Instant Code Clone Search

Mu-Woong Lee Jong-Won Roh Seung-won Hwang
Pohang University of Science and Technology (POSTECH)
Republic of Korea
{sigliel, nbanoh, swhwang}@postech.edu

Sunghun Kim
Hong Kong University of
Science and Technology (HKUST)
hunkim@cse.ust.hk

ABSTRACT

In this paper, we propose a scalable instant code clone search engine for large-scale software repositories. While there are commercial code search engines available, they treat software as text and often fail to find semantically related code. Meanwhile, existing tools for semantic code clone searches take a "post-mortem" approach involving the detection of clones "after" the code development is completed, and hence, fail to return the results instantly. In clear contrast, we combine the strength of these two lines of existing research, by supporting instant code clone detection. To achieve this goal, we propose scalable indexing structures on vector abstractions of code. Our proposed algorithms allow developers to detect clones of a given code segment among the 1.7 million code segments from 492 open source projects in sub-second response times, without compromising the accuracy obtained by a state-of-the-art tool.

Categories and Subject Descriptors

D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement—*Restructuring, reverse engineering, and reengineering*

General Terms

Design, Management

Keywords

Clone detection, code search

1. INTRODUCTION

Clone detection helps software development and maintenance tasks, as unmanaged code clones make program maintenance difficult and may cause *inconsistent clone changes* [13, 18]. Therefore, clone detection research has been an active area for decades, and many practical techniques have been proposed and widely used [2, 11, 15, 24, 26, 29].

Existing code clone detection tools usually take a *post-mortem* approach by detecting the clones "after" code development is completed, as they mainly focus on the *code refactoring* scenario. In such scenario, developers, for example, may run a code clone detector once per month, and based on the gathered clone information, they perform any necessary maintenance work such as refactoring or fixing inconsistent clone changes. In clear contrast, we focus on developing a *preventive* way of finding clones during development process, by enabling an instant and scalable search for the clones of a given code segment. This type of *instant clone detection* encourages to refer to well-tested existing code, rather than reinvents code clones.

In this paper, we propose an instant clone search engine that is scalable to the size of code repositories, *e.g.*, detecting the clones of a given code segment from 492 open source projects (54 million LOC, 1.7 million code segments) in a sub-second response time. Our detector considers structural similarities between code segments, as in a post-mortem clone detection tool, DECKARD [11]. Our key technical contribution is balancing the dual goals of instant response and result quality, by exploiting a multidimensional indexing structure, `R*tree` [3], and proposing dimensionality reduction and I/O optimization techniques.

This type of instant detector would enable many interesting applications. For example, when developers work on one section of a very large project, instant clone search would allow them to easily find and reference other similar code pieces. Conversely, commercial code search engines, such as Koders[1] or Google code search[2], may fail to suggest related code, because they treat software as text [16, 17]. Using a post-mortem clone detector is simply overkill for their purpose, because post-mortem detectors usually focus on finding all of the clone pairs, and it involves an unavoidable and expensive computational cost. In summary, instant clone search is a helpful approach to support rapid and evolving software development.

We summarize our key contributions as follows:

- ▷ We address the problem of how to support instant clone search.
- ▷ We develop clone indexing techniques to achieve sub-second response times for large-scale real-life software repositories without compromising clone search accuracy.
- ▷ For applications where the loss of some accuracy is acceptable, we also propose an approximation scheme, achieving a further speedup by trading off some accuracy.

[1]`http://koders.com/`
[2]`http://www.google.com/codesearch/`

The rest of the paper is organized as follows. Section 2 discusses the preliminaries, based on which Section 3 proposes our algorithms. Section 4 reports our evaluation results. Section 5 surveys related work, and Section 6 concludes this paper.

2. PRELIMINARIES

This section discusses the preliminaries in code clone detection (Section 2.1) and multidimensional indexing (Section 2.2). Building on these preliminaries, we formally define our problem in Section 2.3.

2.1 Code Clone Detection

There have been many code clone detection tools proposed recently, including some tree-based techniques, which abstract code segments as their corresponding parse trees or abstract syntax tree (AST) [2, 29]. Building on this abstraction, clone detection is essentially tree similarity matching, which is known to be inherently expensive [32], *e.g.*, using tree edit distance as a similarity notion.

To overcome this inherent complexity, DECKARD [11] approximated such similarity notion by representing an AST as multi-resolution numerical vectors, known as *characteristic vectors*. In other words, each tree node is represented as a vector representing the frequency of the syntactic elements in the code segment represented by its subtree. With this representation, an expensive tree match can be approximated as inexpensive vector matches. We adopt such abstraction as proposed in [11], since its simplicity makes it possible to develop sophisticated techniques to improve scalability.

2.2 Multidimensional Indexing

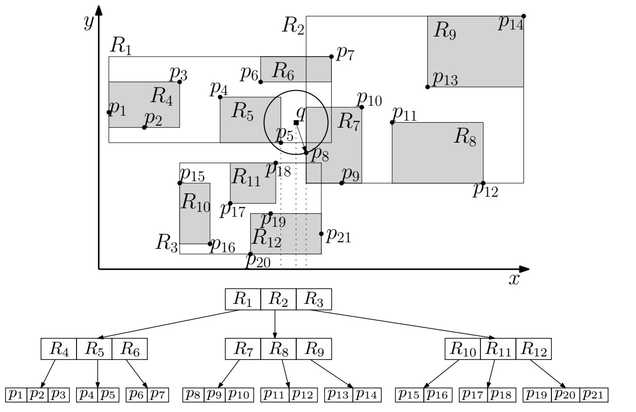
Characteristic vectors representing a code are often high-dimensional, *e.g.*, there are more than two hundreds different syntactic element types in `Java` ASTs. It is thus non-trivial to find matching vectors efficiently. To achieve sub-second response times, we index the code repository using a multidimensional indexing structure, *i.e.*, an `R*tree` [3], which have been widely adopted in database literature.

2.2.1 Naive Adoption

Intuitively, characteristic vectors can be mapped into multidimensional points, which can then be indexed using a multidimensional index structure such as an `R*tree`. A query, also represented as a vector, corresponds to another point, such that finding clones corresponds to finding the k-nearest neighbors (kNNs) of the query point, which has been actively studied in the database community [8, 6].

An `R*tree` is a height-balanced tree data structure, where each node contains a variable number of entries, up to some pre-defined maximum, *i.e.*, *node capacity*. For non-leaf nodes, each entry contains two pieces of data: a pointer to a child node, and the *minimum bounding rectangle* (MBR) of all entries within this child node. For leaf nodes, each entry has a pointer to a raw record stored on disks, and the MBR of this raw record.

To briefly illustrate how this structure can be used to support a kNN query, Figure 1 shows an example of an `R*tree` for 2-dimensional data points. To find kNNs of the query point q, this `R*tree` can be traversed in a best-first search manner, by initially storing the root node's entries in the queue and iteratively retrieving the closest entry in



y
x
q
R_1 R_2 R_3 R_4 R_5 R_6 R_7 R_8 R_9 R_{10} R_{11} R_{12}
p_1 p_2 p_3 p_4 p_5 p_6 p_7 p_8 p_9 p_{10} p_{11} p_{12} p_{13} p_{14} p_{15} p_{16} p_{17} p_{18} p_{19} p_{20} p_{21}

Figure 1: A kNN query on an R*tree for 2-dimensional points

the queue (with respect to Euclidean distance) and enqueuing its child node's entries. This type of search can terminate when the top k closest entries are all pointing raw data points, *e.g.*, p_5 and p_8 when $k = 2$.

2.2.2 Adoption with Dimensionality Reduction

Though adopting an index discussed above enables to find kNN with a tree traversal and avoid a sequential scan, such traversal can be more costly than a scan, when the dimensionality is high. This problem is known as the *dimensionality curse problem* [19]. To avoid this problem, dimensionality reduction techniques are typically used [20, 27], to select only a few important features and reduce the overall dimensionality.

However, in a dimension-reduced space, a kNN search result may differ from that in the original space. To illustrate this new challenge with example data in Figure 1, suppose we simply reduce the dimensionality of the dataset to one, by projecting points onto the x-axis. The closest point in this reduced space is then p_8 (*i.e.*, false positive), while the actual closest point in the original space is p_5.

Due to this challenge, in order to get the k nearest results in the original space from the reduced space, we first need to identify k' candidates that are guaranteed to contain all the correct k results ($k' \geq k$). For instance, in our projection example, retrieving $k = 1$ nearest point in the reduced space (*i.e.*, based on the projection on x-axis) will retrieve false positive p_8, but retrieving $k' \geq 2$ is guaranteed to include the correct answer p_5. To guarantee that k' candidates do not exclude any correct result, *i.e.*, no false negatives, the following *lower-bounding property* should hold.

DEFINITION 1 (LOWER-BOUNDING PROPERTY). *Given a dimensionality reduction function $F()$ and two data vectors v_1 and v_2, distance function $d_f()$ in the reduced space and distance function $d_o()$ in the original space should satisfy:*

$$d_f(F(v_1), F(v_2)) \leq d_o(v_1, v_2)$$

If the dimensionality reduction function $F()$ satisfies Definition 1, we can obtain the k' candidates as follows: First, we find the kNNs of $F(q)$ on the reduced space. We then sort these kNNs by their "real" distances to q on the original space, and choose the k^{th} nearest vector v_k from these kNNs. Finally, we identify all points v satisfying $d_f(F(q), F(v)) \leq d_o(q, v_k)$. This can be done through a simple ε-range search on the reduced space, with $\varepsilon = d_o(q, v_k)$, and these candidates are guaranteed to have all top-k results, as formally proved in [20]. We then rank these candidates by their "real" distances to find the correct k nearest results.

2.3 Problem Definition

This section formally defines the top-k code clone search problem. We define the *code segments* and the *characteristic vectors* in Definition 2 and 3. Definition 4 then defines the distances between vectors. Based on Definition 4, Definition 5 defines top-k code clones of a query code segment.

DEFINITION 2 (CODE SEGMENTS). *Given a code S, its AST T, and a threshold minT, if a subtree T_i of T contains at least minT nodes, then T_i's corresponding part in S is a code segment.*

DEFINITION 3 (CHARACTERISTIC VECTORS). *Given a code segment S_i and the AST T_i of S_i, the characteristic vector $v_i = \langle c_{i(1)}, c_{i(2)}, \cdots, c_{i(d)} \rangle$ of S_i consists of occurrence counters $c_{i(j)}$ of syntactic elements in T_i.*

DEFINITION 4 (DISTANCES BETWEEN VECTORS). *Given two d-dimensional vectors v_1 and v_2, the distance $\|v_1, v_2\|$ between v_1 and v_2 is their L_2-norm,*

$$\|v_1, v_2\| = \sqrt{\textstyle\sum_{i=1}^{d} (c_{1(i)} - c_{2(i)})^2}.$$

DEFINITION 5 (TOP-k CODE CLONES). *Given a set $\mathcal{V}$ of characteristic vectors, a query vector q, and the retrieval size k, top-k clones $\mathcal{TC}_k(q) \subset \mathcal{V}$ is a set of vectors $\mathcal{TC}_k(q) = \{v_1, v_2, \cdots, v_m\}$, where v_i is the i^{th} closest vector from q, $m \geq k$, and $\|q, v_i\| = \|q, v_k\|$ for $\forall i$ satisfying $k < i \leq m$.*

A top-k code clone search query q retrieves a set $\mathcal{TC}_k(q)$, and $\mathcal{TC}_k$ is used as its shorthand. For notational simplicity, we use $\mathcal{TC}_k$ to represent both code clones and their corresponding vectors interchangeably.

3. INSTANT CODE CLONE DETECTION

This section proposes indexing structures and algorithms to solve the top-k code clone search problem. As a baseline, Section 3.1 discusses a sequential scan algorithm, `Scan`. Section 3.2 then proposes a sub-linear algorithm, `FrTCD`, using an `R*tree` index with dimensionality reduction. Though `FrTCD` demonstrates reasonable scalability in medium-scale datasets, it still incurs prohibitive I/O costs. We thus study I/O optimization techniques to further improve the overall performance and build an enhanced algorithm, `InTCD`, upon the optimized `R*tree`s in Section 3.3. Lastly, Section 3.4 discusses how to further boost performance for scenarios where compromising some accuracy can be tolerated.

3.1 Baseline: Sequential Scan

One naive solution to find the clones of a given query code segment would be adopting an existing clone detector, identifying "clusters" of clones as their results. From these results, we can identify the cluster to which the given query code belongs and consider other codes in the same cluster as its clones. However, considering we only need one such cluster, finding all clusters is an overkill.

Alternative solution is to adopt *Locality Sensitive Hashing* (LSH) [7], as used by DECKARD [11] to efficiently find near neighbors (similar vectors) of each characteristic vector. However, LSH is not an exact nearest-neighbor algorithm, as we will empirically show later in Section 4.4.

We thus adopt a straightforward baseline approach for the exact computation, using a sequential scan, called `Scan`, which simply reads the entire repository sequentially and updates $\mathcal{TC}_k$, as Algorithm 1 illustrates.

```
Algorithm 1: Scan (q, k)
  Input  : query vector $q$, retrieval size $k$
  Output: set $\mathcal{TC}_k$ of vectors of top-$k$ clones
1 initialize $\mathcal{TC}_k \leftarrow \{\}$
2 for each $v \in \mathcal{V}$ do
3     UpdateClones ($\mathcal{TC}_k, k, q, v$);
4 return $\mathcal{TC}_k$
```

Specifically, `Scan` sequentially tests each characteristic vector $v \in \mathcal{V}$. For each v, `Scan` tests that v is not farther than any vector in the currently known top-k list $\mathcal{TC}_k$. If v is not farther, `Scan` updates the list $\mathcal{TC}_k$, as Algorithm 2 illustrates.

```
Algorithm 2: UpdateClones ($\mathcal{TC}_k, k, q, v$)
  Input  : set $\mathcal{TC}_k$, retrieval size $k$, query $q$, vector $v$
  /* $tc_i \in \mathcal{TC}_k$ denotes the $i^{th}$ nearest vector in
     $\mathcal{TC}_k$, from $q$                                   */
1 if $|\mathcal{TC}_k| < k$ then $\mathcal{TC}_k \leftarrow \mathcal{TC}_k \cup \{v\}$
2 else if $|\mathcal{TC}_k| \geq k$ and $\|q, v\| \leq \|q, tc_k\|$ then
3     $\mathcal{TC}_k \leftarrow \mathcal{TC}_k \cup \{v\}$
4     remove $\forall tc_i \in \mathcal{TC}_k$ farther than $tc_k$ from $q$
```

3.2 Filtering-then-Ranking Clone Detection

This section proposes a *filtering-then-ranking* top-k code clone search algorithm, called `FrTCD`, using an `R*tree` index. However, naively adopting this type of index structure incurs undesirable higher cost compared to a simple sequential scan, as discussed in Section 2.2.

To overcome this challenge, we first discuss a dimensionality reduction technique in Section 3.2.1. We then discuss how to build an index on this reduced space (Section 3.2.2) and then execute top-k clone queries (Section 3.2.3).

3.2.1 Dimensionality Reduction

For a given set $\mathcal{V}$ of $\mathcal{D}$-dimensional N characteristic vectors $\{v_1, v_2, \cdots, v_N\}$, our goal in dimensionality reduction is to generate lower-dimensional vectors $\mathcal{V}' = \{v'_1, v'_2, \cdots, v'_N\}$, which satisfy the *lower-bounding property* (Definition 1), and make our algorithms efficient.

As discussed in Section 2, it is important to preserve the lower-bounding property to ensure that we can retrieve candidates including all of the correct k results, by searching the reduced space only. Formally, for all v_i and $v_j \in \mathcal{V}$, and their corresponding reduced vectors v'_i and v'_j, the distances measured in the original space and the reduced space should satisfy $\|v'_i, v'_j\| \leq \|v_i, v_j\|$. We can trivially show that selecting any $\mathcal{D}'$-dimensional subspace of the original $\mathcal{D}$-dimensional space ensures the lower-bounding property.

However, not all such subspaces are equally effective. A desirable subspace should reflect the original distances between vectors, or more formally, minimize the sum Δ of differences $\delta_{i,j}$,

$$\Delta = \sum_{\forall i, \forall j, i \neq j} \delta_{i,j} = \sum_{\forall i, \forall j, i \neq j} \|v_i, v_j\| - \|v'_i, v'_j\|,$$

between two distances measured at the original space and the subspace respectively. Finding such subspace is known to be NP-hard [23], which motivates us to develop its approximation schemes.

A straightforward approximation would be a greedy strategy that iteratively picks the dimension that reduces Δ the most. However, this strategy requires recomputing Δ for all remaining dimensions, at each iteration. To avoid such recomputations, we propose to compute the variances of all dimensions once and select the $\mathcal{D}'$ dimensions with the highest variances.

To demonstrate that this variance-based approach is not only more efficient, but also as effective as the greedy strategy discussed above, we briefly compare 10-dimensional subspaces selected from 261-dimensional characteristic vectors obtained from real-life `java` source codes (7,195 files) in Table 1. The results in the table indicate that both approaches produce nearly identical results, while the variance-based method incurs a significantly lower cost.

Table 1: Top 10 selected dimensions

	Variance-based	Greedy strategy
1	identifier	identifier
2	ID_TK	ID_TK
3	unary_expression	unary_expression
4	multiplicative_expression	multiplicative_expression
5	additive_expression	additive_expression
6	**relational_expression**	**shift_expression**
7	**shift_expression**	**relational_expression**
8	equality_expression	equality_expression
9	conditional_expression	conditional_expression
10	assignment_expression	assignment_expression

3.2.2 Index Building

We now discuss how we can build an `R*tree` in the dimension-reduced space. A naive way to create an index is to insert one vector at a time, which incurs an expensive update on the index tree per each vector insertion. In contrast, a *bulk loading* approach amortizes the update cost, by inserting the entire dataset at once [5, 14, 21].

Existing bottom-up bulk loading algorithms first partition the entire dataset and build each partition as a leaf node. Then to build non-leaf nodes, they iteratively apply the same partitioning process to the resulting nodes, until we have only one partition including the entire dataset, which corresponds to the root node of the `R*tree`. Bulk loading tightly packs the index structure to enable fast lookups, and it is reported to boost the building performance by hundredfolds [4].

In particular, we revise a state-of-the-art bottom-up `R*tree` bulk loading algorithm, STR [21], to apply to our problem. STR partitions the given dataset into MBRs, by recursively subdividing each dimension into the same number of slices. Straightforwardly adopting this partitioning policy is not desirable for the dimension-reduced characteristic vectors, as the variance differs significantly over dimensions. That is, in one dimension, points are highly clustered in a small range, while in another, points are well scattered. For such data, partitioning each dimension into the same number of slices would render non-square rectangles (with one side significantly larger than the other), which incurs higher I/O cost than squared blocks, for the L_2 distance function used in our work.

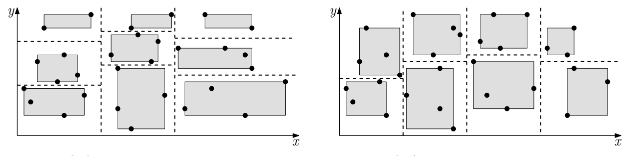

(a) STR packing (b) Our packing

Figure 2: STR partitions the data into 9 MBRs, while our packing algorithm partitions the data into 8 MBRs. Their utilizations are 83.3% and 93.75% respectively.

To illustrate, consider 30 points in a two-dimensional space, where the node capacity C of the tree is 4, *i.e.*, each tree node may hold at most 4 entries, and the x values are much more scattered than y values (Figure 2). At the bottom level of the tree, this dataset should be partitioned into $\lceil \frac{30}{4} \rceil = 8$ or more MBRs. To do this, STR partitions the space by dividing each dimension into an equal number of slices, *i.e.*, 3 slices containing the same number of data points, to render 9 MBRs (Figure 2(a)). In contrast, we divide the dataset into 8 MBRs (2-by-4 as illustrated in Figure 2(b)), to significantly enhance the node utilization of STR, *i.e.*, 83.3% $(\frac{30}{9\times4})$ into 93.75% $(\frac{30}{8\times4})$.

Formally, for a D-dimensional dataset containing N points, we subdivide the i^{th} dimension into $s_i = \lceil r_i/R \rceil$ slices, where r_i is the value range, computed as the difference between the maximum and minimum values of the i^{th} dimension. In other words, a dimension with a high r_i is highly scattered. Assuming points are uniformly scattered, R can be computed as $\prod_{i=1}^{D} \frac{r_i}{R} = \frac{N}{C}$. Algorithm 3 formally describes the data partitioning process.

Algorithm 3: `DataPartitioning` $(E, C, \boldsymbol{r})$

```
Input : entries $E = \{e_1, \cdots, e_N\}$, node capacity $C$,
        value ranges $\boldsymbol{r} = \langle r_1, \cdots, r_D \rangle$
Output: partitioned entries $E$
1 $R \leftarrow \sqrt[D]{\frac{C}{N} \cdot \prod_{i=1}^{D} r_i}$; $s_i \leftarrow \lceil r_i/R \rceil$, for $\forall i \in \{1, \cdots, D\}$
2 repeat
3     for each $s_i$ do
4         $s_i \leftarrow s_i - 1$ if it does not incur an overflow
5 until any decrease incurs an overflow
6 Slice $(E, 1, \langle s_1, \cdots, s_D \rangle)$
```

For a given set of entries E and its value ranges $\langle r_1, \cdots, r_D \rangle$, Algorithm 3 first computes R and $\langle s_1, \cdots, s_D \rangle$ (Line 1). Then Algorithm 3 tries to decrease s_i values (Lines 2-5), to further enhance the node utilization. In our observation, as s_i takes ceiling and thus overestimates the right number of subdivisions in each dimension, the total number of partitions, $\prod_{i=1}^{D} s_i$, may become too large, *i.e.*, incurring the low node utilization. To address this problem, Algorithm 3 decreases each s_i by one, until any decrease incurs an overflow, *i.e.*, with node utilization higher than 100%.

```
Algorithm 4: Slice $(E, i, \boldsymbol{s})$
   Input   : entries $E$, dimension index $i$, $\boldsymbol{s} = \langle s_1, \cdots, s_D \rangle$
1  sort $E$ according to the $i^{th}$ dimension
2  subdivide $E$ into $\{E'_1, \cdots, E'_{s_i}\}$
3  if $i < D$ then
4      for each $E'_i$ do Slice $(E'_i, i+1, \langle s_1, \cdots, s_D \rangle)$
```

Once the "tightest" s_i values are determined, we recursively subdivide each dimension as described in Algorithm 4. Algorithm 4 first sorts the set of entries E in the ascending order of the i^{th} dimension, then divides E into s_i subdivisions, namely E_1, $\cdots$, and E_{s_i}. We make sure the first $s_i - 1$ subdivisions to contain $\lceil |E|/s_i \rceil$ where $|E|$ denotes the number of entries in E. E_{s_i} contains the remaining entries.

To build an R$*$tree, we first partition the reduced vectors using Algorithm 3. The resulting partitions become leaf nodes. Then we perform Algorithm 3 again on the MBRs of the leaf nodes. In this case, we sort them using their centers as representative points of the MBRs. We repeat this partitioning process until we have only one partition, which corresponds to the root of the tree.

3.2.3 Two-phase Query Processing

This section proposes the filtering-then-ranking top-k code clone search algorithm, FrTCD, to evaluate top-k code clone queries. Basically, FrTCD works in two phases of filtering and ranking as Algorithm 5 formally states.

```
Algorithm 5: FrTCD $(q, k, T)$
   Input   : query vector $q$, retrieval size $k$, R*tree T
   Output: set $\mathcal{TC}_k$ of vectors of top-$k$ clones
1  $q' \leftarrow$ the reduced vector of $q$
2  $\mathcal{N} \leftarrow$ the $k$ nearest neighbors of $q'$;          /* from $T$ */
3  $v'_k \leftarrow$ the $k^{th}$ nearest vector in $\mathcal{N}$
   /* by the distances on the original space      */
4  $\varepsilon \leftarrow \|q, v_k\|$; $\mathcal{C} \leftarrow \{v' : \|q', v'\| \leq \varepsilon\}$;      /* from $T$ */
5  initialize $\mathcal{TC}_k \leftarrow \{\}$
6  for each $v$ corresponding to $v' \in \mathcal{C}$ do
7      UpdateClones $(\mathcal{TC}_k, k, q, v)$;
8  return $\mathcal{TC}_k$
```

In the *filtering phase* (Lines 1-4), for a given query vector q, FrTCD identifies candidate vectors $\mathcal{C}$, using a combination of kNN and range search on the reduced space. More precisely, FrTCD finds the kNNs, $\mathcal{N}$, of q', by traversing the R$*$tree in a best-first search manner (Line 2), and computes their distances to q in the original space to choose the k^{th} nearest vector v'_k among $\mathcal{N}$ (Line 3). $\|q, v_k\|$ denotes the actual distance between the given query q and the characteristic vector $v_k \in \mathcal{V}$ corresponding to v'_k.

FrTCD then performs an ε-range search, where $\varepsilon = \|q, v_k\|$, to find all the reduced vectors $\mathcal{C}$ within the distance $\|q, v_k\|$ from q' (Line 4). As $\|q, v_k\| \geq \|q', v'_k\|$, $\mathcal{C}$ is guaranteed to contain correct k results. We can thus prune out the remaining vectors $\mathcal{V}' \setminus \mathcal{C}$, as they are farther away from q, compared to v_k.

In the *ranking phase* (Lines 5-7), FrTCD computes the distance of each characteristic vector v corresponding to $v' \in \mathcal{C}$, from q. Observe that, unlike Scan that computes the distance for all objects, FrTCD computes the distance of a very small subset $\mathcal{C}$ of raw data records ($\mathcal{N} \subset \mathcal{C}$), *i.e.*, a sub-linear algorithm. After FrTCD computes the distances of these candidates to q, we can finalize the list of top-k code clones.

3.3 Interleaved Clone Detection

Though we empirically observe that FrTCD already achieves an acceptable efficiency for medium-scale datasets, as we will later show with our extensive evaluation results, we can also observe that FrTCD has room for further improvements, as much I/O costs are wasted on random accesses on data records– According to our experimental results, 80.5% of the overall response time of FrTCD (3.4 seconds to find the top-20 code clones in the repository of 1.7 million vectors) corresponds to the random access cost.

Based on this observation, we study two I/O optimization techniques, (1) reducing the number of random accesses and (2) reducing the cost of each access.

3.3.1 Vector Packing

We first study how we reduce the number of random accesses by "packing" a group of records to be accessed together, *i.e.*, those with similar values. This type of packing allows us to reach multiple records with a single random access followed by cheaper sorted accesses, which incurs a significantly lower cost than performing a random access per each record. For one-dimensional data, this packing can be implemented straightforwardly, by storing raw data records in the same order as the index key. However, for multi-dimensional data, it is non-trivial to identify an effective one-dimensional sorted order to store records.

Figure 3 illustrates a scenario, reading 20 data records within ε from q requires 20 accesses. However, if these points are packed into two blocks, B_1 and B_2, we only need two random accesses to get to the beginnings of the blocks, and the remaining records can be retrieved by cheaper sequential accesses. While this strategy may incur the overhead of retrieving few more false positive records, the overall I/O cost is greatly reduced, as we also empirically validate in Section 4.

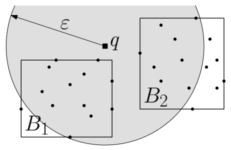


Figure 3: A range query example

For this packing, we apply the same scheme proposed for bulk loading in Section 3.2.2, and store these blocks in several files. After the packing is done, we simply build an R$*$tree on the MBRs of the blocks. Each MBR is computed in the reduced $\mathcal{D}'$-dimensional subspace, and the data blocks contain the original $\mathcal{D}$-dimensional vectors. As we use the same scheme in Section 3.2.2 for this packing, data insertion and deletions can be handled in a similar way that the R$*$trees do without the packing.

3.3.2 Single-phase Query Processing

This section proposes InTCD, adopting the vector packing scheme discussed in the previous section. Specifically, we implement InTCD to (1) further reduce the number of random accesses to index nodes by interleaving the kNN search and the range search in the filtering phase of `FrTCD`, and (2) reduce the cost of random accesses.

***Interleaved Index Traversal*:** Recall that `FrTCD` performs a best-first search to find the kNNs in the reduced space, then performs a range search to find candidates. After this candidate selection, `FrTCD` reads the raw records of the candidates to compute the real distances from the query. In contrast, `InTCD` interleaves these two steps, by concurrently accessing raw records "during" the index traversal.

Basically, `InTCD` traverses the index in the reduced space in the same manner as `FrTCD`. However, during the traversal, when a leaf entry is reached, `InTCD` accesses the raw data block pointed by the leaf entry, without waiting for the index traversal to complete, as the cost of reading few extra raw data is much affordable now with vector packing. Whenever data records are accessed from the leaf entry, `InTCD` updates a sorted list, $\mathcal{TC}_k$, of the current known top-k clones, and we denote the current k^{th}-NN in the list as tc_k.

Our key observation is that tc_k can be used as a pruning boundary, as we can safely prune out both non-leaf and leaf entries that are farther than tc_k. As more data records are accessed, $\mathcal{TC}_k$ converges to the actual top-k results. In this interleaved traversal, each index node is accessed at most once, which enables to outperform `FrTCD` accessing some index nodes twice during the kNN and the range search respectively.

Algorithm 6 formally describes this process of `InTCD`.

Algorithm 6: `InTCD` (q, k, T)

```
   Input  : query vector $q$, retrieval size $k$, R*tree $T$
   Output: set $\mathcal{TC}_k$ of vectors of top-$k$ clones
   /* $tc_i \in \mathcal{TC}_k$ denotes the $i^{th}$ nearest vector in
      $\mathcal{TC}_k$, from $q$                                  */
1  $q' \leftarrow$ the reduced vector of $q$
2  $\mathcal{TC}_k \leftarrow \{\}$; $\mathcal{Q} \leftarrow \{\}$; $\mathcal{H} \leftarrow$ {entries within the root of $T$}
3  while $\mathcal{H}$ is not empty do
4      $e \leftarrow \mathcal{H}.pop()$
5      if $|\mathcal{TC}_k| < k$ or mindist $(q', e) \leq \|q, tc_k\|$ then
6          if $e$ is not a leaf then $\mathcal{H}.push$(children of $e$)
7          else
8              $\mathcal{Q}.push(e)$
9              if $|Q| > \mathcal{W}$ then
10                 $E \leftarrow$ pop block pointers from $\mathcal{Q}$
11                 for each $v \in$ a block of $E$ do
12                     UpdateClones $(\mathcal{TC}_k, k, q, v)$;
13 while $\mathcal{Q}$ is not empty do
14     $E \leftarrow$ pop block pointers from $\mathcal{Q}$
15     for each $v \in$ a block of $E$ do
16         UpdateClones $(\mathcal{TC}_k, k, q, v)$;
17 return $\mathcal{TC}_k$
```

Specifically, to implement this single-scan best-first search, a min heap $\mathcal{H}$ of e is maintained in the ascending order of `mindist` (q', e), where q' denotes the reduced vector of query q, e is an entry of the `R*tree` index, and `mindist` (q', e) denotes the shortest distance between q' and e (Figure 4).

At the beginning, the entries within the root of T are pushed into $\mathcal{H}$ (Line 2). Then iteratively, the entry e in $\mathcal{H}$ with the minimal `mindist` (q', e) is processed. If the `mindist` (q', e) is no farther than the distances of the current tc_k to q, we continue the iterations. Otherwise, we can safely ignore e (Line 5).

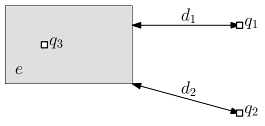


Figure 4: `mindist` (q_i, e) in a 2-dimensional space. `mindist` $(q_1, e) = d_1$, `mindist` $(q_2, e) = d_2$, and `mindist` $(q_3, e) = 0$.

If `mindist` $(q', e) \leq \|q, tc_k\|$, we test if e is a leaf entry or not. If e is not a leaf, then the entries within its child node are pushed into $\mathcal{H}$ (Line 6). Otherwise, we process the raw data block pointed by e.

When processing raw data, a naive approach would be reading each raw data block right away, which incurs high random seek costs. Instead, to reduce the cost of random accesses, we devise an effective scheduling technique, called *delayed loading*, by adopting the idea of Circular SCAN disk scheduling [28].

***Delayed Loading*:** Specifically, we propose a delayed loading scheme, which delays the reading the block of e until we collect multiple blocks to read as the name itself suggests. Figure 5 illustrates how this scheme works in a scenario involving the reading of four blocks B_1, B_2, B_3, and B_4. They are stored on the disk in a sorted order, and the `mindist`s of their corresponding entries e_1, e_2, e_3, and e_4 satisfy `mindist` $(q', e_2) <$ `mindist` $(q', e_4) <$ `mindist` $(q', e_3) <$ `mindist` (q', e_1).

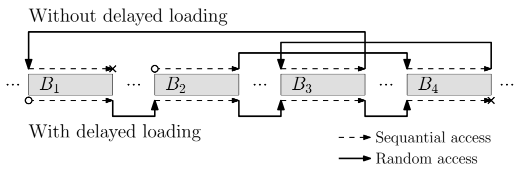


Figure 5: Effectiveness of delayed loading

A naive solution would be reading each block one at a time, which incurs four expensive seeks, but instead, we can read these four blocks at once, in the forward direction as in Figure 5, which incurs cheaper seeks followed by sequential scan.

In Algorithm 6, we maintain a delayed loading queue $\mathcal{Q}$, which contains block pointers e in the ascending order of `mindist` (q', e). Whenever we have a raw data block to read, *i.e.*, when a leaf entry e is reached, we push it into $\mathcal{Q}$ (Line 8). We then delay reading these blocks until we collect a sufficient number of entries in $\mathcal{Q}$, determined by the given threshold $\mathcal{W}$, which we set as 50 in our experiment (Line 9). When this happens, `InTCD` then reads these blocks in batch (Lines 11-12). Once the traversal terminates, we process the remaining blocks in $\mathcal{Q}$ (Lines 13-16).

3.4 Approximate Clone Detection

For applications where some accuracy compromise can be tolerated, approximation is a good strategy to trade accuracy for an even higher performance. In this section, we discuss how we study an approximation scheme, empirically achieving a 19 times speedup against `InTCD`, while compromising no more than 30% of the accuracy.

Specifically, we propose an approximate top-k code clone search algorithm `ApTCD`, which efficiently identifies approximate answers without reading or processing high-dimensional data records $\mathcal{V}$. Instead, we read further dimensionality reduced records $\mathcal{V}'$ and issue a dimensionality-reduced query q'.

Intuitively, as dimensionality of $\mathcal{V}'$ increases, efficiency decreases but accuracy increases. To balance this trade-off, we need to effectively select the subspace $\mathcal{V}'$. This goal is similar to that of the dimensionality reduction we discussed for exact algorithms (Section 3.2.1), our approach should be different, as the reduction this time is applied upon already dimensionality-reduced space. As a result, applying the same technology would be redundant and ineffective.

From the reduction results reported in Table 1, obtained by variance-based ranking, we observed that features with similar variances tend to be correlated from one another. For example, features *identifier* and *ID_TK* with the same variance were also perfectly correlated with each other, which suggests that these two features can be reduced into one feature without any loss of accuracy.

There have been many reduction techniques studied for aggregating correlated features, such as *Principal component analysis* (PCA) [12] and *Piecewise aggregate approximation* (PAA) [31]. In this research, which aims at scalability, we consider PAA with higher scalability.

Figure 6 illustrates how PAA reduces the dimensionality. In the figure, elements in v are sorted in a descending order based on the variances of their corresponding dimensions.

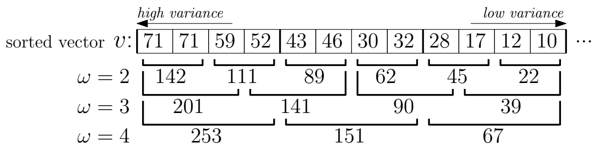


Figure 6: PAA example

To illustrate, consider a scenario of reducing $v \in \mathcal{V}$ into a 3-dimensional vector v'. Our variance-based dimension reduction method for `FrTCD` and `InTCD` simply chooses $v' = \langle 71, 71, 59\rangle$. PAA uses a fixed size disjoint window ω to divide v, then aggregates each window. Subsequently, if $\omega = 2$, $v' = \langle 142, 111, 89\rangle$. Similarly, if $\omega = 3$, $v' = \langle 201, 141, 90\rangle$.

Once we reduce the dimensionality, to evaluate approximate top-k queries, we simply build an `R*tree` over these reduced vectors $\mathcal{V}'$, as shown in Section 3.2.2. We can then find the approximate results by traversing the tree in a best-first manner (Algorithm 7), similarly to our exact algorithms.

As we later discuss in Section 4, our experimental results indicate that applying PAA is effective for achieving even higher performances, *e.g.*, 88 times speedup while compromising 58% of accuracy against `InTCD` when $\mathcal{D} = 12$ and $\omega = 4$.

4. EXPERIMENTAL EVALUATION

This section empirically evaluates our proposed algorithms. First, we describe how we generate datasets and queries in Section 4.1. Second, we evaluate the efficiency and scalability of our algorithms in Section 4.2. Third, we validate the effectiveness of our approximate query processing scheme

Algorithm 7: `ApTCD` (q, k, T)

```
  Input  : query vector $q$, retrieval size $k$, R*tree T
  Output: set $\widetilde{\mathcal{TC}}_k$ of approximate top-$k$ clones
1 $q' \leftarrow$ the reduced vector of $q$
2 $\widetilde{\mathcal{TC}}_k \leftarrow \{\}$; $\mathcal{H} \leftarrow \{$root of $T\}$
3 while $\mathcal{H}$ is not empty do
4     $e \leftarrow \mathcal{H}.pop()$
5     if mindist $(q', e) \leq \|q', \widetilde{tc}_k\|$ then
6         if $e$ is not a leaf then $\mathcal{H}.push$(children of $e$)
7         else UpdateClones($\widetilde{\mathcal{TC}}_k, k, q', e$);
8 return $\widetilde{\mathcal{TC}}_k$
```

in Section 4.3. Lastly, we compare our proposed indexing structure with *Locality Sensitive Hashing* (LSH) that is used by a state-of-the-art post-mortem clone detector, DECKARD (Section 4.4). All experiments were carried out on a machine with a Pentium IV 3.2GHz processor, with 1GB of memory, running Linux.

4.1 Experimental Setup

We generate characteristic vectors, in the same way described in [11] for DECKARD, from two real-life `java` code repositories. The first repository contains 7,195 `java` files from JDK 1.6.0 Update 13, consisting of 2,075,573 lines of code total, and the second repository contains 288,846 `java` files (54,709,384 lines) from 492 Java open source projects hosted on SourceForge, Tigris.org and GoogleCode.

From these repositories, we generated six vector datasets, two from the JDK code set (denoted as $\mathsf{JDK}_{3,5}$), and four from the open source project code set (denoted as $\mathsf{OSP}_{3,5,7,9}$), by varying the parameter `minT` of DECKARD. The dimensionality of each vector is 261. Table 2 summarizes the sizes of the resulting vector datasets and the `minT` setting. Once these vector sets were generated, we randomly chose one hundred vectors from each dataset to use as our queries.

4.2 Efficiency & Scalability

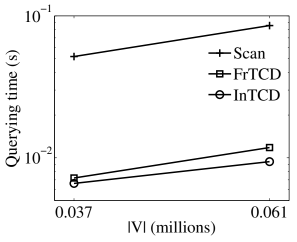
To evaluate the efficiency and scalability, this section reports index building time and query execution time for varying retrieval sizes k and dataset sizes $|\mathcal{V}|$. For this set of experiments, we empirically chose 20 as the index dimensionality. We increase k up to 80 to test scalability, though we may use very small k in practice.

Table 2: Index building time for varying $|\mathcal{V}|$

Dataset	minT	$\|\mathcal{V}\|$	Building time (s)	
			FrTCD	InTCD
JDK_5	50	36,658	0.563	0.867
JDK_3	30	60,582	0.793	1.517
OSP_9	90	612,926	8.968	34.055
OSP_7	70	783,933	11.619	46.725
OSP_5	50	1,072,598	16.939	72.903
OSP_3	30	1,696,806	27.653	128.118

Table 2 summarizes index building time of `InTCD` and `FrTCD`. This table only shows the building time, excluding the data processing time for vector extraction and dimensionality reduction. Observe from the table that `InTCD` takes a relatively longer time than FrTCD to accomplish the vector packing process, as vectors packed into blocks need to be stored, causing extra I/Os. However, owing to this packing scheme, InTCD performs better than FrTCD in the later experiments. Both the indexes for FrTCD and InTCD can be built in minutes, which is acceptable considering that index creation is a (1) one-time and (2) offline process.

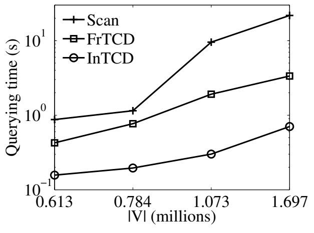
The block sizes for these two datasets were tuned empirically. For JDK datasets, the block size was set as 8KB and each file contained at most 32 blocks, as the performance was optimal with such setting. Similarly, for OSP datasets, we set the block size and file size as 384KB and 120 blocks respectively.



$\lvert\mathcal{V}\rvert$ (millions)	$\lvert\mathcal{TC}_{20}\rvert$
0.037	20.85
0.061	24.05

Figure 7: Querying time for varying $|\mathcal{V}|$, JDK datasets, $k = 20$ (log-scaled). The table lists the average number of clones in $\mathcal{TC}_{20}$.

Figure 7 shows the average querying time for varying $|\mathcal{V}|$ using JDK datasets, and the table shows the average number of vectors in $\mathcal{TC}_k$. (This number can be larger than k due to the ties in the results.) Both FrTCD and InTCD are at least 7 times faster than Scan. Though in this medium-scale dataset, the performance gain of InTCD over FrTCD is less significant, this type of performance gap significantly increases as the scale of the data increases, as we later show with the larger datasets in Figure 8.

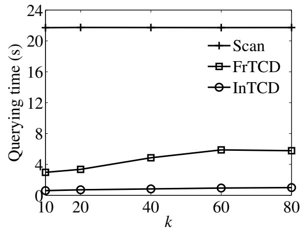


$\lvert\mathcal{V}\rvert$ (millions)	$\lvert\mathcal{TC}_{20}\rvert$
0.613	27.26
0.784	40.55
1.073	56.12
1.697	73.23

Figure 8: Querying time for varying $|\mathcal{V}|$, OSP datasets, $k = 20$ (log-scaled).

Figure 8 shows the average querying time for varying $|\mathcal{V}|$, using OSP datasets. Compared to Figure 7, InTCD achieves higher speed-up over FrTCD, *i.e.*, 31.7 times faster than Scan, which suggests that our proposed I/O optimization techniques are more effective in larger-scale datasets and play a crucial role in enhancing the overall efficiency.

Figure 9 shows the querying time over varying k, using OSP_3 dataset. As Scan reads the entire data once regardless of k, its performance is constant over varying k. In clear contrast, InTCD and FrTCD are 36 and 7.3 times faster than Scan when $k = 10$. Considering that k is typically much smaller than the data size in general search scenarios, this



k	$\lvert\mathcal{TC}_k\rvert$
10	63.60
20	73.23
40	94.92
60	113.78
80	135.93

Figure 9: Querying time over varying k, OSP_3 dataset ($|\mathcal{V}|$=1,697K).

"progressive" behavior of our proposed algorithms, incurring smaller cost for smaller k, is highly desirable.

4.3 Effectiveness of the Approximation

To validate the proposed approximate query processing algorithm, called ApTCD, this section reports its performance and approximation quality for varying approximation settings, compared to InTCD.

Figure 10 shows the performance and quality of our approximation algorithm for varying approximation settings. We varied the aggregation window size ω from 1 to 5, the dimensionality $\mathcal{D}$ of the index from 12 to 32, and k was set to 20.

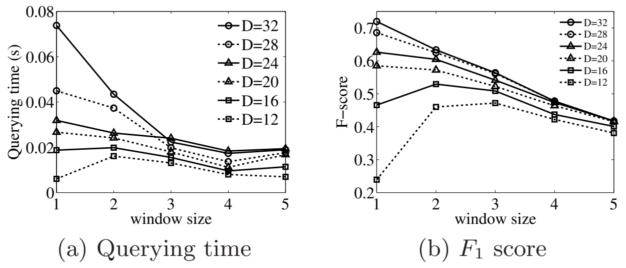


Figure 10: Performance and quality of ApTCD for varying settings, OSP_3 dataset, $k = 20$.

In Figure 10(a), it is clear that the lower dimensional indexes perform better than higher ones in general. Figure 10(b) reports the approximation accuracy, measured using the balanced F-scores (F_1 scores) [25]:

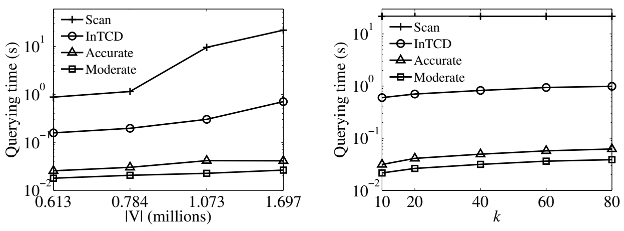
$$F_1 \text{ score} = \frac{2 \cdot \text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}},$$

$$\text{Precision} = \frac{|\widetilde{\mathcal{TC}}_k \cap \mathcal{TC}_k|}{|\widetilde{\mathcal{TC}}_k|}, \quad \text{Recall} = \frac{|\widetilde{\mathcal{TC}}_k \cap \mathcal{TC}_k|}{|\mathcal{TC}_k|},$$

where $\mathcal{TC}_k$ and $\widetilde{\mathcal{TC}}_k$ denote the query result sets using InTCD and ApTCD, respectively. Note that $|\mathcal{TC}_k|$ and $|\widetilde{\mathcal{TC}}_k|$ are not always equal to k, because they may have ties.

Observe from the figure that our proposed reduction using PAA enables a high speed-up without compromising the accuracy much when $\mathcal{D} = 12$ and $\omega = 2$ or 3.

We now compare ApTCD with InTCD using the following two settings. First, we chose the setting where ApTCD is most *accurate*, *i.e.*, $\mathcal{D} = 32$ and $\omega = 1$. Second, we chose a *moderate* setting, where $\mathcal{D} = 24$ and $\omega = 2$. By using these two settings, *accurate* and *moderate*, we compared our approximation scheme with the exact querying algorithm proposed, InTCD.



(a) Effect of $|\mathcal{V}|$, $k = 20$ (b) Effect of k, OSP_3

Figure 11: Approximation performance using OSP

Figure 11 and Table 3 respectively summarizes the performances and accuracies of our approximation scheme using the above two settings. In Figure 11, our approximation scheme in both settings significantly outperformed `InTCD`, and the gap increased as the size of the dataset increases. For OSP_3 dataset, our approximation is 28 and 19 times faster than `InTCD` for *moderate* and *accurate* respectively. Meanwhile, the accuracy was not compromised much, as the precision and recall results in the *accurate* setting show, *e.g.*, constantly higher than 0.7.

Table 3: Approximation quality using OSP datasets

$\|\mathcal{V}\|$ (millions)	k	*Accurate* Precision	*Accurate* Recall	*Moderate* Precision	*Moderate* Recall
0.613	20	0.728	0.723	0.586	0.581
0.784		0.746	0.761	0.577	0.575
1.073		0.761	0.764	0.599	0.591
1.697		0.713	0.741	0.609	0.609
1.697	10	0.725	0.713	0.633	0.637
	20	0.713	0.741	0.609	0.609
	40	0.725	0.730	0.586	0.588
	60	0.710	0.730	0.575	0.593
	80	0.720	0.733	0.578	0.580

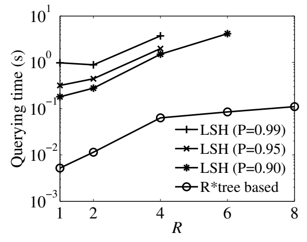
4.4 Comparison with LSH

Lastly, we evaluate the proposed indexing structure used in `InTCD`, compared to the LSH implementation used by DECKARD (Figure 12).

***Experiment Setting*:** For the given R-range query, our `R*tree` based index returns the exact answers, while LSH returns approximate results with probability guarantee P. More precisely, LSH requires two parameters P and R for building the index. For the given query point q, LSH returns all points p such that $\|p, q\| \leq R$ with probability of P or higher. Due to this nature, technically, LSH structure needs to be re-built as R changes, to ensure the probabilistic guarantee, unlike our `R*tree` based approach building one single structure and reusing for arbitrary range R.

To accommodate such difference, we use a favorable setting for LSH, of not considering the rebuilding cost of LSH, to show our approach outperforms even in such unfair setting in Figure 12(a). For queries, we randomly selected 100 vectors from OSP_7, and varied R from 1 to 8.

***Querying Time*:** Observe from Figure 12(a) that, even in unfavorable settings, our index outperforms LSH in terms of query execution time. This experiment also shows that, as R increases, the performance gap also increases, which suggests our approach is more scalable for large R. For example, when $R = 4$ (which means selecting all data points



R	Recall P=0.95	Recall P=0.90
1	1.000	1.000
2	1.000	1.000
4	0.993	0.993
6	-	0.986

(a) Performance (b) LSH accuracy

Figure 12: Range query performance (log-scaled), OSP_7 dataset ($|\mathcal{V}|$=784K).

with in L_2 distance 4 from the query), our index is about 24 times faster than LSH (with $P = 0.9$), and when $R = 6$, ours is 49 times faster than LSH.

***Memory Use*:** Meanwhile, in Figure 12(a), some data points for LSH are missing, which are the cases when LSH could not return results due to memory shortage.

Alternatively, we can use a disk-based implementation of LSH, not to be constrained by memory size. However, its performance will be worse than that of memory-based implementation reported in the figure, which is already outperformed by our approach. DECKARD goes around this problem, by dividing a problem into sub-problems and apply LSH for each sub-problem.

***Accuracy*:** In terms of accuracy, we compare the precision and recall of our approach and LSH. In Figure 12(b), as the precision of LSH was perfect in all settings, we only report its recall when $P = 0.95$ and 0.90 respectively. Observe that, as R increases, *e.g.*, $R = 4$ or 6 in the figure, LSH is more likely to miss some vectors within the range R. In clear contrast, our indexing scheme guarantees the perfect precision and recall in all cases.

Summing up, for instant clone search, our `R*tree` based indexing is more suitable than LSH, by ensuring (1) higher performance, (2) effective memory usage, and (3) perfect accuracy.

5. RELATED WORK

This section surveys existing research on (1) code clone detection, (2) code example recommendation, and (3) code search.

***Clone Detection*:** As already briefly surveyed in Section 2, there have been many clone detection techniques proposed that abstract codes as parsing trees and apply hashing [2] or characteristic vector comparison [11] for clone detection. In a similar way, Wahler *et al.* [29] abstracted codes as XML trees and adopted the concept of *frequent itemsets* to find clones that share frequent tree patterns. Recently, a distributed code clone analysis algorithm, called D-CCFinder [24] was proposed, which improves the scalability of CCFinder [15] by leveraging multi-cluster machines. However, these tools are still not scalable enough to achieve online detection in large-scale repositories.

***Code Recommendation*:** Meanwhile, there have been alternative lines of research taking place, to find code examples that share (1) similar usage patterns or (2) structural similarity.

In the first line of work, Xie *et al.* [30] proposed an approach to abstract codes as API call sequences and identify examples that share similar sequences. Li *et al.* [22] used the frequency of operation calls to define similarity, and then clustered similar examples into a few representative usage types. In the second line of work, Holmes *et al.* [9, 10] proposed some heuristic structural matching techniques to find relevant example codes. However, the former line of work [30, 22] cannot be used for structural similarity search and the latter [9, 10] has limited scalability.

***Code Search Engine*:** There are commercial code search engines, including *Koders* and *Google Code Search* that abstract codes as text and support simple and regular expression keyword matches. However, these engines, which treat codes as simple text, do not support structural matches. Sourcerer [1] stores some structural information on codes in relational tables and provides ranked matching, but does not focus on optimizing search performance.

6. CONCLUSION

In this paper, we introduced scalable and instant code clone search techniques. These techniques open doors to many interesting unexplored applications, such as interleaving clone detection with editing sessions during code development.

We evaluated the accuracy and efficiency of our approach with large-scale real-life software repositories. In addition to exact code clone search, we also developed an approximation algorithm for scenarios where some accuracy compromise can be tolerated, which performs nearly a thousand times faster than the baseline approach. Both the exact and approximation algorithms achieved sub-second response times for large-scale real-life repositories of 1.7 million code segments.

As future work, we are considering the following:

- ▷ More features: In addition to the characteristic vectors, we will consider more features such as the structural relationship between vectors or runtime semantics, to enable more precise matching.
- ▷ Industry-scale detection: To build commercial engines, *e.g.*, to achieve Google's scalability over billions of documents, we need to deal with unexplored issues such as parallelization.

7. ACKNOWLEDGMENTS

Our thanks to Lingxiao Jiang for providing DECKARD source code. This research was supported by National IT Industry Promotion Agency (NIPA) under the program of Software Engineering Technologies Development and the Engineering Research Center of Excellence Program of Korea Ministry of Education, Science and Technology (MEST) / National Research Foundation of Korea (NRF) (Grant 2010-0001728).

8. REFERENCES

[1] S. K. Bajracharya, T. C. Ngo, E. Linstead, Y. Dou, P. Rigor, P. Baldi, and C. V. Lopes. Sourcerer: a search engine for open source code supporting structure-based search. In *OOPSLA Companion*, 2006.

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

[3] N. Beckmann, H. peter Begel, R. Schneider, and B. Seeger. The r*-tree: an efficient and robust access method for points and rectangles. In *SIGMOD*, 1990.

[4] S. Berchtold, C. Böhm, and H.-P. Kriegel. Improving the query performance of high-dimensional index structures by bulk-load operations. In *EDBT*, 1998.

[5] J. V. d. Bercken and B. Seeger. An evaluation of generic bulk loading techniques. In *VLDB*, 2001.

[6] C. Böhm and F. Krebs. The k-nearest neighbour join: Turbo charging the kdd process. *Knowl. Inf. Syst.*, 6(6):728–749, 2004.

[7] A. Gionis, P. Indyk, and R. Motwani. Similarity search in high dimensions via hashing. In *VLDB*, 1999.

[8] G. R. Hjaltason and H. Samet. Distance browsing in spatial databases. *ACM TODS*, 24:265–318, 1999.

[9] R. Holmes and G. C. Murphy. Using structural context to recommend source code examples. In *ICSE*, 2005.

[10] R. Holmes, R. J. Walker, and G. C. Murphy. Approximate structural context matching: An approach to recommend relevant examples. *IEEE Trans. Softw. Eng.*, 32(12):952–970, 2006.

[11] L. Jiang, G. Misherghi, Z. Su, and S. Glondu. DECKARD: Scalable and accurate tree-based detection of code clones. In *ICSE*, 2007.

[12] I. T. Jolliffe. *Principal Component Analysis*. Springer Series in Statistics. Springer, 2nd edition, 2002.

[13] E. Jürgens, B. Hummel, F. Deissenboeck, and M. Feilkas. Static bug detection through analysis of inconsistent clones. In *Software Engineering (Workshops)*, pages 443–446, 2008.

[14] I. Kamel and C. Faloutsos. On packing r-trees. In *CIKM*, 1993.

[15] 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.*, 28(7):654–670, 2002.

[16] J. Kim, S. Lee, S. won Hwang, and S. Kim. Adding examples into java documents. In *ASE*, 2009.

[17] J. Kim, S. Lee, S. won Hwang, and S. Kim. Towards an intelligent code search engine. In *AAAI*, 2010.

[18] M. Kim, V. Sazawal, D. Notkin, and G. Murphy. An empirical study of code clone genealogies. *SIGSOFT Softw. Eng. Notes*, 30(5):187–196, 2005.

[19] F. Korn, B.-U. Pagel, and C. Faloutsos. On the 'dimensionality curse' and the 'self-similarity blessing'. *TKDE*, 13(1):96–111, 2001.

[20] F. Korn, N. Sidiropoulos, C. Faloutsos, E. Siegel, and Z. Protopapas. Fast nearest neighbor search in medical image databases. In *VLDB*, 1996.

[21] S. T. Leutenegger, J. M. Edgington, and M. A. Lopez. STR: A simple and efficient algorithm for r-tree packing. In *ICDE*, 1997.

[22] Y. Li, L. Zhang, G. Li, B. Xie, and J. Sun. Recommending typical usage examples for component retrieval in reuse repositories. In *ICSR*, 2008.

[23] H. Liu and H. Motoda. *Feature Selection for Knowledge Discovery and Data Mining*. Kluwer Academic Publishers, 1998.

[24] S. Livieri, Y. Higo, M. Matushita, and K. Inoue. Very-large scale code clone analysis and visualization of open source programs using distributed CCFinder: D-CCFinder. In *ICSE*, 2007.

[25] C. J. V. Rijsbergen. *Information Retrieval*. Butterworth-Heinemann, 1979.

[26] 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.*, 74(7):470–495, 2009.

[27] T. Seidl and H.-P. Kriegel. Optimal multi-step k-nearest neighbor search. In *SIGMOD*, 1998.

[28] A. Silberschatz, P. B. Galvin, and G. Gagne. *Operating System Concepts*. Wiley Publishing, 8th edition, 2008.

[29] V. Wahler, D. Seipel, J. W. v. Gudenberg, and G. Fischer. Clone detection in source code by frequent itemset techniques. In *SCAM*, 2004.

[30] T. Xie, M. Acharya, S. Thummalapenta, and K. Taneja. Improving software reliability and productivity via mining program source code. In *NSFNGS*, 2008.

[31] B.-K. Yi and C. Faloutsos. Fast time sequence indexing for arbitrary lp norms. In *VLDB*, 2000.

[32] K. Zhang and D. Shasha. Simple fast algorithms for the editing distance between trees and related problems. *SIAM J. Comput.*, 18(6):1245–1262, 1989.