evolves. For this, detection algorithms need to be able to very rapidly adapt results to changing code, even for very large code bases.

Unfortunately, most existing clone detection algorithms are not sufficiently scalable and do not work incrementally but instead operate in batch mode, *i.e.*, they read the entire software system and detect all clones in a single step. When the system changes, the entire detection has to be performed again. This holds independently of the program representation they operate on and the search algorithm they employ.

If used for clone management, batch algorithms either have to be executed on-demand, or clone management has to use pre-computed results that are updated on a regular, *e.g.*, daily, basis. Both approaches cause substantial accidental complexity: on demand execution causes waiting times; pre-computed results can be outdated, causing invalid decisions or edits. This reduces the payoff achievable through the application of clone management tools and threatens their adoption by industry. The bigger the maintained code base—and thus the larger the need for automation of clone management—the greater the incurred accidental complexity, as clone detection times increase. Unfortunately, novel incremental clone detection approaches come at the cost of scalability, and vice versa. They hence do not improve clone management.

*Problem:* Current clone detection approaches are either not incremental or not scalable to very large code bases. Hence, they cannot be used for real-time detection in large systems, reducing their usefulness for clone management.

*Contribution:* This paper presents index-based clone detection as a novel clone detection approach that is both incremental and scalable to very large code bases. It extends practical applicability of clone detection to areas that were previously unfeasible since the systems were too large or since response times were unacceptably long. We outline several case studies that show the scalability of index-based clone detection to very large code bases and demonstrate that incremental updates and response times are sufficiently fast for clone management of very large software. The tool support presented in this paper is available as open source[1].

## II. State of the Art

The first part of this section introduces use cases that demonstrate the need for scalable and incremental clone detection algorithms. The second part outlines existing work in this area and discusses its shortcomings.

### A. Application of Clone Detection

Clone detection is employed for various use cases that impose different requirements on the employed algorithms.

*Evolving source code:* To alleviate the negative consequences of cloning in existing software, clone management tools employ cloning information to guide maintenance activities [19], [20], [21], [22]. A typical application is change propagation support: developers are informed that the change

[1] http://www.conqat.org/

they are currently performing is in cloned code and should possibly be applied to the clones as well. Since empirical studies have demonstrated that unintentionally inconsistent updates to cloned code often represent faults [3], such tools promise to reduce the number of cloning-related bugs. The leverage they can provide, however, depends heavily on the accuracy of the cloning information they operate on. As the software evolves, so do its contained clones. Consequently, cloning information needs to be updated continuously to remain accurate.

For large systems, rerunning detection on the entire system after each change is prohibitively expensive. Instead, clone management computes cloning information on a regular basis, *e.g.*, during a nightly build. Unfortunately, it quickly becomes out-dated once a developer changes code. With out-dated information, however, clone management tools cannot leverage their full value: clone positions can have changed, causing effort for their manual location; new clones might be missing and thus escape management.

To avoid such accidental complexity, clone detection algorithms need to be able to very quickly adapt cloning information to changes in the source code.

*Very large code bases:* For several use cases, clone detection is not employed on a single system, but on families of systems, increasing the size of the analyzed code.

Cross-product cloning analysis is used across a company's product portfolio, across a software ecosystem or for software product lines, to discover reusable code fragments that are candidates for consolidation [26]. For large products, the overall code size can be substantial. Microsoft Windows Vista, *e.g.*, comprises 50 MLOC [27].

To discover copyright infringement or license violations, clone detection is employed to discover duplication between the code base maintained by a company and a collection of open source projects or software from other parties[10], [28]. To provide comprehensive results, different versions of the individual projects might need to be included, resulting in code bases that can easily exceed several hundred MLOC. For these use cases, clone detection algorithms need to scale to very large code bases.

*Summary:* The above use cases make two important requirements for clone detection apparent: a) *incremental* detection to be able to quickly adapt cloning information to changing source code and b) *scalability* to large code bases. Ideally, a single algorithm should support both, in order to be applicable to clone management of large code bases.

### B. Existing Clone Detection Approaches

A multitude of clone detection approaches have been proposed. They differ in the program representation they operate on and in the search algorithms they employ to identify similar code fragments. Independent of whether they operate on text [7], [11], [15], tokens [5], [10], [16], ASTs [6], [12], [14] or program dependence graphs [9], [8], and independent of whether they employ textual differencing [11], [15], suffix-trees [5], [10], [16], subtree hashing [6], [14], anti-unification[29], frequent itemset mining [13], slicing [8], isomorphic subgraph search [9] or a combination of different phases [30], they operate in batch mode: the entire system is processed in a single step by a single machine.

The scalability of these approaches is limited by the amount of resources available on a single machine. The upper size limit on the amount of code that can be processed varies between approaches, but is insufficient for very large code bases. Furthermore, if the analyzed source code changes, batch approaches require the entire detection to be rerun to achieve up-to-date results. Hence, these approaches are neither incremental nor sufficiently scalable.

*Incremental or real-time detection:* Göde and Koschke [17] proposed the first incremental clone detection approach. They employ a generalized suffix-tree that can be updated efficiently when the source code changes. The amount of effort required for the update only depends on the size of the change, not the size of the code base. Unfortunately, generalized suffix-trees require substantially more memory than read-only suffix-trees, since they require additional links that are traversed during the update operations. Since generalized suffix-trees are not easily distributed across different machines, the memory requirements represent the bottleneck w.r.t. scalability. Consequently, the improvement in incremental detection comes at the cost of substantially reduced scalability.

Yamashina et al. [31] propose a tool called *SHINOBI* that provides real-time cloning information to developers inside the IDE. Instead of performing clone detection on demand (and incurring waiting times for developers), SHINOBI maintains a suffix-array on a server from which cloning information for a file opened by a developer can be retrieved efficiently. Unfortunately, the authors do not approach suffix-array maintenance in their work. Real-time cloning information hence appears to be limited to an immutable snapshot of the software. We thus have no indication that their approach works incrementally.

Nguyen et al. [32] present an AST-based incremental approach that computes characteristic vectors for all subtrees of the AST for a file. Clones are detected by searching for similar vectors. If the analyzed software changes, vectors for modified files are simply recomputed. As the algorithm is not distributed, its scalability is limited by the amount of memory available on a single machine. A related approach that also employs AST subtree hashing is proposed by Chilowicz et al. [33]. Both approaches require parsers that are, unfortunately, hard to obtain for legacy languages such as PL/I or COBOL [34]. However, such systems often contain substantial amounts of cloning [3]—making clone management for them especially relevant. Instead, our approach does not require a parser.

*Scalable detection:* Livieri et al. [35] propose a general distribution model that distributes clone detection across many machines to improve scalability. Their distribution model essentially partitions source code into pieces small enough (*e.g.*, 15 MB) to be analyzed on a single machine. Clone

detection is then performed on all pairs of pieces. Different pairs can be analyzed on different machines. Finally, results for individual pairs are composed into a single result for the entire code base. Since the number of pairs of pieces increases *quadratically* with system size, the analysis time for large systems is substantial. The increase in scalability thus comes at the cost of computation time.

*Summary:* Batch-mode clone detection is not incremental. The limited memory available on a single machine furthermore restricts its scalability. Novel incremental detection approaches come at the cost of scalability, and vice versa. In a nutshell, no existing approach is both incremental and scalable to very large code bases.

## III. Index-Based Code Clone Detection

This section introduces index-based clone detection as a novel detection approach for type 1 and 2 clones[2] that is both incremental and scalable to very large code bases.

### A. Architecture

In this section, we outline the architecture of the proposed index-based clone detection approach and point out where its components and their responsibilities differ from existing approaches.

*Clone detection pipeline:* Clone detection comprises several phases that are executed in a pipeline fashion, where each phase builds on the results of the previous phase. In our clone detection tool ConQAT [16], this architecture is made explicit through a visual data flow language that is used to configure the clone detection pipeline. From a high level perspective, we differentiate three phases, namely preprocessing, detection and postprocessing:

*Preprocessing* reads code from disk and splits it into tokens. Normalization is performed on the tokens to remove subtle differences, such as different comments or variable names. Normalization impacts both precision and recall of the clone detection results. Normalized tokens are then grouped into statements. The result of preprocessing is a list of normalized statements[3] for each file.

*Detection* searches the global statement list for equal substrings. The result of detection is hence cloning information on the level of statement sequences.

*Postprocessing* creates cloning information on the level of code regions from cloning information on the level of normalized statements. Furthermore, depending on the clone detection use case, detected clones can be filtered, presented in a project dashboard or written to a report for interactive analysis in an IDE or for use by clone management tools.

[2]Type 1 and 2 clones can differ in whitespace, commentation, identifier names and constant values [1].

[3]Clone detection can work on a list of tokens as well, but working on the list of statements typically gives higher precision. In the remainder we will use the term statement, but all of the algorithms work for tokens as well.

*Performance properties:* The tasks performed during preprocessing (reading from disk, scanning and normalization) are linear in time and space w.r.t. the size of the input files. Furthermore, preprocessing is trivially parallelizable, since individual files can be processed independently. The bottleneck that determines clone detection performance, particularly scalability and the ability to perform incremental updates, is hence the algorithm employed in the *detection* phase.

Index-based clone detection, consequently, represents a novel approach for the detection phase. Both pre- and postprocessing components, in contrast, are not novel but reused from our state of the art clone detector ConQAT. The clones detected by the index-based approach are identical to those detected by our existing suffix-tree-based algorithm—but, importantly, performance in terms of scalability and ability to perform incremental updates is improved significantly.

### B. Clone Index

This section describes the *clone index*, the central data structure used for our detection algorithm. It allows the lookup of all clones for a single file (and thus also for the entire system), and can be updated efficiently, when files are added, removed, or modified.

The list of all clones of a system is not a suitable substitute for a clone index, as efficient update is not possible. Adding a new file may potentially introduce new clones to any of the existing files and thus a comparison to all files is required if no additional data structure is used.

The core idea of the clone index is similar to the inverted index used in document retrieval systems (*c.f.*, [36], pp. 560–663). There, a mapping from each word to all its occurrences is maintained. Similarly, the clone index maintains a mapping from sequences of normalized statements to their occurrences. More precisely, the clone index is a list of tuples *(file, statement index, sequence hash, info)*, where

- **file** is the name of the file,
- **statement index** is an integer denoting the position in the list of normalized statements for the file,
- **sequence hash** is a hash code for the next $n$ normalized statements in the file starting from the *statement index* ($n$ is a constant called *chunk length* and is usually set to the minimal clone length), and
- **info** contains any additional data, which is not required for the algorithms, but might be useful when producing the list of clones, such as the start and end lines of the statement sequence.

The clone index contains the described tuples for all files and all possible statement indices, *i.e.*, for a single file the statement sequences $(1, \ldots, n)$, $(2, \ldots, (n+1))$, $(3, \ldots, (n+2))$, etc. are stored. Our detection algorithm requires lookups of tuples both by file and by sequence hash, so both should be supported efficiently. Other than that, no restrictions are placed on the index data structure, so there are different implementations possible, depending on the actual use-case (*c.f.*, Section IV). These include in-memory indices based on two hash tables or search trees for the lookups, and disk-based

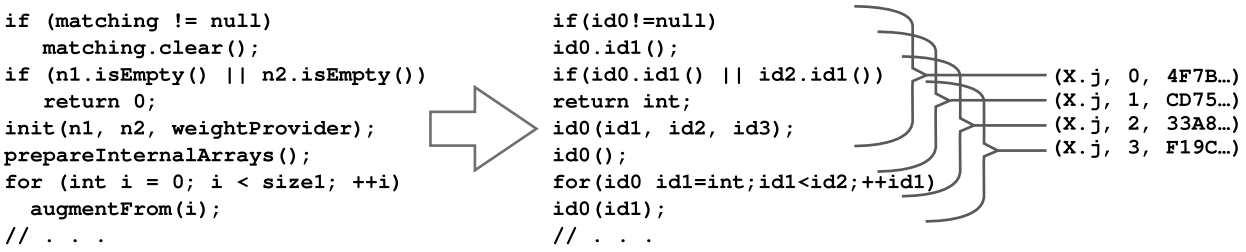


Fig. 1. The original file (left), its normalization (center), and the corresponding clone index (right).

indices which allow persisting the clone index over time and processing amounts of code which are too large to fit into main memory. The latter may be based on database systems, or on one of the many optimized (and often distributed) key-value stores [37], [38].

In Fig. 1, the correspondence between an input file "X.j"[4] and the clone index is visualized for a chunk length of 5. The field which requires most explanation is the *sequence hash*. The reason for using statement sequences in the index instead of individual statements is that the statement sequences are more unique (two identical statement sequences are less likely than two identical statements) and are already quite similar to the clones. If there are two entries in the index with the same sequence, we already have a clone of length at least $n$. The reason for storing a hash in the index instead of the entire sequence is for saving space, as this way the size of the index is independent of the choice of $n$, and usually the hash is shorter than the sequence's contents even for small values of $n$. We use the MD5 hashing algorithm [39] which calculates 128 bit hash values and is typically used in cryptographic applications, such as the calculation of message signatures. As our algorithm only works on the hash values, several statement sequences with the same MD5 hash value would cause false positives in the reported clones. While there are cryptographic attacks which can generate messages with the same hash value [40], the case of different statement sequences producing the same MD5 hash is so unlikely in our setting, that it can be neglected for practical purposes.

### C. Clone Retrieval

Clone retrieval is the process of extracting all clones for a single file from the index. Usually we assume that the file is contained in the index, but of course the same process can be applied to find clones between the index and an external file as well. Tuples with the same sequence hash already indicate clones with a length of at least $n$ (where $n$ is the globally constant chunk length). The goal of clone retrieval is to report only maximal clones, *i.e.*, clone groups which are not completely contained in another clone group. The overall algorithm, which calculates the longest subsequences of duplicated chunks, is sketched in Fig. 2 and explained in more detail next.

The first step (up to Line 6) is to create the list $c$ of duplicated chunks. This list stores for each statement of the input file all tuples from the index with the same sequence hash as the sequence found in the file. The index used to access the list $c$ corresponds to the statement index in the input file. The setup is depicted in Fig. 3. There is a clone in X.j of length 10 (6 tuples with chunk length 5) with the file Y.j, and a clone of length 7 with both Y.j and Z.j.

In the main loop (starting from Line 7), we first check whether any new clones might start at this position. If there is only a single tuple with this hash (which has to belong to the inspected file at the current location) we skip this loop

[4]We use the name X.j instead of X.java as an abbreviation in the figures.

```
 1: function reportClones (filename)
 2:     let f be the list of tuples corresponding to filename
            sorted by statement index either read from
            the index or calculated on the fly
 3:     let c be a list with c(0) = ∅
 4:     for i := 1 to length(f) do
 5:         retrieve tuples with same sequence hash as f(i)
 6:         store this set as c(i)
 7:     for i := 1 to length(c) do
 8:         if |c(i)| < 2 or c(i) ⊆̃ c(i − 1) then
 9:             continue with next loop iteration
10:         let a := c(i)
11:         for j := i + 1 to length(c) do
12:             let a′ := a ∩̃ c(j)
13:             if |a′| < |a| then
14:                 report clones from c(i) to a (see text)
15:             a := a′
16:             if |a| < 2 or a ⊆̃ c(i − 1) then
17:                 break inner loop
```

Fig. 2. Clone Retrieval Algorithm

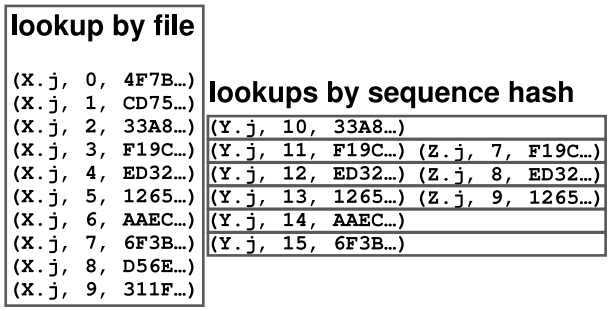


Fig. 3. Lookups performed for retrieval

iteration. The same holds if all tuples at position $i$ have already been present at position $i-1$, as in this case any clone group found at position $i$ would be included in a clone group starting at position $i-1$. Although we use the subset operator in the algorithm description, this is not really a subset operation, as of course the *statement index* of the tuples in $c(i)$ will be increased by 1 compared to the corresponding ones in $c(i-1)$ and the *hash* and *info* fields will differ.

The set $a$ introduced in Line 10 is called the *active set* and contains all tuples corresponding to clones which have not yet been reported. At each iteration of the inner loop the set $a$ is reduced to tuples which are also present in $c(j)$; again the intersection operator has to account for the increased statement index and different hash and info fields. The new value is stored in $a'$. Clones are only reported, if tuples are lost in Line 12, as otherwise all current clones could be prolonged by one statement. Clone reporting matches tuples that, after correction of the statement index, appear in both $c(i)$ and $a$; each matched pair corresponds to a single clone. Its location can be extracted from the filename and info fields. All clones in a single reporting step belong to one clone group. Line 16 early exits the inner loop if either no more clones are starting from position $i$ (*i.e.*, $a$ is too small), or if all tuples from $a$ have already been in $c(i-1)$, corrected for statement index. In this case they have already been reported in the previous iteration of the outer loop.

This algorithm returns all clone groups with at least one clone instance in the given file and with a minimal length of chunk length $n$. Shorter clones cannot be detected with the index, so $n$ must be chosen equal to or smaller than the minimal clone length (typically 7 or 10). Of course, reported clones can be easily filtered to only include clones with a length $l > n$.

One problem of this algorithm is that clone classes with multiple instances in the same file are encountered and reported multiple times. Furthermore, when calculating the clone groups for all files in a system, clone groups will be reported more than once as well. Both cases can be avoided, by checking whether the first element of $a'$ (with respect to a fixed order) is equal to $f(j)$ and only report in this case.

*Complexity:* For the discussion of complexity we denote the number of statements by $|f|$ and the number of tuples returned for its $i$-th statement by $|c(i)|$ (just as in the algorithm). The number of tuples in the index is denoted by $N$. For the first part of the algorithm the number of queries to the index is exactly $|f|+1$. Assuming that a single query returning $q$ elements can be performed in $O(q+\log N)$, which is true for typical index implementations, the first part up to Line 6 requires at most time $O(\sum_i |c(i)| + |f| \log N)$.

The set operations used in the algorithm are easily implemented in linear time if the sets $c(i)$ are managed as sorted lists. Thus, the running time of the part starting from Line 7 is bounded by $O(|f|^2 \max c(i))$, which seems to be rather inefficient. It should be noted, however, that the worst-case is hard to construct and nearly never appears in real-world systems. For both the case that a file contains not a single clone (*i.e.*, $|c(i)| = 1$ for all $i$), and that an exact duplicate exists for a file but no clone to other files, the runtime of this part improves to $O(|f|)$. As the overall performance of the clone retrieval algorithm strongly depends on the structure of the analyzed system, practical measurements are important, which are reported on in Sec. IV.

### D. Index Maintenance

By the term *index maintenance* we understand all steps required to keep the index up to date in the presence of code changes. For index maintenance, only two operations are needed, namely *addition* and *removal* of single files. Modifications of files can be reduced to a *remove* operation followed by an *addition*[5] and index creation is just addition of all existing files starting from an empty index. In the index-based model, both operations are extremely simple. To add a new file, it has to be read and preprocessed to produce its sequence of normalized statements. From this sequence, all possible contiguous sequences of length $n$ (where $n$ is the chunk length) are generated, which are then hashed and inserted as tuples into the index. Similarly, the removal of a file consists of the removal of all tuples which contain the respective file. Depending on the implementation of the index, addition and removal of tuples can cause additional processing steps (such as rebalancing search trees, or recovering freed disk space), but these are not considered here.

We may assume that preprocessing time of a file is linear in its size[6]. Depending on the index structure used, addition and removal of single tuples typically requires expected amortized processing time $O(1)$ (*e.g.*, for hash tables), or $O(\log N)$ where $N$ is the number of stored tuples (*e.g.*, for search trees). Thus, the index maintenance operations can be expected to run in linear time or time $O(|f| \log N)$.

### E. Implementation Considerations

There are two factors affecting the processing time required for index maintenance and clone retrieval: the chunk size $n$, and the implementation of the clone index. The size of $n$ affects both index maintenance, which is more efficient for small $n$, and clone retrieval, which benefits from larger $n$. Additionally, $n$ has to be chosen smaller than the minimal clone length, as only clones consisting of at least $n$ statements can be found using the index. For practical purposes we usually choose $n$ smaller than or equal to the minimal clone length we are interested in. Values of $n$ smaller than 5 typically lead to a large number of tuples with the same hash value an thus severely affect clone retrieval.

The index implementation is affected both by the choice of the data structure and the kind of storage medium used. As

[5]This simplification makes sense only if a single file is small compared to the entire code base, which holds for most systems. If a system only consists of a small number of huge files, more refined update operations would be required.

[6]The preprocessing phase of most token-based clone detectors runs in linear time, as both tokenization and normalization with constant look-ahead are linear time operations.

long as the system is small enough to fit into main memory, this is of course the best medium to use in terms of access and latency times. If the system's size exceeds a certain threshold or the index should be persisted and maintained over a longer period of time, disk-based implementations are preferable. Finally, for distributed operation, network-based distributed index implementations are used. The data structure should be chosen based on the mode of operation.

If the index will not live long enough to require any changes, no index maintenance is required. The detection of clones can be split into separate index creation (possibly followed by an index optimization phase) and clone retrieval phases. Often this allows the use of more compact or efficient data structures. Examples of different index implementations will be presented in the next section.

## IV. Case Studies

This section summarizes the results from three case studies. Section IV-A focusses on the batch detection use-case and compares the results with our existing implementation of a suffix-tree-based detector [16]. Section IV-B investigates clone detection for very large code bases by distribution to multiple machines. Finally, Section IV-C presents the use-case of a continuously updated clone index that provides real time clone detection information. Depending on the use-case the algorithm has to be adapted by using a suitable index implementation, which is described in the corresponding section. For all experiments a minimal clone length of 10 statements[7] was chosen. The index implementation used is described in each of the sub sections.

As the algorithm proposed in this paper is exact[8], *i.e.*, reports all exact duplicates in the normalized statements, we omit a discussion of precision and recall. Both are only affected by the normalization phase (and possibly post-processing), which is outside the scope of the paper.

The case studies in Section IV-A and Section IV-C were performed on a Mac with a Linux operating system, an Intel Core 2 Duo CPU with 2.4 GHz, and 4 GB of RAM. The case study for the distributed clone detection was conducted on Google's computing infrastructure.

### A. Batch Clone Detection

In this case study, we show that the index-based approach detects the same clones as our suffix-tree-based implementation and compare execution times.

*Index implementation:* We utilized an in-memory clone index implementation that is based on a simple list which is only sorted once after the creation of the index. This is possible as we are in a batch setting where the index is not changed after creation. The sorted list allows (amortized) constant time insertions, as sorting is only performed in the end, and logarithmic query time using binary search.

[7]This minimal clone length provides, according to our experiences, a good trade-off between precision and recall.

[8]The only possible source of error is the MD5 hashing. Our first case study (Sec. IV-A) shows that even for several million lines of source code our algorithms yields the same results as an exact suffix-tree-based algorithm. Thus we conclude that hash collisions are so unlikely in practice, that we can consider the algorithm exact.

TABLE I
Analyzed systems

| | Jabref | Commercial | Linux Kernel |
|---|---|---|---|
| Version | 2.6 | n/a | 2.6.33.2 |
| Language | Java | ABAP | C |
| Files | 562 | 2,631 | 25,663 |
| Lines of code | 114,887 | 460,730 | 11,250,148 |
| Clones | 419 | 12,769 | 60,353 |
| Clone classes | 160 | 2,625 | 21,212 |

TABLE II
Execution time (batch CD)

| | Jabref | Commercial | Linux Kernel |
|---|---|---|---|
| Suffix-tree | 7.3 sec | 28.7 sec | 166 min 13 sec |
| Index-based | 6.7 sec | 28.7 sec | 47 min 29 sec |

*Design and Procedure:* We used two open source and one commercial software system (called *Commercial* for non-disclosure reasons) as study objects. Overview information is shown in Table I. We chose systems developed in different programming languages and from different organizations to increase the transferability of the results.

To compare the execution time of both approaches, we set up two clone detection configurations for ConQAT which used the same normalization. The only difference was the actual clone detection algorithm: the first configuration used our suffix-tree-based detection algorithm, whereas the other used the index-based implementation. We used our existing suffix-tree-based implementation for the comparison, to assure that the exact same clones are found and therefore allow for comparability of execution times. We ensured that the algorithms detected the exact same clones by comparing automatically generated textual reports of all clones.

*Results:* The comparison of the files with all reported clones revealed that both algorithms yielded exactly the same clones. Table II illustrates the execution time of the index-based approach compared to the suffix-tree-based clone detection. The time for the index-based approach consists of the time required to build the index and the time to retrieve all clones. For Jabref and Commercial, we took the fastest execution of several runs, to eliminate measurement errors due to background processes executed on the machine.

*Discussion:* For each of the analyzed systems, the index-based algorithm is as fast or faster than the suffix-tree-based algorithm. For the largest system, it is more than three times faster. As the suffix-tree-based algorithm runs in worst-case linear time and the index-based one in *expected* linear time (if the system consists of sufficiently many files), we account the difference in speed to the constants involved. Especially the simpler data structures used in the index-based algorithm might use the memory hierarchy more efficiently, which

especially shows for larger code sizes. These results indicate that index-based detection could, in principle, substitute suffix-tree-based detection, which is employed by many existing tools [10], [17], [12], [16], [5].

### B. Distributed Clone Detection

To evaluate the scalability of our algorithm to large code bases and distribution on multiple machines, we performed a case study using Google's computing infrastructure.

*Index implementation:* We used an index implemented on top of Bigtable [37], a key-value store supporting distributed access and storage. To reduce network overhead, composite operations are used for index access.

*Design and Procedure:* Google's MapReduce[9] infrastructure [41] is used as distribution mechanism. We implemented two separate MapReduce programs. Both use the map phase to distribute the detection subtasks for the individual files; the reduce phase is skipped.

The first MapReduce program constructs the clone index and stores it in a Bigtable. As the addition of different files to the index is completely independent, it can be easily parallelized. The mapper receives the names and contents of the files, performs preprocessing and normalization, and stores the resulting tuples in the index.

The second one retrieves all clones and calculates the clone coverage[10] for all files in the index. Calculation of the clone coverage for individual files is completely independent once the index has been calculated. Thus, again the mapper retrieves the names of files to be processed, calculates the clone coverage value, and stores it in the Bigtable.

We applied the detection to third party open source software, including, *e.g.*, WebKit, Subversion, and Boost. In total, detection processed 73.2 MLOC of Java, C, and C++ code in 201,283 files. To evaluate scalability, we executed both index creation and coverage calculation as separate jobs, both on different numbers of machines[11].

In addition, to evaluate scalability to ultra-large code bases, we measured the runtime of the index construction job on about 120 million C/C++ files from the code indexed by Google Code Search[12] on 1000 machines. The overall amount of code processed accumulates to 2.9 GLOC[13]. The only relevant use case we see for such amounts of code is copyright infringement analysis, for which only cloning between a single project against the large index is relevant. We thus did not attempt to detect all clones for it.

[9]MapReduce is a technique for executing processing and aggregation tasks on large input sets on a cluster of commodity hardware both efficiently and reliably.

[10]The clone coverage is the fraction of statements of a file (or system) which are contained in at least one clone and thus can be interpreted as the probability, that a change to an arbitrary code location affects a clone and thus potentially has to be replicated at another place.

[11]The machines used have Intel Xeon processors from which only a single core was used, and the task allocated about 3 GB RAM on each.

[12]http://www.google.com/codesearch

[13]More precisely these are 2,915,947,163 lines of code.

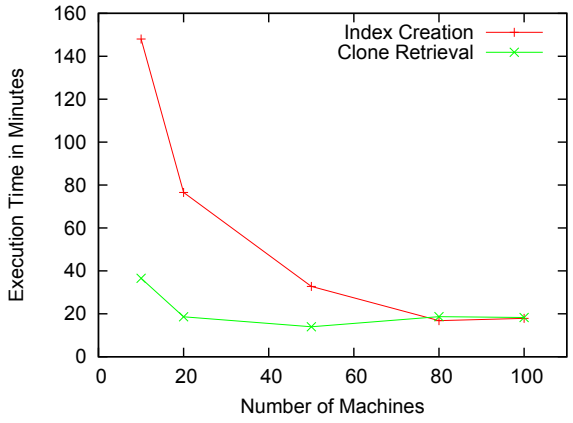


Fig. 4. Execution time (dist. CD)

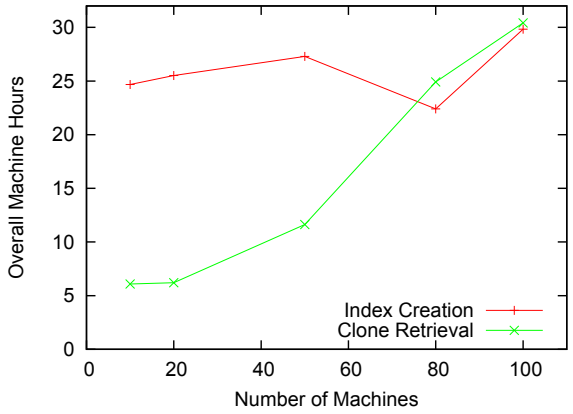


Fig. 5. Overall machine time (dist. CD)

*Results:* Using 100 machines, index creation and coverage computation for the 73.2 MLOC of code together took about 36 minutes. For 10 machines, the processing time is still only slightly above 3 hours. The times for index creation and clone retrieval are depicted in Fig. 4 for 10, 20, 50, 80, and 100 machines. The plot in Fig. 5 shows the overall machine time spent, which is simply the product of the running time multiplied by the number of machines.

The creation of the clone index for the 2.9 GLOC of C/C++ sources in the Google Code Search index required less than 7 hours on 1000 machines.

*Discussion:* As the cluster is concurrently used for other tasks, the measured times vary based on its overall load. This, however, can only increase the required processing time, so the reported numbers may be interpreted as a worst-case when running on an isolated cluster. The second observation is the saturation of the execution time curve, which is especially visible for the clone retrieval task. MapReduce distributes work as large processing blocks (in our case lists of files to be processed). Towards the end of the job, most machines are usually waiting for a small number of machines which had a slightly larger computing task. Large files or files with

many clones can cause some of these file lists to require substantially longer processing times, causing this saturation effect. The algorithm thus scales well up to a certain number of machines. It seems that using more than about 30 machines for *retrieval* does not make sense for a code base of the given size. However, the large job processing 2.9 GLOC demonstrates the (absence of) limits for the *index construction* part.

Since a substantial part of the time is spent on network communication, these times may not be compared with those of the previous section. Instead, a better comparison would be with [35], where about 400 MLOC were processed on 80 machines within 51 hours. While our code base is about a factor of 5.5 smaller, using 80 machines, index-based detection is 86 times faster. Using only ten machines, it is still a factor of 16.5 faster (on the smaller code base). For a conclusive comparison, we would have to run both approaches on the same code and hardware. However, these numbers indicate that the gained speedup is not only due to reduced code size or the use of slightly different hardware, but also because of algorithmic improvement.

### C. Real Time Clone Detection

This case study investigates suitability for real-time clone detection on large code that is modified continuously.

*Index implementation:* We used a persistent clone index implementation based on Berkeley DB[14], a high-performance embedded database.

*Design and Procedure:* To evaluate the support for real time clone management, we measured the time required to (1) build the index, (2) update the index in response to changes to the system, and (3) query the index. For this, we used code of version 3.3 of the Eclipse SDK consisting of 209.312 Java files which comprise 42.693.793 lines of code. Since our approach is incremental, a full indexing has to happen only once. To assess the performance of the index building, we constructed the index for the Eclipse source code and measured the overall time required. Furthermore, to evaluate the incremental update capabilities of the index, we removed 1,000 randomly selected files from the Eclipse index and re-added them afterwards. Likewise, for the query capability, we queried the index for the clone classes of 1,000 randomly selected files.

*Results:* On the test system, the index creation process for the Eclipse SDK including writing the clone index to the database took 7 hours and 4 minutes. The clone index for the Eclipse SDK occupied 5.6 GB on the hard disk. In comparison, the source code itself needed 1.8 GB disk space, *i.e.*, the index used about 3 times the space of the original system. The test system was able to answer a query for a single file at an average time of 0.91 seconds, and a median query time of 0.21 seconds. Only 14 of the 1000 files had a query time of over 10 seconds. On average they had a size of 3 kLOC and 350 clones. The update of the clone index including writing it to the database took 0.85 seconds on average per file. Table III illustrates these performance measurement results at a glance.

TABLE III
CLONE MANAGEMENT PERFORMANCE

| | |
|---|---|
| Index creation (complete) | 7 hr 4 min |
| Index query (per file) | 0.21 sec median<br>0.91 sec average |
| Index update (per file) | 0.85 sec average |

*Discussion:* The results from this case study indicate that our approach is capable of supporting real time clone management. The average time for an index query is, in our opinion, fast enough to support interactive display of clone information when a source file is opened in the IDE. Furthermore, the performance of index updates allows for continuous index maintenance.

## V. CONCLUSION AND FUTURE WORK

This paper presents a novel clone detection approach that employs an index for efficient clone retrieval. To the best of our knowledge, it is the first approach that is at the same time incremental and scalable to very large code bases.

The clone index not only allows the retrieval of all clones contained in a system, but also supports the efficient retrieval of the clones that cover a specific file. This allows clone management tools to provision developers in real-time with cloning information while they maintain code. For a case study on 42 MLOC of Eclipse, average retrieval time was below 1 second, demonstrating applicability to very large software. Since the clone index can be updated incrementally while the software changes, cloning information can be kept accurate at all times.

The clone index can be distributed across different machines, enabling index creation, maintenance and clone retrieval to be parallelized. Index-based clone detection thus imposes no limits on scalability—it can simply be improved by adding hardware resources—while retaining its ability for incremental updates. For the case study, 100 machines performed clone detection in 73 MLOC of open source code in 36 minutes.

For the analyzed systems, index-based clone detection (employing an in-memory index) outperforms suffix-tree-based clone detection. This indicates that the index-based detection algorithm can potentially substitute suffix-tree-based algorithms, which are employed by many existing clone detectors.

For future work, we plan to develop algorithms that employ the clone index to detect type 3 clones[15]. One approach is to use hash functions that are robust to further classes of changes, such as *e.g.*, n-gram based hashing as proposed by Smith and Horwitz in [42]. Another approach is to use locality sensitive hashing, as *e.g.*, employed by [14] and adapt the retrieval to also find similar, not only identical, hashes. Furthermore, based

[14]http://www.oracle.com/technology/products/berkeley-db/index.html

[15]Type 3 clones can differ beyond the token level; statements can be inserted, changed or removed [1].

on our existing tool support for clone detection in natural language documents [43], we plan to employ the index-based approach for plagiarism detection in text documents.

Our implementation of index-based clone detection (except the Google-specific parts) is available as open source.

## ACKNOWLEDGMENTS

The authors want to thank Birgit Penzenstadler, Florian Deissenboeck and Daniel Ratiu for inspiring discussions and helpful comments on the paper.

## REFERENCES

[1] R. Koschke, "Survey of research on software clones," in *Duplication, Redundancy, and Similarity in Software*, 2007.

[2] C. K. Roy and J. R. Cordy, "A survey on software clone detection research," Queen's University, Kingston, Canada, Tech. Rep. 2007-541, 2007.

[3] E. Juergens, F. Deissenboeck, B. Hummel, and S. Wagner, "Do code clones matter?" in *ICSE'09*, 2009.

[4] T. Bakota, R. Ferenc, and T. Gyimothy, "Clone smells in software evolution," in *ICSM'07*, 2007.

[5] B. S. Baker, "On finding duplication and near-duplication in large software systems," in *WCRE'95*, 1995.

[6] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier, "Clone detection using abstract syntax trees," in *ICSM'98*, 1998.

[7] S. Ducasse, M. Rieger, and S. Demeyer, "A language independent approach for detecting duplicated code," in *ICSM'99*, 1999.

[8] R. Komondoor and S. Horwitz, "Using slicing to identify duplication in source code," in *SAS'01*, 2001.

[9] J. Krinke, "Identifying similar code with program dependence graphs," in *WCRE'01*, 2001.

[10] 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.

[11] J. R. Cordy, T. R. Dean, and N. Synytskyy, "Practical language-independent detection of near-miss clones," in *CASCON'04*, 2004.

[12] R. Koschke, R. Falke, and P. Frenzel, "Clone detection using abstract syntax suffix trees," in *WCRE'06*, 2006.

[13] Z. Li, S. Lu, S. Myagmar, and Y. Zhou, "CP-Miner: Finding copy-paste and related bugs in large-scale software code," *IEEE Trans. Softw. Eng*, vol. 32, no. 3, pp. 176–192, 2006.

[14] L. Jiang, G. Misherghi, Z. Su, and S. Glondu, "DECKARD: Scalable and accurate tree-based detection of code clones," in *ICSE'07*, 2007.

[15] C. K. Roy and J. R. Cordy, "NICAD: Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization," in *ICPC'08*, 2008.

[16] E. Juergens, F. Deissenboeck, and B. Hummel, "Clonedetective – a workbench for clone detection research," in *ICSE'09*, 2009.

[17] N. Göde and R. Koschke, "Incremental clone detection," in *CSMR'09*, 2009.

[18] J. Mayrand, C. Leblanc, and E. Merlo, "Experiment on the automatic detection of function clones in a software system using metrics," in *ICSM'96*, 1996.

[19] R. Koschke, "Frontiers of software clone management," in *FoSM'08*, 2008.

[20] P. Jablonski and D. Hou, "CReN: a tool for tracking copy-and-paste code clones and renaming identifiers consistently in the IDE," in *Eclipse'07*, 2007.

[21] E. Duala-Ekoko and M. P. Robillard, "Tracking code clones in evolving software," in *ICSE'07*, 2007.

[22] E. Juergens and F. Deissenboeck, "How much is a clone?" in *SQM'10*, 2010.

[23] M. Rieger, S. Ducasse, and M. Lanza, "Insights into system-wide code duplication," in *WCRE'04*, 2004.

[24] M. de Wit, A. Zaidman, and A. van Deursen, "Managing code clones using dynamic change tracking and resolution," in *ICSM'09*, 2009.

[25] C. J. Kapser and M. W. Godfre, "Improved tool support for the investigation of duplication in software," in *ICSM'05*, 2005.

[26] T. Mende, F. Beckwermert, R. Koschke, and G. Meier, "Supporting the grow-and-prune model in software product lines evolution using clone detection," in *CSMR'08*, 2008.

[27] N. Nagappan, B. Murphy, and V. Basili, "The influence of organizational structure on software quality: an empirical case study," in *ICSE'08*, 2008.

[28] D. German, M. Di Penta, Y. Guéhéneuc, and G. Antoniol, "Code siblings: Technical and legal implications of copying code between applications," in *MSR'09*, 2009.

[29] P. Bulychev and M. Minea, "An evaluation of duplicate code detection using anti-unification," in *IWSC'09*, 2009.

[30] Y. Jia, D. Binkley, M. Harman, J. Krinke, and M. Matsushita, "KClone: a proposed approach to fast precise code clone detection," in *IWSC'09*, 2009.

[31] 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 *WCRE'09*, 2009.

[32] 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," *ICSM'09*, 2009.

[33] M. Chilowicz, É. Duris, and G. Roussel, "Syntax tree fingerprinting for source code similarity detection," in *Proc. of ICPC'09*, 2009.

[34] R. Lämmel and C. Verhoef, "Semi-automatic grammar recovery," *Softw. Pract. Exp.*, vol. 31, no. 15, pp. 1395–1438, 2001.

[35] 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 *ICSE'07*, 2007.

[36] D. E. Knuth, *The Art of Computer Programming*, 2nd ed. Addison-Wesley, 1997, vol. 3: Sorting and Searching.

[37] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, "Bigtable: A distributed storage system for structured data," *ACM Trans. Comput. Syst.*, vol. 26, no. 2, 2008.

[38] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: Amazon's highly available key-value store," in *SOSP'07*, 2007.

[39] R. Rivest, "The MD5 Message-Digest Algorithm," RFC 1321 (Informational), Internet Engineering Task Force, 1992.

[40] M. Stevens, A. Sotirov, J. Appelbaum, A. K. Lenstra, D. Molnar, D. A. Osvik, and B. de Weger, "Short chosen-prefix collisions for MD5 and the creation of a rogue CA certificate," in *CRYPTO'09*, 2009.

[41] J. Dean and S. Ghemawat, "MapReduce: a flexible data processing tool," *Commun. ACM*, vol. 53, no. 1, pp. 72–77, 2010.

[42] R. Smith and S. Horwitz, "Detecting and measuring similarity in code clones," in *Proc. of IWSC'09*, 2009.

[43] E. Juergens, F. Deissenboeck, M. Feilkas, B. Hummel, B. Schaetz, S. Wagner, C. Domann, and J. Streit, "Can clone detection support quality assessments of requirements specifications?" in *ICSE'10*, 2010.