BE-Tree: An Index Structure to Efficiently Match Boolean Expressions over High-dimensional Discrete Space

Mohammad Sadoghi
Department of Computer Science
University of Toronto, Canada
mo@cs.toronto.edu

Hans-Arno Jacobsen
Dept. of Electrical and Computer Engineering
University of Toronto, Canada
jacobsen@eecg.toronto.edu

ABSTRACT

BE-Tree is a novel dynamic tree data structure designed to efficiently index Boolean expressions over a high-dimensional discrete space. BE-Tree copes with both high-dimensionality and expressiveness of Boolean expressions by introducing a novel two-phase space-cutting technique that specifically utilizes the discrete and finite domain properties of the space. Furthermore, BE-Tree employs self-adjustment policies to dynamically adapt the tree as the workload changes. We conduct a comprehensive evaluation to demonstrate the superiority of BE-Tree in comparison with state-of-the-art index structures designed for matching Boolean expressions.

Categories and Subject Descriptors

H.3.3 [**Information Search and Retrieval**]: Information filtering

General Terms

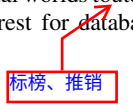
Algorithms, Design, Measurement, Experimentation, Performance

Keywords

Boolean expressions, Data Structure, Algorithm, Publish/Subscribe, and Complex Event Processing

1. INTRODUCTION

The efficient indexing of Boolean expressions is a common problem at the center of a number of data management applications. For example, for event processing and publish/subscribe, Boolean expressions represent events and subscriber interests [2, 13, 6, 26], for online advertising and information filtering, Boolean expressions represent advertiser profiles and filters [26, 14], for algorithmic trading, they represent market events and investment strategies [25], and for approximate string matching, they represent string patterns using q-grams [8]. In all scenarios, key challenges are the scaling to millions of expressions and to sub-second matching latency. We use a data management scenario for co-spaces as in-depth example. Co-spaces are an emerging concept to model the co-existence of physical and virtual worlds ~~touted~~ by the Claremont Report as an area of rising interest for database researchers [1].



Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SIGMOD'11, June 12–16, 2011, Athens, Greece.
Copyright 2011 ACM 978-1-4503-0661-4/11/06 ...$10.00.

Consider, for example, a mobile shopping application, where a shopper enters a physical mall and her mobile device submits her shopping preferences (i.e., subscriptions) to the virtual mall database: [*genre* = classics, *era* $\in \{1920s, 1940s\}$, *price* BETWEEN $[20, 40]$, *ranking* < 5, *format* $\notin \{$paperback$\}$]. Now, assume a new promotion (i.e., an event) matches the shopper's interests and the item detail is pushed to her mobile device. An example of a matching item is as follows: [*genre* = classics, *title* = '*The Sun Also Rises*', *author* = Hemingway, *era* = $1920s$, *price* = 26, *ranking* = 2, *format* = hardcover]. In the example, both the shopper's interest and the promotional item are defined over multiple attributes (i.e., dimensions in space) such as *genre* and *price* in which each attribute has a discrete and finite domain. Furthermore, each attribute of interest is constrained to a set of values with an operator. The triple consisting of attribute, operator, and set of values is referred to as a *Boolean predicate*. A conjunction of Boolean predicates, which here represents both the shopper's interest and the promotional item, is a *Boolean expression*.

There are four major challenges to efficiently index Boolean expressions. First, the index must scale to millions of Boolean expressions defined over a high-dimensional space and afford efficient lookup (i.e., expression matching.) Second, the index must support predicates with an expressive set of operators. Third, the index must enable dynamic insertion and deletion of expressions. Lastly, the index must adapt to changing workload patterns and expression schemata.

However, existing techniques are inadequate to satisfy these four requirements. For instance, techniques used in expert and rule-based systems support expressive predicate languages [15], but are unable to scale to millions of expressions. Recent work addresses the scalability limitation, but either restricts the predicate expressiveness [13] or assumes a static environment in which the index is constructed offline [2, 26, 14]. Our goal is to address scalability, expressiveness, dynamic construction, and adaptation by proposing a novel, self-adjusting index structure that is specifically geared towards high-dimensionality over discrete and finite domains. To achieve these goals, we propose **BE**-Tree a novel tree structure to efficiently index and match large sets of **B**oolean **E**xpressions defined over an expressive predicate language in a high-dimensional space. BE-Tree is dynamically constructed through a novel two-phase space-cutting technique that exploits the discrete and finite structure of both the subscription and event space. Another distinct feature of BE-Tree is a novel self-adjusting mechanism that adapts as subscription and event workloads change.

We make the following five contributions in this paper:

1. Unify subscription and event language to enable a more expressive matching semantics (Sec. 3),
2. propose a novel data structure, BE-Tree, that supports an extensive set of operators and dynamic expression schemata,

gracefully scales to millions of subscriptions, thousands of dimensions, and dozens of predicates per subscription and event (Sec. 4),

3. develop a novel self-adjusting mechanism for BE-Tree that continuously adapts to both subscription and event workload changes (Sec. 5),
4. present the first comprehensive evaluation framework that benchmarks state-of-the-art matching algorithms, including SCAN [27], SIFT [27], Gryphon [2], Propagation [13], and k-index [26] (Sec. 7), and
5. present a novel approach to leverage real-world high-dimensional datasets for evaluation (Sec. 7).

2. RELATED WORK

Problems related to indexing Boolean expressions have been studied in many contexts: Expert systems [15], active databases [19, 10], trigger processing [18, 7], publish/subscribe matching [27, 2, 13, 6, 5, 26, 14], XPath/XML matching (e.g.,[11, 23]), and stream processing (e.g., [12, 9]). Indexing in multi-dimensional space has also been extensively studied (e.g., [17, 3, 4].)

The work on expert systems, active databases, and trigger processing [15, 19, 10, 18, 7] as well as certain publish/subscribe (pub/sub) systems (e.g., [5]) focus on language expressiveness and not on scaling to thousands of dimensions and millions of expressions. In addition, most XPath/XML matchings are based on a completely different language from what BE-Tree supports and is not in the scope of this work (e.g., [11]). These approaches are therefore not directly applicable, and our review concentrates on pub/sub matching algorithms [27, 2, 13, 6, 22, 20, 21, 26, 14].

Two main categories of matching algorithms have been proposed: counting-based [27, 13, 26] and tree-based [2, 6, 21] approaches. These approaches can further be classified as either key-based, in which for each expression a set of predicates are chosen as identifier [13], or as non-key based [27, 6, 26]. Counting-based methods aim to minimize the number of predicate evaluations by constructing an inverted index over all unique predicates. The two most efficient counting-based algorithms are Propagation [13], a key-based method, and the k-index [26], a non-key-based method. Likewise, tree-based methods are designed to reduce predicate evaluations and to recursively divide the search space by eliminating subscriptions on encountering unsatisfiable predicates. The most prominent tree-based method, Gryphon, is a static, non-key based algorithm [2]. Our BE-Tree is a novel tree-based approach, which also employs keys, that we show to outperform existing work [27, 2, 13, 26].

The latest advancement of counting-based algorithms is k-index [26], which gracefully scales to thousands of dimensions and supports equality predicates ($\in$) and non-equality predicates ($\notin$). k-index partitions subscriptions based on their number of predicates to prune subscriptions with too few matching predicates; however, k-index is static and does not support dynamic insertion and deletion. What distinguishes our BE-Tree from k-index is that BE-Tree is fully dynamic, naturally supports richer predicate operators (e.g. range operators), and adapts to workload changes.

Another emerging area of research is to improve language expressiveness. For example, the subscription languages in [6, 14] support matching complex Boolean expression, while k-index supports either Conjunctive Normal Form or Disjunctive Normal Form [26]. Most important, the latest technique that supports matching complex Boolean expressions relies on existing matching algorithm to match the low-level conjunctive expressions [14]; thus, it ~~dovetails~~ well with BE-Tree. Finally, there is a paradigm shift, manifest in what is referred to as symmetric pub/sub, in which event producers are also able to impose filtering conditions on event subscribers [22, 20], which substantially improves the expressive power of the event language. This new paradigm is also supported by our BE-Tree and proposed matching semantics [24].

Finally, matching in high-dimensional space diverges from classical database indexing in four crucial ways. First, BE-Tree has to cope with data of much higher dimensionality (order of thousands), that is orders of magnitude larger than capabilities of existing high-dimensional indexing structures [17, 3]. Second, Expressions indexed by BE-Tree impose restrictions only on a small subspace and are fully defined everywhere else. As a result, there is a high degree of overlap among expressions which renders current indexing techniques inapplicable. For instance, X-Tree [3], often the most suitable index for high-dimensional data, degenerates to a sequential scan when all expressions overlap. Third, classical high-dimensional indexing focuses on utilizing disk space and reducing random accesses, as opposed to directly optimizing the matching process (lookup time); disk is assumed as storage medium and disk I/O is the bottleneck. BE-Tree, on the other hand, is a main memory structure. Forth, BE-Tree aims to support discrete, finite domains, while many high-dimensional indexing structures are designed for continuous unbounded domains that are unable to benefit from the finite and discrete domain properties.

3. EXPRESSION MATCHING MODEL

Expression Language Traditionally, pub/sub matching algorithms take as input a set of subscriptions (a conjunction of Boolean predicates) and an event (an assignment of a value to each attribute), and return the subset of subscriptions satisfied by the event. Unlike most existing work, we model both subscriptions and events as Boolean expression. This generalization gives rise to more expressive matching semantics while still encompassing the traditional pub/sub matching problem. Each Boolean expression is a conjunction of Boolean predicates. A predicate is a triple, consisting of an attribute uniquely representing a dimension in n-dimensional space, an operator, and a set of values, denoted by $P^{\texttt{attr},\texttt{opt},\texttt{val}}(x)$, or more concisely as $P(x)$. A predicate either accepts or rejects an input x such that $P^{\texttt{attr},\texttt{opt},\texttt{val}}(x) : x \longrightarrow \{\texttt{True},\texttt{False}\}$, where $x \in \texttt{Dom}(P^{\texttt{attr}})$ and $P^{\texttt{attr}}$ is the predicate's attribute. Formally, a Boolean expression be is defined over an n-dimensional space as follows:

$$be = \{P_1^{\texttt{attr},\texttt{opt},\texttt{val}}(x) \wedge \cdots \wedge P_k^{\texttt{attr},\texttt{opt},\texttt{val}}(x)\},$$
$$where\ k \leq n;\ \ i,j \leq k,\ P_i^{\texttt{attr}} = P_j^{\texttt{attr}}\ \ iff\ \ i = j$$

We support an expressive set of operators for the most common data types: relational operators ($<, \leq, =, \neq, \geq, >$), set operators ($\in, \notin$), and the SQL BETWEEN operator.

Matching Semantics Our unique formulation of subscriptions and events as expressions enables us to support a wide range of matching semantics. We start with the classical pub/sub matching problem: *Given an event e and a set of subscriptions, find all subscriptions s_i satisfied by e.* We refer to this problem as *stabbing subscription*[1] $\texttt{SQ}(e)$, and specify the problem as follows:

$$\texttt{SQ}(e) = \{s_i |\ \forall P_q^{\texttt{attr},\texttt{opt},\texttt{val}}(x) \in s_i,\ \exists P_o^{\texttt{attr},\texttt{opt},\texttt{val}}(x) \in e,$$
$$P_q^{\texttt{attr}} = P_o^{\texttt{attr}}, \exists x \in \texttt{Dom}(P_q^{\texttt{attr}}),\ P_q(x) \wedge P_o(x)\}$$

A complete list of matching semantics supported by BE-Tree is presented in the extended paper [24], our supported semantics also includes the symmetric pub/sub matching [22, 20].

[1]This is a generalization of stabbing query, which determines which of a collection of intervals overlap a query point.

吻合

4. BE-TREE ORGANIZATION

BE-Tree dynamically indexes large sets of expressions (i.e., subscriptions) and efficiently determines which of these expressions match an input expression (i.e., event). BE-Tree supports Boolean expressions with an expressive set of operators defined over a high-dimensional space. The main challenge in indexing a high-dimensional space is to effectively cut the space in order to prune the search at lookup time. BE-Tree copes with this challenge—the curse of dimensionality—through a novel two-phase space-cutting technique that significantly reduces the complexity and the level of uncertainty of choosing an effective criterion to recursively cut the space and to identify highly dense subspaces. The two-phases BE-Tree employs are: (1) *space partitioning* which is the global structuring to determine the best splitting $\texttt{attr}_i$, i.e., the i^{th} dimension (Sec. 4.2) and (2) *space clustering* which is the local structuring for each partition to determine the best grouping of expressions with respect to the expressions' range of values for $\texttt{attr}_i$ (Sec. 4.3).

This two-phase approach, the space partitioning followed by the space clustering, introduces new challenges such as how to determine the right balance between the space partitioning and clustering, and how to develop a robust principle to alternate between both. These new challenges are addressed in BE-Tree by exploiting the underlying discrete and finite domain properties of the space. We begin by discussing the structure and the dynamics of BE-Tree before presenting the main design principles behind BE-Tree. All these are prerequisites to the actual, but much simpler, expression matching with BE-Tree, described in Sec. 6.

4.1 BE-Tree Structure

BE-Tree is an n-ary tree structure in which a leaf node contains a set of expressions and an internal node contains partial predicate information (e.g., an attribute and a range of values) about the expressions in its descendant leaf nodes. We distinguish among three classes of nodes: a partition node (p-node) which maintains the space partitioning information (an attribute), a cluster node (c-node) which maintains the space clustering information (a range of values), and a leaf node (l-node) which stores the actual expressions. Moreover, p-nodes and c-nodes are organized in a special directory structure for fast space pruning. Thus, a set of p-nodes are organized in a *partition directory* (p-directory), and a set of c-nodes are organized in a *cluster directory* (c-directory.) Before giving a detailed account of each node type and the BE-Tree dynamics, we outline the structural properties of BE-Tree as shown in Fig. 1, and give an example showing the overall dynamics of BE-Tree.

Example *Initially,* BE-Tree *has an empty root node which consists of a c-node that points only to an l-n... Upon arrival, new expressions (subscriptions) are inserted into the root's l-node, and once the size of the l-node exceeds the leaf capacity—a tunable system parameter—the space partitioning phase is triggered and a new* $\texttt{attr}_i$ *for splitting the l-node is chosen. The new* $\texttt{attr}_i$ *results in the creation of a new p-node. The* $\texttt{attr}_i$ *is chosen based on statistics gathered from expressions (subscriptions) in the overflowing l-node. The selected attribute is passed on to the space clustering phase that divides the domain of the* $\texttt{attr}_i$ *into a set of intervals, in which each range of values is assigned to a new c-node, and all the expressions having a predicate on* $\texttt{attr}_i$*, in the overflowing l-node, are distributed across these newly created c-nodes based on the c-nodes' range of permitted values. In brief,* BE-Tree *recursively partitions and clusters the space, which together recursively identify and refine dense subspaces, in order to maintain the size of each l-node below a threshold.*

Strictly speaking, in BE-Tree, each p-node is assigned an $\texttt{attr}_i$ such that all the expressions in its descendant l-nodes must have a predicate defined over $\texttt{attr}_i$. Similarly, each c-node is associated with a predicate $P_i^{\texttt{attr,opt,val}}(x)$ (i.e., a range of permitted values), and all the expressions in its descendant l-nodes must have a predicate $P_j^{\texttt{attr,opt,val}}(x)$ such that

$$(P_i^{\texttt{attr}} = P_j^{\texttt{attr}}) \wedge (\forall x \in \texttt{Dom}(P_j^{\texttt{attr}}),\ P_j(x) \rightarrow P_i(x)).$$

Provided that each c-node is denoted by a predicate $P_j(x)$, we can assign each l-node a key $\texttt{key}_\texttt{j}$ defined as a conjunction of all c-nodes' predicate along the path from the root to the l_j-node.

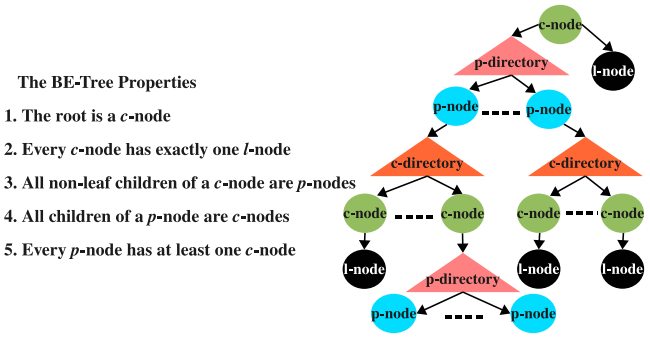


Figure 1: The BE-Tree data structure.

In what follows, in order to uniquely identify a node, a unique id is assigned to each node. For example, in order to refer to the j^{th} l-node, we write l_j-node. In addition, we adopt the following notation: $\texttt{max}_{cap}$ (maximum leaf capacity), $\texttt{min}_{size}$ (minimum partition size), $\texttt{max}_{cap}^j$ (l_j-node maximum capacity), $\texttt{rank}_{\texttt{window}}$ (window for updating the rank), and θ (l-node recycling threshold).

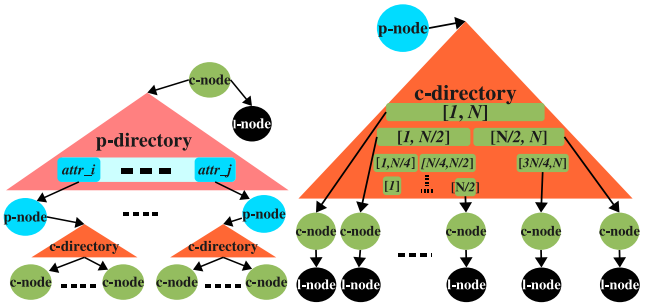
4.2 Space Partitioning

In BE-Tree, space partitioning, conceptually a global adjusting mechanism, is the first phase of our space-cutting technique. The space partitioning is triggered after an l_j-node overflows and uses a scoring function (cf. Sec. 5) to rank each candidate $\texttt{attr}_i$ in order to determine the best attribute for partitioning. Thus, the highest ranking attribute, appearing in at least $\texttt{min}_{size}$ number of expressions, implying that the attribute has a sufficient discriminating power, is chosen for the space partitioning phase. Essentially, this process identifies the next highest ranking dimension, only as the need arises, to segregate expressions into smaller groups based on a high-ranking attribute in order to prune the search space more effectively while coping with the curse of dimensionality.

Upon successful selection of an $\texttt{attr}_i$ for space partitioning, a new p-node for $\texttt{attr}_i$ is added to the parent of the overflowing l_j-node, and the set of expressions in the l_j-node is divided based on whether or not they have a predicate defined on $\texttt{attr}_i$. The partitioning procedure is repeatedly applied to the l_j-node to keep its size below $\texttt{max}_{cap}^j$.

The need for $\texttt{min}_{size}$ is to avoid ineffective partitioning. For instance, if none of the expressions in an l_j-node have a predicate on a common attribute, then there is no computational incentive to form a partition. Therefore, a natural problem that might arise in the space partitioning for any given l_j-node is the handling of scenarios for which no candidate attribute exists with size larger than $\texttt{min}_{size}$. For such cases, we introduce the notion of an extended l_j-node in which the size of the l_j-node, $\texttt{max}_{cap}^j$, is increased by a constant factor $\texttt{max}_{cap}$ such that $\texttt{max}_{cap}^j = \texttt{max}_{cap}^j + \texttt{max}_{cap}$. Thus, in order to support dynamic expansion and contraction of the leaf node size, after every successful partitioning, the l_i-node capacity is re-evaluated as follows:

$$\texttt{max}_{cap}^j = \begin{cases} \left\lceil \frac{|l_j\text{-node}|}{\texttt{max}_{cap}} \right\rceil \times \texttt{max}_{cap}, & \text{if } |l_j\text{-node}| > 0 \\ \texttt{max}_{cap}, & \text{otherwise} \end{cases}$$



(a) Partition directory (b) Clustering directory

Figure 2: The BE-Tree directories.

Another subtle point in the space partitioning is how to guide the attribute selection such that (1) it guarantees that subsequent space partitioning on lower levels of BE-Tree do not ineffectively cycle over a single $\texttt{attr}_i$ (2) it enables dynamic insertions and deletions without performance deterioration. To achieve these properties, we must ensure that in any path from the root to a leaf node, each $\texttt{attr}_i$ is selected at most once and that a deterministic clustering is employed after each partitioning (cf. Sec. 4.3). Moreover, we show in Sec. 4.3, the attribute selection restriction is not a limitation; in fact, we prove that it is sufficient to pick each $\texttt{attr}_i$ at most once yet fully exploiting the domain of $attr_i$ because when $\texttt{attr}_i$ is selected, then reselecting it at a lower level of the tree provides no additional benefit.

Moreover, it is evident that the number of partitions for each c-node grows linearly in the dimensionality of space. Since each edge from a c-node leading to a p-node is uniquely identified by a single attribute, we can employ a hash table over all edges leaving the node, potentially scaling BE-Tree to thousands of dimensions. The inner working of the partition directory is shown in Fig. 2a.

Finally, as the split operator is required to eliminate overflowing l-nodes, similarly, a merge operator is necessary to eliminate underflowing l-nodes. If an l-node is underflowing, then its contents are either merged with its grandparent's l-node or reinserted into BE-Tree. The latter approach is preferred because it provides an opportunity to avoid deterioration of BE-Tree. In any case, if an l-node is empty and its c-node has no other children, then the l-node is removed. This node removal naturally propagates upward removing any non-leaf nodes with no outgoing edges.

4.3 Space Clustering

Our proposed space partitioning reduces the problem of high-dimensional indexing into one-dimensional interval indexing. Interval indexing is addressed in our space clustering phase, conceptually a local adjusting mechanism. The key insight of our space clustering, and ultimately of our BE-Tree, is a deterministic clustering policy to group overlapping expressions (into regions) and a deterministic policy to alternate between the space partitioning and the space clustering. The absence of a predictable policy gives rise to the dilemma of whether to further pursue the space clustering or to switch back to the space partitioning. Besides, once a region is partitioned, that region can no longer be split without running into the cascading split problem [16]. Thus, a deterministic clustering policy that is influenced by the insertion sequence is either prone to ineffective regions that do not take advantage of the dimension selectivity to effectively prune the search space or prone to suffer from substantial performance overhead due to the cascading split problem. Therefore, to achieve determinism in our space clustering, while supporting dynamic insertion and deletion, our structure must be independent of insertion sequence.

Clustering Directory Organization To address these challenges, we propose a novel grid-based approach, with unique splitting and merging policies, to build the clustering directory in BE-Tree. The clustering directory is a hierarchical structure that organizes the space into sets of expressions by recursively cutting the space in halves. A key feature of this grid-based clustering is a novel forced split rule that (1) avoids the cascading split problem and that (2) enables deterministic clustering and partition-clustering alteration strategies that are independent of the insertion sequence. To describe the dynamics of our clustering directory, first, we formally define few concepts.

A *bucket* represents an interval boundary (range of values) over $attr_i$, and an expression is assigned to a bucket over $attr_i$ only if the set of values defined by the expression's predicate on $attr_i$ is covered by that bucket; for brevity, we say an expression is assigned to a bucket if the expression is enclosed by that bucket. Furthermore, a bucket has a minimal interval boundary which is a best-effort-smallest interval that encloses all of its expressions. Each bucket is associated with exactly one c-node in BE-Tree, which is responsible for storing and maintaining information about the bucket's assigned expressions. We further distinguish among four types of buckets: An *open bucket* that is a bucket with a not yet partitioned c-node; a *leaf bucket* that is a bucket that has no children (a bucket that has not been split); an *atomic bucket* that is a single-valued bucket that cannot further be split, an indivisible bucket; and a *home bucket* that is the smallest possible bucket that encloses an expression. Furthermore, an atomic bucket is also a leaf bucket, but the reverse is not necessarily true.

Essentially the clustering directory is constructed based on the following three rules to avoid the cascading split problem and to achieve the deterministic properties of BE-Tree: (1) an expression is always inserted into the smallest bucket that encloses it (*insertion rule*), (2) a non-atomic bucket (i.e., a multi-valued, divisible bucket) is always split before its c-node switches back to the space partitioning (*forced split rule*), and (3) an underflowing leaf bucket is merged with its parent only if the parent is an open bucket (*merge rule*).

The cascading split problem is avoided, first, due to the forced split rule because a bucket is always split before it is partitioned and, second, due to the insertion and merge rules because both current and future expressions are always placed in the smallest bucket that encloses them. Therefore, the partitioned c-node always remains the home bucket to all of its expressions. As a result, there is no benefit or need to further split a bucket that is the home to all of its expressions. This home bucket's uniqueness property is achieved through a rigorous splitting policy that deterministically cuts the space in half independent of the insertion sequence such that each expression could uniquely be associated to a *home bucket*.

The deterministic clustering is achieved through a grid-based organization of space in which each overflowing bucket is split in halves. The deterministic partition-clustering alteration is also achieved through the forced split rule which always enforces the split of non-atomic buckets before initiating the space partitioning. Thus, the space partitioning is always applied to expressions that are residing in their home bucket such that if the space clustering is further pursued, these expressions are unaffected by it.

缩拢

In short, the deterministic and no split cascading properties of BE-Tree are obtained by satisfying the below specified BE-Tree invariance. A complete proof of BE-Tree's invariance is presented in the technical report [24]. Most importantly, these desired properties are possible only due to the existence of an atomic bucket, which itself is possible only due to the underlying discrete domain property.

INVARIANCE. *Every expression always resides in the smallest*

bucket that encloses it, and the c-node of a non-atomic leaf bucket is never partitioned.

Subsequently, we present our unique clustering directory structure. We also propose a predicate transformation, which converts our expressive set of operators into an interval boundary that is compatible with our clustering directory.

Space Clustering Structure The main guiding principles of the BE-Tree space clustering are the insertion and the forced split rules. Both are used to dynamically construct BE-Tree as follows. The clustering directory starts with an empty top-level bucket which spans the entire domain for $\texttt{attr}_i$. Once the leaf node associated to the top-level bucket overflows, the bucket is split in half resulting in the creation of two new child buckets; each child bucket is also assigned a new c-node and l-node. Subsequently, expressions in the overflowed leaf node that are enclosed by either of the two child buckets are moved accordingly. This process is recursively applied until either an atomic bucket is reached or every bucket's c-node has its l_j-node below $\texttt{max}_{cap}^{j}$. Finally, an overflowing non-leaf or an atomic bucket is handled by switching to the partitioning mode. A snapshot of the clustering directory is shown in Fig. 2b.

In summary, the two-phase space-cutting begins with the space partitioning followed by a sequence of space clusterings until a safe point (no split cascading), i.e., a non-leaf or an atomic bucket, is reached at which point a fresh instance of the two-phase space-cutting, starting with the space partitioning, begins. Also, as explained in Sec. 4.2, each attribute is selected at most once in any path along the root of BE-Tree to a leaf node because switching back to the space partitioning occurs only at a safe point, in which the overflowing leaf node, which is subjected to partitioning, is associated with a home bucket, and a further clustering is no longer beneficial. In other words, once an attribute is chosen, before switching to the partitioning mode, the space clustering strategy will exploit the entire space to fully leverage the dimension selectivity independent of the insertion sequence.

The clustering directory supports indexing one-dimensional intervals. However, our predicate language supports a richer set of predicates. In Table 1, we show the conversion of predicates with different types of operators into one-dimensional intervals, where v_{min} and v_{max} are the smallest and the largest possible values in the domain, and $\{v_1, \cdots, v_k\}$ is sorted in ascending order. All predicates transformations are simple algebraic conversions except for the $(\neq, \notin, \in)$ operators. A predicate $P_j^{(attr_i, \neq, v_1)}(x)$ implies that every value in the domain of $attr_i$ is acceptable except for v_1. Therefore, under the uniform distribution assumption, the predicate $P_j(x)$ is satisfied with a high probability by an event expression having a predicate on $attr_i$. This observation supports the transformation of the $\neq$ operator into a one-dimensional interval $[v_{min}, v_{max}]$. This transformation results in an early pruning of expressions with a predicate on $\texttt{attr}_i$ during the matching of an event that does not have any predicate defined on $\texttt{attr}_i$. This pruning strategy is especially effective because the number of predicates per expression is on the order of tens while the number of space dimensions is on the order of thousands. Likewise, the $\notin$ operator is a generalization of the $\neq$ operator, which filters out a set of values from the domain instead of a single value. Thus, by converting $\notin$ to an interval that spans the entire domain, again, we are imposing a filtering strategy to effectively reduce the search space. Finally, applying a similar conversion to the $\in$ operator results in a further improved filtering strategy compared with the $(\neq, \notin)$ conversion because, now, the assigned interval is bounded by the minimum and the maximum values in the predicate with the $\in$ operator.

BE-Tree Operations The BE-Tree matching follows a typical tree traversal operation in which starting from the root multiple paths of a tree may be traversed until all relevant l-nodes are reached; the matching pseudocode is given in Sec. 6. In contrast, for insertion, the tree traversal follows exactly one path, which is explained in-depth in Sec. 6. The deletion algorithm is conceptually similar to the matching algorithm and is omitted for brevity.

Table 1: Transformations

Predicates	Interval-based
$i < v_1$	$[v_{min}, v_1 - 1]$
$i \leq v_1$	$[v_{min}, v_1]$
$i = v_1$	$[v_1, v_1]$
$i \neq v_1$	$[v_{min}, v_{max}]$
$i > v_1$	$[v_1 + 1, v_{max}]$
$i \geq v_1$	$[v_1, v_{max}]$
$i \in \{v_1, \cdots, v_k\}$	$[v_1, v_k]$
$i \notin \{v_1, \cdots, v_k\}$	$[v_{min}, v_{max}]$
i BETWEEN v_1, v_2	$[v_1, v_2]$

BE-Tree Complexity The height of BE-Tree is bounded by $O(k \log N)$, where k is the maximum number of predicates per expression and N is the domain cardinality. Thus, the height of BE-Tree, unlike for other tree-based matching structures such as Gryphon [2], does not grow in the number of subscriptions avoiding memory- and performance-detrimental tree degeneration; the complete proof is presented in the technical report [24].

不利的，有害的

5. BE-TREE SELF-ADJUSTMENT

BE-Tree self-adjustment is based on a novel cost-based ranking function and adaptation policies that utilize this ranking function.

Cost-based Ranking Function We present a novel ranking objective that directly reduces the matching cost as opposed to a ranking that is founded solely on popularity measures and, consequently, biased towards either the least or the most popular key [13], such ranking is a deviation from the actual index objective. Thus, we propose a ranking objective to directly reduce the matching cost based on the notion of *false candidates: the retrieved expressions (subscriptions) that are not matched by an input expression (events)*.

We formalize the index objective based on the matching cost as follows. The matching cost is defined as the total number of predicate evaluations broken down into *minimizing false candidate computations* and *minimizing true candidate computations*. The false candidate computation is the total number of predicate evaluations until an unsatisfied predicate is reached which discards the prior computations along the search path, therefore, penalizing multiple search paths of the tree that are discarded eventually. Also, the false candidate computation tracks the total number of predicates evaluated for each unsatisfied expression; therefore, penalizing keys that produce many false candidates. The true candidate computation is the number of predicate evaluations before reporting a set of expressions as matched, namely, promoting the evaluation of the common predicate exactly once.

We define a ranking model for each node in BE-Tree using our proposed matching cost. For an improved ranking accuracy, we also introduce the notion of *covered* and *subsumed* predicates. The covered predicates are defined as all predicates $P_l(x)$ in each l_j-node's expressions such that there exists a $P_i(x) \in \texttt{key}_\texttt{j}$ and $P_i^{\texttt{attr}} = P_l^{\texttt{attr}}$ because by the definition of the leaf node's key, all the $P_l(x)$ must be covered by $P_i(x)$. However, if $P_i(x)$ and $P_l(x)$ are also equivalent, i.e., $\forall x \in \texttt{Dom}(P_i^{\texttt{attr}})\ P_i(x) \leftrightarrow P_l(x)$, then $P_l(x)$ is considered subsumed and not covered. Thus, subsumed predicates are preferred because the covered predicates are approximate and must be re-evaluated at the leaf level for each expression. The ranking model assigns a rank to each node n_i using the function $Rank(n_i)$ which is a combination of the $Loss(n_i)$ and $Gain(l_j)$ functions. $Loss(n_i)$ computes for each node the false candidates generated over a window of m events. $Gain(l_j)$ is defined for each l_j-node, and it is the combination of the number of

Does this transformation introduce false positives?

Algorithm 1 `MatchBETree`($event$, $cnode$, $matchedSub$)

1: $matchedSub \leftarrow$ `CheckSub`($cnode.lnode$)
{Iterate through event's predicates}
2: **for** $i \leftarrow 1$ **to** `NumOfPred`($event$) **do**
3: $attr \leftarrow event.pred[i].attr$
{Check the c-node's p-directory (hashtable) for $attr$}
4: $pnode \leftarrow$ `SearchPDir`($attr$, $cnode.pdir$)
{If $attr$ exists in the p-directory}
5: **if** $pnode \neq$ `NULL` **then**
6: `SearchCDir`($event$, $pnode.cdir$, $matchedSub$)

subsumed and covered predicates for each of its expressions. Formally, $Rank(n_i)$, $Loss(n_i)$, and $Gain(l_j)$ are defined as follows:

$$Rank(n_i) = \begin{cases} \beta Gain(n_i) - \gamma Loss(n_i) & \text{if } n_i \text{ is a } l\text{-node} \\ (\sum_{n_j \in des(n_i)} Rank(n_j)) - \gamma Loss(n_i) & \text{otherwise,} \end{cases}$$

where $0 \leq \beta, \gamma \leq 1$, and $des(n_i)$ returns n_i's immediate descendants

$$Loss(n_i) = \sum_{e' \in window_m(n_i)} \frac{\#\ \texttt{discarded pred eval for}\ e'}{|window_m(n_i)|}$$

$$Gain(l_j) = \alpha_s Gain_s(l_j) + \alpha_c Gain_c(l_j),\ 0 \leq \alpha_s, \alpha_c \leq 1$$

$$Gain_s(l_j) = \#\ \texttt{subsumed pred},\quad Gain_c(l_j) = \#\ \texttt{covered pred}$$

The proposed ranking model is simply generalized for splitting an overflowing node l_j-node using a new $\texttt{attr}_i$, and it is given by

$$Rank(l_i) = \beta Gain(l_i) - \gamma Loss(l_i),\ \ where\ 0 \leq \beta, \gamma \leq 1,$$

where $Gain(l_i)$ is approximated by the number of expressions that have a predicate on $\texttt{attr}_i$ and $Loss(l_i)$ is estimated by constructing a histogram in which $Loss(l_i)$ is the average bucket size in the histogram. Essentially, the average bucket size estimates the selectivity of $\texttt{attr}_i$, meaning, in the worst case the number of false candidates is equal to the average bucket size. Alternatively, in an optimistic approach $Loss(l_i)$ is initially set to zero to eliminate any histogram construction and to rely on the matching feedback mechanism for adjusting the ranking if necessary. This optimistic approach initially estimates the popularity of $\texttt{attr}_i$ as opposed to selectivity of $\texttt{attr}_i$. Based on our experimental evaluation, the optimistic approach results in an improved matching and insertion time. Similarly, based on our empirical evidence, this model can be further simplified by setting all parameters β, γ, α_s, and α_c to 1; hence, eliminating them. This simplification is possible because nearly in all our experiments the rate of false candidate was sufficiently low such that altering the values of these parameters had no influence on the overall BE-Tree's matching rate.

Adaptation Policies In BE-Tree, we also consider three strategies to utilize our ranking function and to further avoid tree degeneration: recycling l-node, reinserting Boolean expression, and exploration vs. exploitation [24]; in the interest of space we only discuss the first property. Recycling l-node is another BE-Tree's self-adjusting policy that monitors each node over a window, $\texttt{rank}_{\texttt{window}}$, of the number of insertion, deletion, and matching operations. If at the end of each node's window, the rank of a node drops below the threshold, θ, then the entire contents of that node, including all of its descendant leaf nodes (if any), are removed and re-inserted into BE-Tree.

6. BE-TREE IMPLEMENTATION

Matching Pseudocode Event matching consists of two routines: (1) MatchBETree (Alg. 1) which checks subscriptions in a leaf node and traverses through BE-Tree's p-directory and (2) SearchCDir (Alg. 2) which traverses through BE-Tree's c-directory.

Algorithm 2 `SearchCDir`($event$, $cdir$, $matchedSub$)

1: `MatchBETree`($event$, $cdir.cnode$, $matchedSub$)
2: **if** `IsEnclosed`($event$, $cdir.lChild$) **then**
3: `SearchCDir`($event$, $cdir.lChild$, $matchedSub$)
4: **if** `IsEnclosed`($event$, $cdir.rChild$) **then**
5: `SearchCDir`($event$, $cdir.rChild$, $matchedSub$)

Algorithm 3 `InsertBETree`(sub, $cnode$, $cdir$)

1: {Find $attr$ with max score not yet used for partitioning}
2: **if** $cnode.pdir \neq$ `NULL` **then**
3: **for** $i = 1$ **to** `NumOfPred`(sub) **do**
4: **if** !`IsUsed`($sub.pred[i]$) **then**
5: $attr \leftarrow sub.pred[i].attr$
6: $pnode \leftarrow$ `SearchPDir`($attr$, $cnode.pdir$)
7: **if** $pnode \neq$ `NULL` **then**
8: $foundPartition =$ **true**;
9: **if** $maxScore < pnode.score$ **then**
10: $maxPnode \leftarrow pnode$
11: $maxScore \leftarrow pnode.score$
{if no partitioning found then insert into the l-node}
12: **if** !$foundPartition$ **then**
13: `Insert`(sub, $cnode.lnode$)
{if c-node is the root then partition; otherwise cluster}
14: **if** `isRoot`($cnode$) **then**
15: `SpacePartitioning`($cnode$)
16: **else**
17: `SpaceClustering`($cdir$)
18: **else**
19: $maxCdir \leftarrow$ `InsertCDir`(sub, $maxPnode.cdir$)
20: `InsertBETree`(sub, $maxCdir.cnode$, $maxCdir$)
21: `UpdatePartitionScore`($maxPnode$)

Algorithm 4 `InsertCDir`(sub, $cdir$)

1: **if** `IsLeaf`($cdir$) **then**
2: **return** $cdir$
3: **else**
4: **if** `IsEnclosed`(sub, $cdir.lChild$) **then**
5: **return** `InsertCDir`(sub, $cdir.lChild$)
6: **else if** `IsEnclosed`(sub, $cdir.rChild$) **then**
7: **return** `InsertCDir`(sub, $cdir.rChild$)
8: **return** $cdir$

MatchBETree algorithm takes as inputs: an event, a c-node (BE-Tree's root initially), and a list to store the matched subscriptions. The algorithm, first, checks all subscriptions in the c-node's leaf to find the matching subscriptions (Line 1). Second, for every $\texttt{attr}_i$ in the event's predicates, it searches the c-node's p-directory (Line 4). Lastly, the algorithm calls SearchCDir on all relevant p-nodes. (Line 6).

SearchCDir takes as inputs: an event, c-directory, and a list to store the matched subscriptions. The algorithm is as follows: it calls MatchBETree on the c-node of the current c-directory (Line 1), and it recursively calls SearchCDir on the bucket's left child if the left child encloses the event (Line 3), and on the bucket's right child if the right child encloses the event (Line 5).

Insertion Pseudocode Unlike matching, insertion is rather involved because it also manages the overall dynamics of BE-Tree, i.e., the space partitioning and the space clustering. To insert, BE-Tree's root and a subscription is passed to InsertBETree (Alg. 3) that attempts to find the most effective l-node that encloses the subscription. Basically, the insertion is done recursively in two stages. Initially, the p-directory is searched for every unused $\texttt{attr}_i$ in the subscription, and the $\texttt{attr}_{max}$ with highest p-node score is selected (Lines 2-11); an unused $\texttt{attr}_i$ is one that has not been selected at a higher level of BE-Tree by InsertBETree. Alternatively,

Algorithm 5 SpacePartitioning($cnode$)

```
1: lnode ← cnode.lnode
2: while IsOverflowed(lnode) do
3:     attr ← GetNextHighestScoreUnusedAttr(lnode)
       {Create new partition for the next highest score attr}
4:     pnode ← CreatePDir(attr, cnode.pdir)
       {Move all the subscriptions with predicate on attr}
5:     for sub ∈ lnode do
6:         if sub has attr then
7:             cdir ← InsertCDir(sub, pnode.cdir)
8:             Move(sub, lnode, cdir.cnode.lnode)
9:     SpaceClustering(pnode.cdir)
10: UpdateClusterCapacity(lnode)
```

Algorithm 6 SpaceClustering($cdir$)

```
1: lnode ← cdir.cnode.lnode
2: if !isOverflowed(lnode) then
3:     return
4: if !IsLeaf(cdir) or IsAtomic(cdir) then
5:     SpacePartitioning(cdir.cnode)
6: else
7:     cdir.lChild ← [cdir.startBound, cdir.endBound/2]
8:     cdir.rChild ← [cdir.endBound/2, cdir.endBound]
9:     for sub ∈ lnode do
10:        if IsEnclosed(sub, cdir.lChild) then
11:            Move(sub, lnode, cdir.lChild.cnode.lnode)
12:        else if IsEnclosed(sub, cdir.rChild) then
13:            Move(sub, lnode, cdir.rChild.cnode.lnode)
14:    SpacePartitioning(cdir.cnode)
15:    SpaceClustering(cdir.lChild)
16:    SpaceClustering(cdir.rChild)
17: UpdateClusterCapacity(lnode)
```

if no such $\texttt{attr}_{max}$ is found, then the subscription is inserted into the l-node of the current c-node (Line 13). However, if an $\texttt{attr}_{max}$ is found, then the subscription is pushed down to its corresponding p-node (Line 19).

On the one hand, when $\texttt{attr}_{max}$ is found (Line 19), the c-directory of the corresponding p-node is searched for the smallest possible c-node that encloses the subscription, the search is done through InsertCDir (Alg. 4). Upon choosing the smallest c-node, the subscription advances to the next level of BE-Tree, and the routine InsertBETree is recursively called on the new c-node. On the other hand, when no $\texttt{attr}_{max}$ is found (Line 13), the l-node at the current level is declared as the best l-node to hold the new subscription so that InsertBETree's recursion reaches the base case and terminates; however, after the insertion into the l-node, the node may overflow, which, in turn, triggers the BE-Tree's two-phase space-cutting technique: partitioning and clustering.

In particular, if the chosen l-node is at the root level, in which no partitioning or clustering has yet taken place, then the space partitioning is invoked first (Line 15) because the space clustering is feasible only after the space is partitioned; otherwise, the space clustering is invoked (Line 17).

SpacePartitioning (Alg. 5) proceeds as follows. It uses a scoring function (e.g., selectivity or popularity) to find an unused attribute with the highest ranking, $\texttt{attr}_{max}$, that appears in predicates of the overflowing subscription set (Line 3); consequently, a new p-node is created for the $\texttt{attr}_{max}$. Next, the algorithm iterates over all the subscriptions in the overflowing l-node, and moves all subscriptions having a predicate defined over $\texttt{attr}_{max}$ into the c-directory of the $\texttt{attr}_{max}$'s p-node (Lines 5-8). Lastly, the space clustering is called on the c-directory to resolve any potential overflows resulting from moving subscriptions (Line 9); the entire process is repeated until the l-node is no longer overflowing.

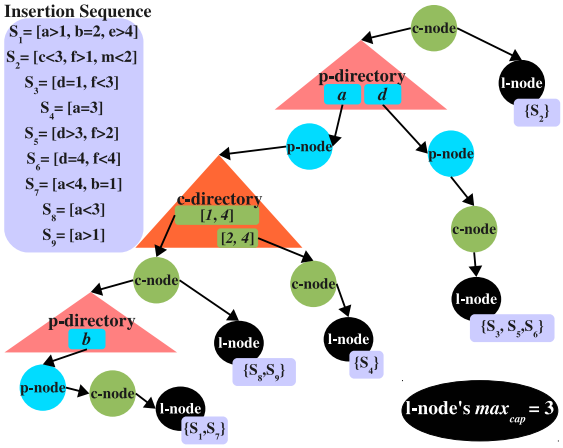


Figure 3: An Insertion Example

SpaceClustering (Alg. 6) is always invoked after the space is partitioned in order to resolve any overflowing l-node by recursively cutting the space in half, and only if the space clustering is unfeasible, then it switches back to the space partitioning. The space clustering is unfeasible when an overflowing l-node is associated to a c-directory bucket that is either a non-leaf bucket, in which further splitting does not reduce the l-node size, or an atomic bucket, in which further splitting is not possible (Line 4). If the space clustering is feasible, then the algorithm splits the current bucket directory in half and moves subscriptions accordingly (Lines 7-13). Next the algorithm recursively calls SpacePartitioning on the current bucket (Line 14) and calls SpaceClustering on the current bucket's left and right child (Lines 15-16).

Next we present an example to further ~~elucidate~~ the insertion algorithm. The final BE-Tree is shown in Fig 3.

说明，阐明

Example *Initially,* BE-Tree *is empty. After inserting* S_1-S_4*, the root's* l*-node overflows, and based on* InsertBETree *(Line 15), the root is partitioned, and the attribute* $[a]$ *is selected as* $attr_{max}$*, and* S_1 *and* S_4 *are pushed down to the next level of* BE-Tree*. After inserting* S_5-S_6*, another overflow occurs at the root, which results in selecting* $[d]$ *as* $attr_{max}$*, and, similarly,* S_3*,* S_5*, and* S_6 *are pushed down the tree. After inserting* S_7-S_8*, both having predicates defined on* $[a]$*, they are directed toward the* l*-node containing* S_1 *and* S_4 *along the root's* p*-node with value* $[a]$*. Consequently, this node will overflow, and based on* InsertBETree *(Line 17), the* l*-node is clustered. Finally, the insertion of* S_9 *will overflow the top-level (non-leaf)* c*-directory, reachable through the root's* p*-node with value* $[a]$*; thereby, triggering the space partitioning and selecting* $[b]$ *as the next* $attr_{max}$ *through* SpaceClustering *(Line 5).*

7. EVALUATIONS

We present a comprehensive evaluation of BE-Tree using both synthetic and real datasets. The experiments were ran on an Intel 2.66 MHz Quad-core machine with 4GB of memory running Ubuntu 10.04. All algorithms are implemented in C and compiled using gcc 4.4.3 with optimizations set to $O3$.

We compare BE-Tree with several popular matching algorithms over a variety of controlled experimental conditions: *workload distribution*, *workload size*, *space dimensionality*, *average subscription and event size*, *dimension cardinality*, *predicate selectivity*, *dimension selectivity*, *subscription expressiveness*, and *event matching probability*.

We also ran experiments to determine optimal choices for BE-Tree internal parameters, cf. Sec. 7.3.1. The values used throughout our experiments are: $\texttt{max}_{cap}$ ranging from 5-160, $\texttt{min}_{size} = 3$,

Table 2:

(a) Levels of Expressiveness

Op Class	Operators
Min	$(=)$
Low	$(=, \in)$
Med	$(<, \leq, =, \geq, >, \in,$ BETWEEN)
High	$(<, \leq, =, \neq, \geq, >, \in, \notin,$ BETWEEN)

(b) $P_j^{(i,=,v_*)}$ Inferred Pred

$i \neq v_1$ where $v_1 \neq v_*$
$i < v_1$ where $v_1 \geq v_*$
$i > v_1$ where $v_1 \leq v_*$
$i \in \{v_1, \cdots, v_k\} \cup \{v_*\}$
$i \notin \{v_1, \cdots, v_k\} - \{v_*\}$
i BETWEEN v_1, v_2 where $v_1 \leq v_*$ and $v_2 \geq v_*$

$\texttt{rank}_{window}$ for each node is 10% of expressions in the node's subtree, and $\theta = 0$.

7.1 Datasets

Synthetic Dataset One of the key challenges in generating synthetic datasets with high-dimensions is the inability to control the matching probability. With no control mechanism, the matching probability would be virtually zero. To address this concern, we developed a novel workload generation framework, *BEGen*[2]. The workloads are generated in two steps: (1) a set of base expressions with only equality predicates are generated, in which a predicate's attribute is chosen based on either a uniform or a Zipf distribution; (2) for each base expression e_B, we generate a set of derived expressions, e_i, such that

$$\forall P_q(x) \in e_i,\ \exists P_o(x) \in e_B,\ P_q^{\texttt{attr}} = P_o^{\texttt{attr}}, \forall x\ P_o(x) \rightarrow P_q(x).$$

Moreover, each predicate in a base expression is kept with probability $\Pr_{pred}$ in its derived expressions, and each predicate in derived expressions is transformed with probability $\Pr_{trans}$ using one of the inferred predicate rules given in Table 2(b). In our synthetic experiments, base expressions model events, and derived expressions model subscriptions. In addition, the probability $\Pr_{pred}$ and $\Pr_{trans}$ are chosen such that with a high probability, we avoid generating any duplicate subscriptions. Thus, if the average number of predicates per event is x and the average number of predicates per subscriptions is y, we choose x and y such that $\binom{x}{y}$ is large enough such that duplicate subscriptions are unlikely. For example, by tuning the number of generated base expressions, we can control the matching probability for a given subscription workload. For instance, to generate a workload size of $1,000$ with matching probability 1%, we generate 100 base expressions and 10 derived expressions for each base expression.

In our evaluation, we assign up to 6 values for $(\in, \notin)$, and on average, we use a predicate range size of 12% of the domain size for the BETWEEN operator, and we randomly pick a value for the remaining operators. The value of each parameter in our synthetic workload is summarized in Table 3 (columns 1-8), in which each column corresponds to a different workload profile while each row corresponds to the actual value of the workload parameters. Lastly, Table 2(a) captures our four levels of operator expressiveness.

Real Dataset In the absence of a standard benchmark for evaluating matching algorithms with real data, as part of our *BEGen* framework, we propose a novel generation of real workloads from public domain data. We focus on data extracted from the DBLP repository. In particular, we use the proceeding titles and author names as two sources of data extracted from DBLP. We, first, employ a de-duplication technique to eliminate duplicate entries and to convert the data into a set of q-grams. This conversion is based on tokenization of a string into a set of q-grams (sequence of q consecutive characters). For example, a 3-gram tokenization of "string" is given by {'str', 'tri', 'rin','ing'}. Second, we use a novel transformation to convert each string from a collection of q-grams into a Boolean expression. Therefore, we model the collection of q-grams {'str', 'tri', 'rin', 'ing'} by a set of equality predicates as follows: ['*st*' = 'r', '*tr*' = 'i','*ri*'='n', and '*in*'='g']. Our 3-grams-based transformation results in Boolean expressions in a space of 677 dimensions. For the real datasets, we also carry out experiments in which we control the degree of matching probability using the profile generation technique that was used for the synthetic datasets. Table 3 (last 4 columns) summarizes various workloads generated using the real data sets.

[2]http://msrg.org/datasets/BEGen

7.2 Matching Algorithms

The algorithms in our comparison studies are (1) SCAN (a sequential scan of the subscriptions), (2) SIFT (the counting algorithm [27] enhanced with the enumeration technique of [26] to support range operators), (3) k-ind (the conjunction algorithm implemented over k-index [26]), (4) GR (the Gryphon algorithm [2]), (5) P (the Propagation algorithm [13], in which subscriptions are clustered based on their most selective attribute), (6a) BE (our fully dynamic version of BE-Tree in which the index is constructed by individually inserting each subscription, and (6b) BE-B (our batching version of BE-Tree in which all subscriptions are known in advance resulting in a better initial statistics to guide the space partitioning at the root level). Unlike the construction of our dynamic BE-Tree, we have constructed k-index, Gryphon, and Propagation using a static workload in which all subscriptions are known in advance. In addition, we have implemented the specialized Gryphon algorithm that supports only equality predicates and the generic Gryphon algorithm that supports arbitrary predicates [2] including all optimizations proposed in [2].

In our experiments, we distinguish between four levels of predicate expressiveness because not all algorithms can naturally support our expressive set of operators. In particular, SIFT [27] and k-index [26] naturally support only a weak semantics for operators $\notin$ and $\neq$ in which subscription predicates with inequality on $attr_i$ are also matched by an input event that does not define any predicate on $attr_i$. The common alternative semantic is to consider a subscription as matched only if an event provides a value for all subscription predicates with inequality (strong semantics). To support inequality operators with the strong semantics, a default value must be added to both subscriptions' inequality predicate and the unspecified attributes in the event [26]. This scheme results in an unacceptable performance as space dimensionality increases for the strong semantics. Thus, we do not consider SIFT and k-index for the inequality (strong semantics) experiments. Similarly, Propagation does not support inequality predicates as access predicates and relies on a post processing step to resolve inequalities. Thus, we do not consider Propagation in the inequality experiments either.

7.3 Experiment Results

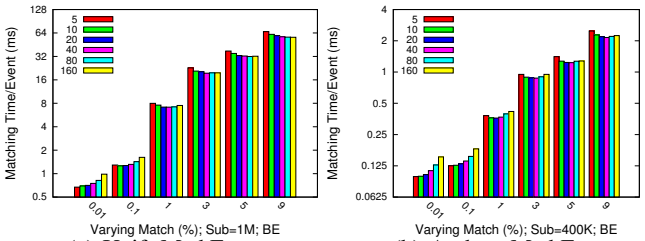
In our micro experiments, we study the BE-Tree parameters and their relation to the overall performance of BE-Tree. Most importantly, we establish that the maximum l-node capacity is the main parameter of BE-Tree and provide a systematic guideline on how to adjust it. We then shift gears to focus on macro experiments in which an extensive evaluation and comparison of BE-Tree with other related approaches are conducted. Finally, we illustrate the effectiveness of BE-Tree's self-adjustment under changing workloads.

7.3.1 Micro Experiments

Primary Parameter The most important parameter of BE-Tree is the maximum l-node capacity size, $\texttt{max}_{cap}$, which triggers our two-phase space-cutting technique. In Fig. 4, for various workloads with different matching probability, the effect of varying the

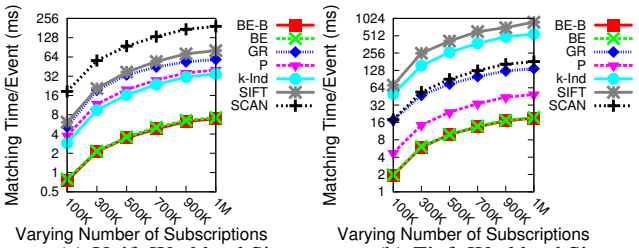
Table 3: Synthetic and Real Workload Properties

	Workload Size	Number of Dimensions	Dimension Cardinality	Predicate Selectivity	Dimension Selectivity	Sub/Event Size	% Equality Pred	Match Prob		DBLP (Author)	DBLP (Title)	Match Prob (Author)	Match Prob (Title)
Size	100K-1M	1M	100K	100K	100K	100K	1M	1M		100-760K	50-250K	400k	150
Number of Dim	400	50-1400	400	400	400	400	400	400		677	677	677	677
Cardinality	48	48	48-150K	48	2-10	48	48	48		26	26	26	26
Avg. Sub Size	7	7	7	7	7	5-66	7	7		8	35	8	30
Avg. Event Size	15	15	15	15	15	13-81	15	15		8	35	16	43
Pred Avg. Range Size %	12	12	12	6-50	—	12	12	12		—	—	12	12
% Equality Pred	0.3	0.3	0.3	0.3	1.0	0.3	0.2-1.0	0.3		1.0	1.0	0.3	0.3
Op Class	Med	Med	Med	Med	Min	Med	Med	Lo-Hi		Min	Min	Lo-Hi	Lo-Hi
Match Prob %	1	1	1	1	—	1	1	0.01-9		—	—	0.01-9	0.01-9



(a) Unif: Med Exp (b) Author: Med Exp

Figure 4: Varying the l-node Capacities



(a) Unif: Workload Size (b) Zipf: Workload Size

Figure 5: Varying Workload Size

l-node capacity is shown. Although, there is a correlation between the optimal value of maximum l-node capacity and the degree of the matching probability, the effect is not significant. Thus, we conclude that BE-Tree is not highly sensitive to the maximum l-node capacity parameter. The results of varying l-node capacities are summarized in Table 4, which we use as a guiding principle throughout our evaluation for choosing the optimal value with respect to the degree of matching probability. Another important factor is that the l-node capacity is a tunable parameter which can be dynamically adjusted (increased or decreased) based on the matching feedback to tune future executions of the two-phase space-cutting technique.

Table 4: Guiding principles for adjusting $\texttt{max}_{cap}$

	$\texttt{max}_{cap}$
Match Prob $<$ 1%	5
1% $\leq$ Match Prob $<$ 10%	20
Match Prob $\geq$ 10%	160

BE-Tree Construction The construction time of the dynamic BE-Tree, for the largest workload of up to one million subscriptions, was under 5 seconds in our experiments. For the main datasets, the BE-Tree's average construction time and index size are summarized in Table 5.

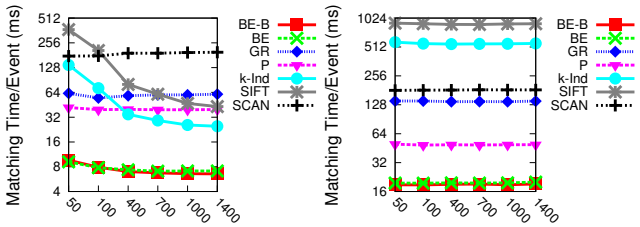
Table 5: BE-Tree Construction Time & Size

Data Sets	Construction Time (second)	Index Size (MB)
Unif (1M)	4.75	84
Zipf (1M)	4.27	82
Author (760K)	4.46	74
Title (250K)	2.48	37

7.3.2 *Macro Experiments*

Effect of Workload Distribution The major distinguishing factor among matching algorithms is the workload distribution, which clearly sets apart key vs. non-key based methods. Both k-index and SIFT (non-key-based methods) are highly sensitive to the distribution of the workload whereas BE-Tree and Propagation (key-based methods) are robust with respect to the distribution. The effects of the distribution are shown in Fig. 5-11 in which the graphs on the left column correspond to a uniform distribution (for choosing predicates' attributes) while the graphs on the right column correspond to a Zipf distribution. The general trend is that under uniform distribution BE-Tree, k-index, Propagation, Gryphon, SIFT all outperform SCAN that benefits only from sequential memory access. However, under Zipf distribution both k-index and SIFT are outperformed by SCAN in some cases. The poor matching time is attributed to few popular attributes that are common among all subscriptions. Therefore, for every event about 80-90% of subscriptions have at least one satisfied predicate which translates into a large number of random memory accesses to increment subscription counters in SIFT and to sort and scan through k-index hashtable buckets (refereed to as the posting list in [26]). Overall, the BE-Tree matching time is at least four times better than the next best algorithm (k-index) for uniform distribution and at least two and half times better (Propagation) for Zipf distribution.

Effect of Workload Size Next, we consider the matching time as we increase the number of subscriptions processed. Fig. 5a,b illustrate the effect on matching time as the number of subscriptions increases in which all algorithms scale linearly with respect to the number of matched subscriptions. In these experiments, BE-Tree exhibits an 80% better matching time as compared to the next best algorithm for the uniform workload and a 63% better matching time for the Zipf workload.



(a) Unif: Dimensionality (b) Zipf: Dimensionality

Figure 6: Varying Space Dimensionality

Effect of Dimensionality Unlike the workload size, the effect of space dimensionality is more subtle; all algorithms with exception of k-index and SIFT are unaffected as the dimensionality varies, Fig. 6a,b. Both k-index and SIFT substantially suffer in lower di-

mensionality for the uniform workload in which subscriptions tend to share many common predicates, which results in high overlap among subscriptions. Therefore, k-index and SIFT are sensitive to degree of overlap among subscriptions and achieve peak matching time when subscriptions are distributed into a set of disjoint subspaces. For instance, when dimension is set to $d = 50$, BE-Tree improves over k-index by 93% and for $d = 1400$, BE-Tree improves over k-index by 75% in which k-index is the second best algorithm for such high dimensionality. However, for the Zipf workload, Fig. 6b, k-index and SIFT matching time does not improve as the dimensionality increases because of the existence of few popular dimensions, resulting in a large overlap among subscriptions. Thus, BE-Tree improves over k-index by 97%.

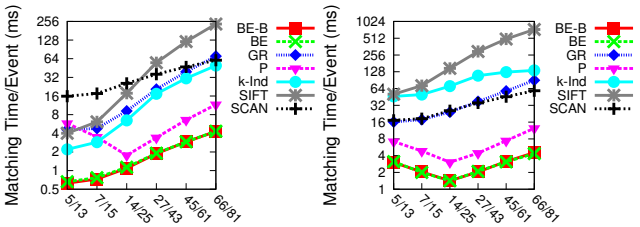


Figure 7: Varying Subscription/Event Size

Effect of Subscription/Event Size Another key workload characteristic is the average number of predicates per subscription and event, Fig. 7a,b. In particular, both k-index and SIFT are highly sensitive to the number of predicates: in addition to increasing the overlap among subscriptions, for k-index it also translates into a longer sorting time and for SIFT it translates into a larger number of retrieving and scanning hashtable buckets. The Propagation algorithm starts with a lower matching time because subscriptions have fewer predicates and the chances of finding an equality access predicate with high selectivity is lower, as subscriptions are not evenly distributed in space. As a result, the Propagation algorithm reaches its optimal performance when average subscription size reaches 14, and no noticeable benefit is gained as the subscription size further increases, instead the response time gradually decreases due to an increase in computation cost for checking each predicate. In general, BE-Tree gracefully scales as the number of predicates increases because of its multi-layer structure and improves over the next best algorithm by 63% for the uniform and by 65% for the Zipf workload.

Effect of Dimension Cardinality The importance of increasing the dimension cardinality is twofold: the matching rate and the memory requirement. The matching rate of most algorithms scales gracefully as the dimension cardinality increases, e.g., BE-Tree and Propagation (Fig. 8). In short, BE-Tree improves over Propagation's matching time by 66% for the uniform workload and improves over Propagation's matching time by 59% for the Zipf workload, Fig. 8a,b, respectively.

However, unlike in BE-Tree, the memory footprint of k-index and SIFT blows up exponentially as we increase the dimension cardinality, while keeping constant the ratio of predicate range size with respect to cardinality. Both approaches rely on the enumeration technique to resolve range predicates. For example, in order to cope with the operator BETWEEN $[v_1, v_2]$, the enumeration essentially transforms the value of $v_2 - v_1$ from a decimal to a unary representation—an exponential transformation. Therefore, we were unable to run k-index and SIFT on workloads with cardinality of 6K and beyond. For instance, the workload with a 6K cardinality has on average a predicate range size of 150 which in turn replaces a single range predicate with 150 equality predicates.

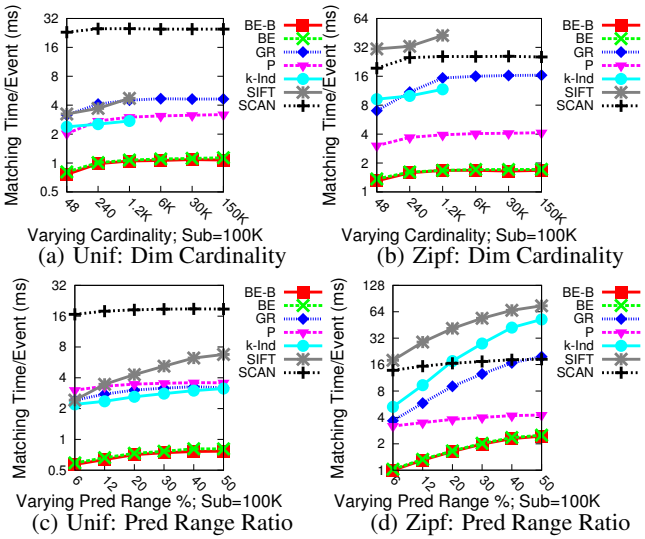


Figure 8: Varying Cardinality & Pred Range/Cardinality Ratio

To further analyze the role of predicate range sizes, we devise another experiment that varies the predicate range size while fixing the cardinality.

Effect of Predicate Selectivity In fact, the ratio of predicate range size with respect to the dimension cardinality is inversely proportional to the predicate selectivity. The predicate selectivity has a small influence on Propagation, which relies solely on selective equality predicates, while it has a huge influence on k-index and SIFT, which do not utilize the predicate selectivity information. Therefore, as shown in Fig. 8c,d, as the ratio of the predicate range size increases (selectivity decreases), the search space pruning mechanism of k-index and SIFT suffer due to the increased number of false candidates.

compensates

In general, a low selective predicate causes a less effective pruning of the search space, and BE-Tree compensate for the low selectivity with a deeper tree structure that provides a greater opportunity to prune the search space by using both highly selective predicates (equality) and low selective predicates (range operators). As a result, BE-Tree improves over the next best algorithms by 76% and 43% for the uniform and the Zipf workloads, respectively (cf. Fig. 8c,d.)

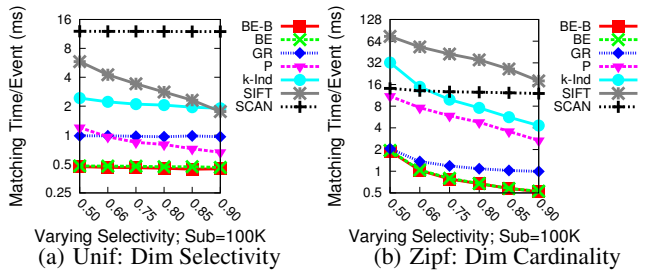


Figure 9: Varying Dimension Selectivity

Effect of Dimension Selectivity A notable workload characteristic is dimension selectivity of the space, which could have a direct influence on the ability of the matching algorithm to effectively prune the search space. The result of our dimension selectivity is captured in Fig. 9. For a uniform workload the robustness of BE-Tree and Gryphon is evident because as the dimension selectivity varies only a negligible increase in matching time of at most 1% is observed while for Propagation and k-index a significant increase in matching time of upto 80% and 27% is observed, respectively. The importance of dimension selectivity is further magnified for a

Zipf workload (cf. Fig. 9b) in which a few dimensions are dominant. Therefore, for low selectivity a substantial overhead is incurred due to a larger number of false candidates. For instance, as we decrease selectivity from 0.9 to 0.5, the response time is increased by 411% and 650% for Propagation and k-index, respectively. Therefore, low selectivity results in a less effective pruning of the search space, and BE-Tree prunes the search space more effectively by employing a multi-layer tree structure.

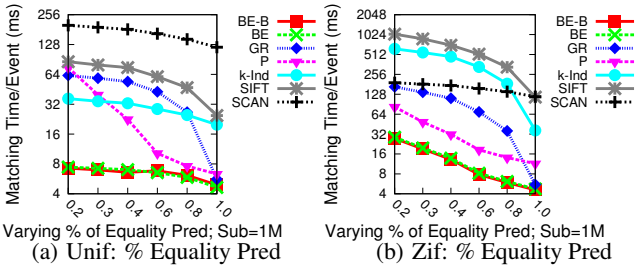


Figure 10: Varying % of Equality Predicates

Effect of Percentage of Equality Predicates In this experiment, we study the effects of ratio of equality vs. non-equality predicates for each subscription. The general trend is that the matching times for all algorithms improve as the percentage of subscription equality predicates increases because the overlap among subscriptions is reduced, Fig. 10a,b. Most notably, when subscriptions consist only of equality predicates, the specialized Gryphon (for equality predicates) results in a substantial performance gain, being the best algorithm after BE-Tree, compared with the generic Gryphon. The Propagation algorithm also improves significantly when subscriptions are restricted to only equality predicates because there is a better chance to find more effective access predicates. However, among all algorithms, Gryphon and Propagation, are the most highly sensitive to the percentage of equality predicates. As the percentage of equality predicate decreases, their performance substantially deteriorates. For instance, for the uniform workload, Fig. 10a, BE-Tree improves over Gryphon matching time by 88% when subscriptions only have few equality predicates and by 20% when subscriptions only have equality predicates.

Effect of Percentage of Matching Probability As the matching probability increases, the number of candidate subscriptions and the event matching time also increases. Therefore, we studied the effects of varying matching probability under both uniform and Zipf workload based on different levels of predicate expressiveness. Under a uniform workload with low and medium expressiveness, while keeping the matching probability below 1%, k-index outperforms both Propagation and SIFT, while BE-Tree improves over k-index by 97%. However, as the matching probability goes beyond 3%, the Propagation algorithm begins to outperform both k-index and SIFT, while BE-Tree improves over Propagation by up to 33%, even as the matching probability reaches 9%, Fig. 11a,c. As the matching probability goes beyond 35%, the success of BE-Tree continues as BE-Tree becomes marginally the better algorithm followed by Propagation and SCAN (omitted in the interest of space [24].) In general, an increase in matching probability results in an increase in the number of candidate matches. Therefore, SIFT is forced to scan large hashtable buckets, using random access, to increment subscription counters for each of the satisfied predicates. Similarly, k-index is forced to scan large buckets (i.e., posting lists) with a reduced chance of pruning and an increased application of sorting to advance through each bucket. For the Zipf distribution, Propagation remains the next best algorithm after BE-Tree, Fig. 11b,d. Furthermore, in the experiments where the highest

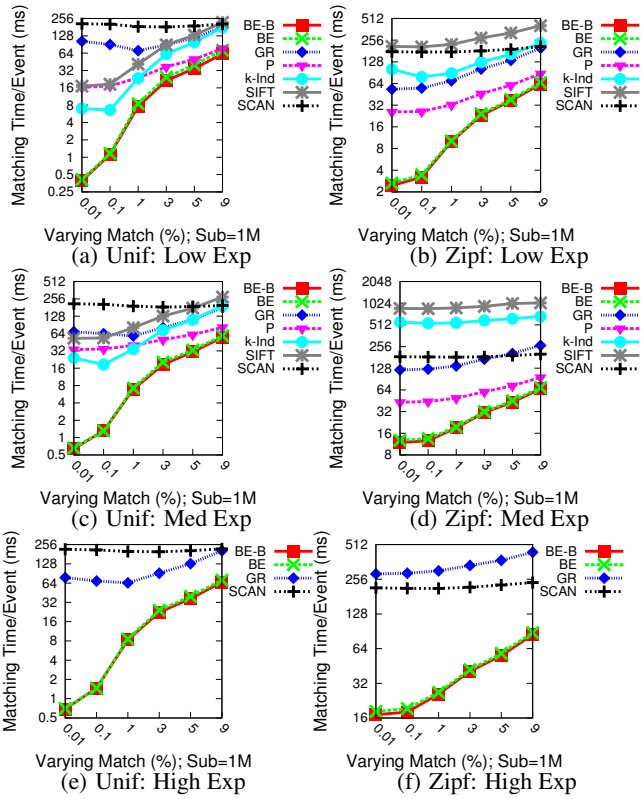


Figure 11: Varying Matching Probability

level of expressiveness was used, BE-Tree dominates both Gryphon and SCAN by orders of magnitude, Fig. 11e,f.

Real Datasets In our synthetic experiments in which the percentage of equality predicates was varied, BE-Tree and Gryphon were the top performing algorithms followed by Propagation and k-index. Similar trends are also observed for real datasets. In particular, for the author dataset, Fig. 12a, with an average of 8 predicates per subscriptions, BE-Tree improves over Gryphon by 37% while substantially improving over Propagation by over 98%. For the title dataset with much larger number of predicates per subscriptions, that is at around 35 predicates per subscriptions, the gap between BE-Tree and the other algorithms further widens. This is due to the novel scoring that exploits interrelationships within dimensions and the multi-layer structure of BE-Tree that effectively utilizes most of the subscription predicates to reduce the search space. Therefore, as demonstrated in Fig. 12b, BE-Tree improves over Gryphon by 51% and significantly improves over Propagation by more than 99%, up to three orders of magnitude. Moreover, we have conducted the matching probability experiments with varying degree of expressiveness which produced similar results as for the synthetic dataset, not included due to lack of space (cf. available in a technical report [24]).

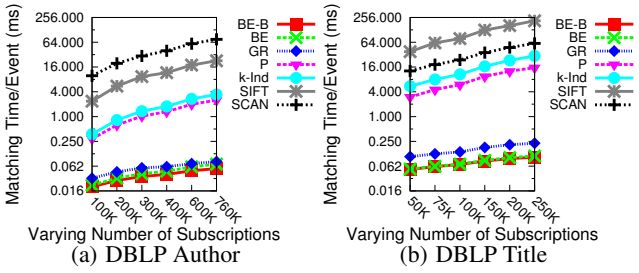


Figure 12: Varying the workload size

Adapting to Subscription Changes In our last experiment, we

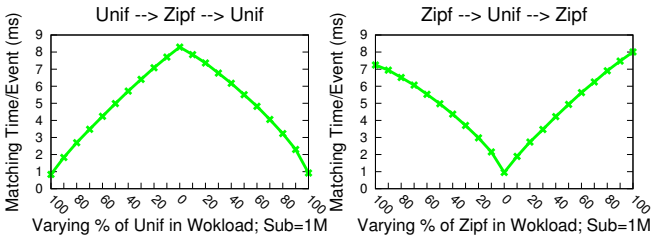


Figure 13: Dynamic Workload

studied the self-adjusting mechanism of BE-Tree. The experiment setup is as follow. First, we fixed the event workload in which half of the events are generated using a uniform while the other half using a Zipf distribution. Second, we generated uniform and Zipf subscription workloads with 0.1% matching probability. In each experiment, we start by individually inserting each subscription from workload X into BE-Tree, then we individually remove each subscription from BE-Tree and individually insert subscriptions from the workload Y until BE-Tree contains only subscriptions from workload Y, then we reverse this process until we gradually switch back to the original workload X. After a fixed number of deletions and insertions, we run our event workload, and record the matching time. The objective is to illustrate that BE-Tree adapts to workload changes and the performance of BE-Tree does not significantly deteriorate even in extreme situations in which the distribution rapidly and dramatically changes. In the first experiment, while transitioning from uniform to Zipf and back again, the matching time at the end of the transition approaches the original performance, Fig. 13a. A similar adaptation was observed when we transitioned from Zipf to uniform and back again, Fig. 13b.

8. CONCLUSIONS

In this work, we presented BE-Tree, a novel index structure to efficiently index and match **B**oolean **E**xpressions defined over a high-dimensional discrete space. We introduced a novel two-phase space-cutting technique to cope with the curse of dimensionality underlying the subscription and event space, which underlies many application domains, and we developed a cost model and self-adjustment policies that enable BE-Tree to actively adapt to workload changes. Moreover, we demonstrated that BE-Tree is a generic index for Boolean expressions that supports predicates with expressive operators.

To summarize our evaluation, let us consider three main workload categories: (1) workloads with uniform distribution and a low-to-high degree of expressiveness, (2) workloads with Zipf distribution and a low-to-high degree of expressiveness, and (3) real-world and synthetic workloads with minimum degree of expressiveness (equality predicates only). From best to worst performing algorithms, in the first category (uniform) we have: BE-Tree, k-index [26], Propagation [13], and SIFT [27]. In the second category (Zipf) we have: BE-Tree, Propagation, k-index, and SIFT. Lastly, in the third category we have: BE-Tree, Gryphon [2], Propagation, and k-index. The general trends are that non-key based algorithms, i.e., k-index and SIFT, do poorly on workloads that consist of few popular dimensions (i.e., low dimensional space and Zipf distribution) because of the significant increase in the number of false candidates that have to be considered. Also, Gryphon and Propagation are highly sensitive to the degree of expressiveness of subscriptions. Finally, BE-Tree dominated in every category.

9. REFERENCES

[1] R. Agrawal, A. Ailamaki, P. A. Bernstein, et al. The Claremont report on database research. *SIGMOD Rec.'08*.

[2] M. K. Aguilera, R. E. Strom, D. C. Sturman, M. Astley, and T. D. Chandra. Matching events in a content-based subscription system. In *PODC'99*.

[3] S. Berchtold, D. A. Keim, and H.-P. Kriegel. The X-tree: An index structure for high-dimensional data. In *VLDB'96*.

[4] K. S. Beyer, J. Goldstein, R. Ramakrishnan, and U. Shaft. When is "nearest neighbor" meaningful? In *ICDT'99*.

[5] L. Brenna, A. Demers, J. Gehrke, M. Hong, J. Ossher, B. Panda, M. Riedewald, M. Thatte, and W. White. Cayuga: high-performance event processing engine. *SIGMOD'07*.

[6] A. Campailla, S. Chaki, E. Clarke, S. Jha, and H. Veith. Efficient filtering in publish-subscribe systems using binary decision diagrams. In *ICSE'01*.

[7] S. Ceri, R. Cochrane, and J. Widom. Practical applications of triggers and constraints: Success and lingering issues (10-year award). In *VLDB'00*.

[8] A. Chandel, O. Hassanzadeh, N. Koudas, M. Sadoghi, and D. Srivastava. Benchmarking declarative approximate selection predicates. In *SIGMOD'07*.

[9] B. Chandramouli, J. Goldstein, and D. Maier. High-performance dynamic pattern matching over disordered streams. *PVLDB'10*.

[10] U. Dayal, E. N. Hanson, and J. Widom. Active database systems. In *Modern Database Systems'95*.

[11] Y. Diao, M. Altinel, M. J. Franklin, H. Zhang, and P. Fischer. Path sharing and predicate evaluation for high-performance XML filtering. *ACM TODS'03*.

[12] N. Dindar, B. Güç, P. Lau, A. Ozal, M. Soner, and N. Tatbul. Dejavu: declarative pattern matching over live and archived streams of events. In *SIGMOD'09*.

[13] F. Fabret, H.-A. Jacobsen, F. Llirbat, J. Pereira, K. A. Ross, and D. Shasha. Filtering algorithms and implementation for fast pub/sub systems. *SIGMOD'01*.

[14] M. Fontoura, S. Sadanandan, J. Shanmugasundaram, S. Vassilvitski, E. Vee, S. Venkatesan, and J. Zien. Efficiently evaluating complex Boolean expressions. In *SIGMOD'10*.

[15] C. L. Forgy. Rete: A fast algorithm for the many pattern/ many object pattern match problem. *AI'82*.

[16] M. Freeston. A general solution of n-dimensional B-tree problem. *SIGMOD'95*.

[17] A. Guttman. R-tree dynamic index structure for spatial searching. *SIGMOD'84*.

[18] E. N. Hanson, C. Carnes, L. Huang, M. Konyala, L. Noronha, S. Parthasarathy, J. Park, and A. Vernon. Scalable trigger processing. In *ICDE'99*.

[19] E. N. Hanson, M. Chaabouni, C.-H. Kim, and Y.-W. Wang. A predicate matching algorithm for database rule systems. In *SIGMOD'90*.

[20] H. Leung and H.-A. Jacobsen. Efficient matching for state-persistent publish/subscribe systems. In *CASCON'03*.

[21] G. Li, S. Hou, and H.-A. Jacobsen. A unified approach to routing, covering and merging in publish/subscribe systems based on modified binary decision diagrams. In *ICDCS'03*.

[22] W. Rjaibi, K. R. Dittrich, and D. Jaepel. Event matching in symmetric subscription systems. In *CASCON'02*.

[23] M. Sadoghi, I. Burcea, and H.-A. Jacobsen. GPX-Matcher: A generic Boolean predicate-based XPath expression matcher. In *EDBT'11*.

[24] M. Sadoghi and H.-A. Jacobsen. Indexing Boolean expression over high-dimensional space. *Technical Report CSRG-608, University of Toronto, 2010*.

[25] M. Sadoghi, H.-A. Jacobsen, M. Labrecque, W. Shum, and H. Singh. Efficient event processing through reconfigurable hardware for algorithmic trading. *PVLDB'10*.

[26] S. Whang, C. Brower, J. Shanmugasundaram, S. Vassilvitskii, E. Vee, R. Yerneni, and H. Garcia-Molina. Indexing Boolean expressions. In *VLDB'09*.

[27] T. Yan and H. Garcia-Molina. Index structures for selective dissemination of information under the Boolean model. *ACM TODS'94*.

引用134 2016-4-27
引用777 2016-4-27
引用51 2016-4-27
引用615 2016-4-27
引用214 2016-4-27