Online Appendix to: Analysis and Optimization for Boolean Expression Indexing

Mohammad Sadoghi, University of Toronto
Hans-Arno Jacobsen, University of Toronto

The core part of this appendix is dedicated to BE-Tree deletion algorithm, detailed correctness proofs for BE-Tree, and additional experiments including comparison with state-of-the-art GPU matching algorithms.

A. STRINGS TO BOOLEAN EXPRESSIONS CONVERSION

String tokenization using $q$-grams maps the string into a high-dimensional vector space model, in which the domain of each dimension is binary. The size of this space is exponential in the length of $q$-grams. For instance, $q$-grams of size three results in a space with $26^3$ dimensions.

The vector space model representation of a tokenized string (e.g., \{‘str’, ‘tri’, ‘rin’, ‘ing’\}) can be expressed by setting dimensions associated to each of its grams (e.g., ‘str’, ‘tri’, ‘rin’, and ‘ing’) to 1 and everything else to 0. Alternatively, we can concisely express this as binary equality predicates, e.g., the predicate with dimension (i.e., attribute) ‘str’ must be equal to 1 and ignoring dimension with value 0. Therefore, instead of expressing a tokenized string as a vector of 0s or 1s, we can express it as a set of equality predicates. We can also take this one step further by going beyond binary domains to capture more interesting relationships among $q$-grams and as a byproduct reducing the space dimensionality.

The original model mapped each $q$-gram to a dimension, instead we can map only the prefix of each $q$-gram to a dimension and map the rest as a value in the corresponding dimension. For example, the $q$-gram ‘str’ is mapped to the quality predicate \[’st’ = ’r’\], ‘st’ now represents the predicate’s dimension and ‘r’ is mapped to the value in this dimension.

In this new mapping, the number of dimension is reduced from $26^3$ to $26^2$, and there is a new opportunity to express similarity among overlapping $q$-grams. For example, the $q$-grams ‘str’ and ‘ste’ are now both mapped to dimensions represented by ‘st’ (already signifying a similarity among these two $q$-grams), but also the value ‘r’ and ‘e’ can play an important role. For instance, since the letters ‘r’ and ‘e’ are adjacent on standard U.S. keyboard, then it is possible, that due to typing error $q$-grams ‘str’ was entered as ‘ste’. This relationship can be captured by mapping ‘r’ to ‘e’ to spatially close values.

B. BE-TREE DELETION ALGORITHM

Deleting subscription consists of two routines: (1) $\text{DeleteBE}\text{Tree}$ (Algorithm A) which checks all subscriptions in a leaf node and traverses through BE-Tree’s $p$-directory and (2) $\text{SearchDeleteCDir}$ (Algorithm A) which traverses through BE-Tree’s $c$-directory.

$\text{DeleteBE}\text{Tree}$ algorithm takes as inputs: a unique subscription to be removed, a $c$-node (BE-Tree’s root initially). The algorithm, first, checks all subscriptions in the $c$-node’s leaf searching for the subscription to be removed (Line 1). Second, for every $attr_i$ in the subscription’s predicates, it searches the $c$-node’s $p$-directory (Line 6). Next, the algorithm calls $\text{SearchDeleteCDir}$ on all
Algorithm 1 DeleteBETree\((\text{sub}, \text{cnode})\)

1: \(\text{isFound} \leftarrow \text{deleteSubFromLeaf}(\text{cnode.lnode})\)
2: if \(\text{isFound}\) then
3:   \{Iterate through subscription’s predicates\}
4:   for \(i \leftarrow 1\) to \(\text{NumOfPred}(\text{sub})\) do
5:     \(\text{attr} \leftarrow \text{sub.pred}[i].\text{attr}\)
6:     \(\text{pnode} \leftarrow \text{SearchPDir}(\text{attr}, \text{cnode.pdir})\)
7:     \{If attr exists in the p-directory\}
8:     if \(\text{pnode} \neq \text{NULL}\) then
9:       \(\text{isFound} \leftarrow \text{SearchDeleteCDir}(\text{sub}, \text{pnode.cdir})\)
10:      if \(\text{isFound}\) then
11:         break
12:   \}
13: else if \(\text{isEmpty}(\text{pnode})\) then
14:       \(\text{removePNode}(\text{pnode})\)
15:   if \(\text{isEmpty}(\text{cnode.pdir})\) then
16:       \(\text{removePDir}(\text{cnode.pdir})\)
17:   if \(\text{isEmpty}(\text{cnode.lnode})\) then
18:       \(\text{removeLNode}(\text{cnode.lnode})\)
19: else if \(\text{isEmpty}(\text{cnode})\) then
20:       \(\text{removeCNode}(\text{cnode})\)
21: return \(\text{isFound}\)

Algorithm 2 SearchDeleteCDir\((\text{sub}, \text{cdir})\)

1: if \(\text{IsEnclosed}(\text{event}, \text{cdir.lChild})\) then
2:   \(\text{isFound} \leftarrow \text{SearchDeleteCDir}(\text{sub}, \text{cdir.lChild})\)
3: else if \(\text{IsEnclosed}(\text{event}, \text{cdir.rChild})\) then
4:   \(\text{isFound} \leftarrow \text{SearchDeleteCDir}(\text{sub}, \text{cdir.rChild})\)
5: else
6:   \(\text{isFound} \leftarrow \text{DeleteBETree}(\text{sub}, \text{cdir.cnode})\)
7: if \(\text{isFound}\) then
8:   if \(\text{isEmpty}(\text{cdir.lChild})\) then
9:     \(\text{removeBucket}(\text{cdir.lChild})\)
10: if \(\text{isEmpty}(\text{cdir.rChild})\) then
11:     \(\text{removeBucket}(\text{cdir.rChild})\)
12: if \(\text{isEmpty}(\text{cdir})\) then
13:     \(\text{removeCDir}(\text{cdir})\)
14: return \(\text{isFound}\)

relevant p-nodes (Line 8). Now at any point, if the subscription is found (Line 9), the search for the subscription is stopped. Once the subscription is found and removed, then the algorithm enters the garbage collection mode and begins removing any empty nodes and directories (Lines 11-19). First, it removes any empty p-node that held the deleted subscription (Line 13). After removing the p-node, if the p-directory becomes empty, then the p-directory is also removed (Line 15). Next if the l-node of the current c-node is empty, then the l-node is also removed (Line 17). Finally, once l-node and p-directory for the current c-node are removed, the c-node itself is also removed (Line 19). Notably, there is no difference between removing a c-node and a p-node; once a node becomes empty (having no descents), then it is simply removed.

SearchDeleteCDir takes as inputs: a unique subscription to be removed, and a c-directory. The algorithm is as follows: it recursively calls SearchDeleteCDir on the bucket’s left child if the left child encloses the subscription (Line 2) or on the bucket’s right child if the right child encloses the subscription (Line 4). If neither left nor right buckets enclose the subscription, then the algorithm
calls $\text{DeleteBETree}$ on the $c$-node of the current $c$-directory (Line 6). Once the subscription to be removed is found, then any empty left or right buckets and $c$-directory are removed (Lines 7-13).

C. BE-TREE PROOF OF CORRECTNESS

We first prove a general property, namely, no deadlock occurs in the insertion process; thus, there always exists a bucket in the clustering directory that can accommodate newly inserted expressions.

**Lemma C.1.** A leaf bucket is either an open or a discrete bucket.

**Proof.** There are two main operations: (1) bucket splitting that is the result of insertions and (2) bucket merging that is the result of deletions. In the case of bucket splitting, Lemma C.1 follows simply from the forced split rule because a bucket that is associated with a partitioned $c$-node, must have been split once, implying that it cannot be a leaf unless it is a discrete bucket. Therefore, if a leaf bucket is not discrete, then it is always open because it has no children and has not yet been split. In the case of bucket merging, if a bucket is underflowing, then it can be merged with its immediate parent only if the parent is an open bucket resulting, again, in a single open leaf bucket.

**Lemma C.2.** A discrete and non-leaf bucket is always the home bucket for all the expressions that it is hosting, i.e., the smallest bucket that can enclose them.

**Proof.** Since a discrete bucket is the smallest possible bucket in the clustering directory, it is always the home bucket for its expressions. A non-leaf bucket by definition must have at least two children, and given the insertion rule, it follows that a non-leaf bucket is always the home bucket for its expressions.

**Theorem C.3.** The insertion of an expression incurs no deadlock, namely, there exists either the home bucket for the expression or an open leaf bucket that encloses it.

**Proof.** If there exists a home bucket for an expression, then the correctness is trivial because the expression is simply inserted into the home bucket, and there is no deadlock. However, if the home bucket does not exist, then we must show that there exists at least an open bucket that encloses it. In fact, a much stronger claim can be made which is that if the home bucket does not exist, then there is exactly one open leaf bucket, a unique non-discrete leaf bucket, that encloses it. If the home bucket does not exist, then there must be exactly one leaf bucket that encloses its home bucket due to the recursively splitting the space in half. Since the leaf bucket encloses the expression’s home bucket, it is not a discrete bucket. Therefore, based on Lemma C.1, the leaf bucket must be an open leaf bucket. Hence, no deadlock exists.

Next, we define the notions of safe partitioning and clustering, i.e., avoiding the cascading split problem, followed by a proof for the invariance of BE-Tree that guarantees safe partitioning and clustering.

**Theorem C.4.** A $c$-node clustering, i.e., bucket splitting, is safe (avoids the cascading split problem) only if the $c$-node is associated with an open bucket.

**Proof.** The proof follows simply from the fact that the $c$-node of an open bucket is not partitioned yet. Hence, the $c$-node clustering will not result in the cascade splitting problem and the clustering is safe.

**Theorem C.5.** A $c$-node partitioning is safe (avoids the cascading split problem) only if it is associated with a bucket that is either a discrete bucket or non-leaf bucket.

**Proof.** Based on Lemma C.2, a discrete and non-leaf bucket is always the home bucket for all the expressions that it is hosting. The safe partition proof follows from the fact that there is no benefit or need to further split a bucket that is the home to all of its expressions. Therefore, the cascading split problem is always avoided. Hence, $c$-node partitioning of a discrete bucket or non-leaf bucket is always safe.
INVARiance. Every expression always resides in the smallest bucket that encloses it, and the c-node of a non-discrete leaf bucket is never partitioned.

Proof. It follows from Lemma C.2 that a discrete and non-leaf bucket is always the home bucket for all the expressions that it is hosting, i.e., the smallest bucket that can enclose the expressions. We also need to show this for expressions that reside in a non-discrete leaf bucket, which trivially follows from the definition of a leaf bucket. A leaf bucket is at the lowest level of the clustering directory having no children. Therefore, no other smaller bucket exists in the directory to enclose the non-discrete leaf bucket’s expressions. Hence, every expression always resides in the smallest bucket that encloses it.

It follows from Lemma C.1 that a non-discrete leaf bucket is always open. Hence, the c-node of a non-discrete leaf bucket is never partitioned.

Theorem C.6. BE-Tree’s two-phase space-cutting (space partitioning and clustering) is always safe and always satisfies the BE-Tree invariance.

Proof. It follows from the invariance that every expression always resides in the smallest bucket that encloses it. Thus, a discrete and non-leaf bucket is always the home bucket for all the expressions that it is hosting. Therefore, partitioning a discrete and non-leaf bucket is always safe according to Theorem C.5. In addition, the invariance also enforces that a c-node of a non-discrete leaf bucket is never partitioned because it is not necessarily the home bucket for all of its expressions.

It follows from Theorem C.4 that the space clustering is trivially always safe if it applies to only open buckets.

D. BE-TREE HEIGHT COMPLEXITY PROOF

Theorem D.1. The height of BE-Tree is bounded by \( O(k \log N) \).

Proof. The number of p-nodes along each path to an l-node is bounded by the number of predicates in each expression. Hence, if there are at most \( k \) predicates in each expression, then there are at most \( k \) p-nodes along each path. Furthermore, the clustering directory is bounded by \( O(\log N) \), where \( N \) is the cardinality of the domain. Consequently, the height of BE-Tree is bounded by \( O(k \log N) \), where \( k \) is the maximum number of predicates per expression.

E. BE-TREE OPERATIONS COMPLEXITY PROOF

Theorem E.1. The cost of BE-Tree insertion is bounded by \( O(k \log N) \), which is a single-path traversal operation.

Proof. The BE-Tree insertion algorithm chooses the node with the highest score at every level of the tree, which results in a single-path tree traversal. Since the height of the tree is bounded by \( O(k \log N) \), then it simply follows that the BE-Tree insertion is also bounded by \( O(k \log N) \).

Theorem E.2. The BE-Tree matching (i.e., searching) algorithm is a multi-path traversal operation.

Proof. Similar to most multi-dimensional algorithms (such as R-Tree), BE-Tree must follow all relevant paths when matching an event (i.e., searching for all matching subscriptions). Thus, in the worst case, the entire tree must be traversed.

Theorem E.3. The BE-Tree’s deletion algorithm is a multi-path traversal operation.

Proof. Similar to most multi-dimensional algorithms (such as R-Tree), BE-Tree must follow all relevant paths when searching for a subscription to be deleted. Thus, in the worst case, the entire tree must be traversed.

Theorem E.4. The deletion cost of augmented BE-Tree is bounded by \( O(k \log N) \), which is a single-path traversal operation.
PROOF. Each subscription is associated with exactly one leaf node. Thus, if the subscription identifier and the leaf node holding it, i.e., the pair \((\text{subscription id}, \text{l-node})\), are maintained in a hashtable (having \(O(1)\) access time), then the deletion operation can efficiently be implemented using exactly a single-path traversal. As a result, once the subscription’s leaf node is identified using the hashtable and removed, then this information must propagated upward to the root. This backtracking requires following a single path, and given that the longest path is bounded by \(O(k \log N)\), then the deletion cost of the augmented \(\text{BE-Tree}\) is bounded by \(O(k \log N)\). \(\square\)

**Theorem E.5.** The cost of \(\text{BE-Tree}\) update is bounded by \(O(k \log N)\), which is a single-path traversal operation.

**Proof.** The update algorithm is implemented as a deletion operation, i.e., a single-path traversal operation bounded by \(O(k \log N)\), followed by an insertion operation, i.e., a single-path traversal operation bounded by \(O(k \log N)\); hence, it follows that \(\text{BE-Tree}\) update cost is also bounded by \(O(k \log N)\). \(\square\)

F. ADDITIONAL EXPERIMENTAL RESULTS

In this section, we explore the effect of matching probability as it reaches 50%, and the robustness of sequential \(\text{BE-Tree}\) compared to parallel hardware acceleration using GPUs.

F.1. Effect of Percentage of Matching Probability

The effect of leaf node capacity as increasing matching probability is shown in Figure 1.

Fig. 1. Extended Matching Probability with Different l-node Capacities

The extended matching probability results are shown in Figure 2, we observed that \(\text{BE-Tree}\) is dominant when increasing the matching probability up to 50%.
F.2. Effect of GPUs Hardware Acceleration and Location-based Matching

In this final experiment, two major algorithms were considered. Our PC-based algorithm, namely, BE-Tree, a single-threaded algorithm, and a GPU-based algorithm, for which we have drawn results from the GPU-based algorithm CLCB ran on Nvidia GTX 460 with 1GB of memory using the CUDA Toolkit 4.1 [Margara and Cugola 2011; Cugola and Margara 2012; Margara and Cugola 2013].

We experiment with two of our publicly released BE-Tree algorithms. BE-Tree 1.1 (which includes our bitmap optimization), whereas BE-Tree 1.3\(^1\), which further improves low-level implementations and includes both bitmap optimization and 2-dimensional subscription representation. The Bloom filter optimization is not included in BE-Tree 1.3.

We rely on default location-based workloads used in [Cugola and Margara 2012], for serving as a common benchmark. The workload consists of executing 1000 events over 2.5 millions subscriptions. Both events and subscriptions consist of 3-5 predicates drawn uniformly from 100 dimensions, where the cardinality of each domain is 65K. The subscriptions and events in the location-based workload have only 2-4 regular predicates plus an additional location-based predicate. Since the location predicates generated in [Cugola and Margara 2012] have higher selectivity, the average number of matched subscriptions are increased from 3 to 70 when moving from the workloads without to with location predicates.

The experimental results are summarized in Table I. The CLCB algorithm (ran on GPUs) on average processes each event in 0.306\(\,ms\) for location-based workloads while BE-Tree 1.3 can a sustain an average matching time of 0.045\(\,ms\) for non-location-based workload and matching time of 0.067\(\,ms\) for location-based workload (nearly 5x faster than CLCB). The slight increase in BE-Tree matching is due to increased number of matched subscriptions.

Therefore, contrary to the reported results in [Cugola and Margara 2012; Margara and Cugola 2013] (based on BE-Tree 1.1), in fact, not only BE-Tree’s matching time does not deteriorate when adding location-based information (because BE-Tree makes no distinction between location

\(^1\)The BE-Tree 1.1 and 1.3 binaries are available on http://www.cs.toronto.edu/~mo/projects.html
and non-location-based predicates and BE-Tree’s underlying novel two-phase space-cutting technique can utilize all types of predicates), but BE-Tree also outperforms the CLCB algorithm ran on GPUs [Margara and Cugola 2011; Cugola and Margara 2012; Margara and Cugola 2013]. The incorrect BE-Tree results reported in [Cugola and Margara 2012; Margara and Cugola 2013] was due to supplying BE-Tree with wrong parameters, not visible to the authors in [Cugola and Margara 2012; Margara and Cugola 2013]; thus, the authors were unaware of this misconfiguration [Cugola and Margara 2012; Margara and Cugola 2013].

### Table I. Comparing BE-Tree (PC) and CLCB (GPU)

<table>
<thead>
<tr>
<th>Workload Type</th>
<th>BE-Tree 1.1</th>
<th>BE-Tree 1.3</th>
<th>CLCB</th>
</tr>
</thead>
<tbody>
<tr>
<td>without location</td>
<td>0.081 ms</td>
<td>0.045 ms</td>
<td>N/A</td>
</tr>
<tr>
<td>with location</td>
<td>0.144 ms</td>
<td>0.067 ms</td>
<td>0.306 ms</td>
</tr>
</tbody>
</table>

ACM Transactions on Database Systems, Vol. V, No. N, Article A, Publication date: January YYYY.