2012 IEEE 28th International Conference on Data Engineering

Relevance Matters: Capitalizing on Less
Top-k Matching in Publish/Subscribe

Mohammad Sadoghi [1], Hans-Arno Jacobsen [2]

Department of Computer Science, University of Toronto
[1]mo@cs.toronto.edu [2]jacobsen@eecg.toronto.edu

***Abstract*—The efficient processing of large collections of Boolean expressions plays a central role in major data intensive applications ranging from user-centric processing and personalization to real-time data analysis. Emerging applications such as computational advertising and selective information dissemination demand determining and presenting to an end-user only the most relevant content that is both user-consumable and suitable for limited screen real estate of target devices. To retrieve the most relevant content, we present BE*-Tree, a novel indexing data structure designed for effective hierarchical top-k pattern matching, which as its by-product also reduces the operational cost of processing millions of patterns.**

To further reduce processing cost, BE*-Tree employs an adaptive and non-rigid space-cutting technique designed to efficiently index Boolean expressions over a high-dimensional continuous space. At the core of BE*-Tree lie two innovative ideas: (1) a bi-directional tree expansion build as a top-down (data and space clustering) and a bottom-up growths (space clustering), which together enable indexing only non-empty continuous sub-spaces, and (2) an overlap-free splitting strategy. Finally, the performance of BE*-Tree is proven through a comprehensive experimental comparison against state-of-the-art index structures for matching Boolean expressions.

I. Introduction

The efficient processing of Boolean expressions is a key functionality required by major data intensive applications ranging from user-centric processing and personalization to real-time data analysis. Among user-centric applications, there are targeted Web advertising, online job sites, and location-based services; common to all are specifications (e.g., advertising campaigns, job profiles, service descriptions) modeled as Boolean expressions (subscriptions) and incoming user information (e.g., user profiles) modeled as events [1], [2], [3]. Among real-time data analysis applications, there are (complex) event processing, intrusion detection, and computational finance; again, common among them are predefined set of patterns (e.g., attack specifications and investment strategies) modeled as Boolean expressions (subscriptions) and streams of incoming data (e.g., stock feeds) modeled as events [4], [5], [6], [2].

Unique to user-centric processing and personalization are strict requirements to determine only the most relevant content that is both user-consumable and suitable for the often limited screen real estate of client devices. Shared among user-centric processing and real-time analysis applications are two requirements: (1) scaling to millions of patterns and specifications (expressions) and (2) matching latency constraints in the sub-second range.

To put into context the importance of finding the most relevant Boolean expression, we extract real scenarios from prominent user-centric applications such as targeted Web advertising (e.g., Google, Microsoft, Yahoo!) and online job sites (e.g., Monster).

In Web advertising, a demographic targeting service specifies constraints such as [*credit-score* > 650 $(wt = 0.4)$, *num-visits* > 10 $(wt = 0.2)$, *age* BETWEEN $\{22, 36\}$ $(wt = 0.4)$] while an incoming user's profile includes information such as [*credit-score* $= 732$ $(wt = 0.2)$, *num-visits* $= 17$ $(wt = 0.2)$, *age* $= 27$ $(wt = 0.6)$]. Thus, only the most relevant ads for the user are displayed; the ad relevance is computed based on a scoring function that takes as inputs the constraints' weights (wt) assigned by user and advertiser. Similarly, in online job sites, an employer submits the job details, [*category* = 'green jobs' $(wt = 0.2)$, *hours/week* > 15 $(wt = 0.5)$, *hourly-rate* < 45 $(wt = 0.3)$], and when a job seeker visits with the profile, [*category* = 'green jobs' $(wt = 0.4)$, *hours/week* $= 20$ $(wt = 0.1)$, and *hourly-rate* $= 30$ $(wt = 0.5)$], she is connected to the most relevant employer [1], [2], [3]; again the relevance is computed based on the weights of the constraints. These scenarios demand the indexing of millions of Boolean expressions while having matching latency requirements in the sub-second range to avoid the deterioration of user-perceived latency when browsing and shopping online.

In these examples, advertisers' and employers' needs and users' profiles are captured using multiple attributes (i.e., dimensions in space) such as *age* and *credit-score*, in which the importance of each attribute is explicitly captured using the attributes' weight. Additionally, each attribute is constrained to a range of values on either continuous or discrete domains using an operator. The resulting quadruple formed by an attribute, an operator, a range of values, and a predicate weight is referred to as a *Boolean predicate*. A conjunction of Boolean predicates, here representing both the advertiser's and employer's needs and the user's profiles, constitutes a *Boolean expression (BE)*.

In short, we extract four requirements of paramount importance from applications that rely on processing Boolean expressions. First and foremost, the index must enable top-k matching to quickly retrieve only the most relevant expressions, which is not addressed by prior Boolean expression matching approaches [4], [5], [7]. This relevance computation must be based on a generic scoring function, which is also not addressed by prior approaches that model relevance identification as part of matching [1]. Second, the index must support predicates with an expressive set of operators over a continuous domain, which, again, is not supported by prior matching approaches [5], [2], [7]. Third, the index must enable

1084-4627/12 $26.00 © 2012 IEEE
DOI 10.1109/ICDE.2012.38

dynamic insertion and deletion of expressions, often disregarded as requirement in matching algorithm designs [4], [2]. Finally, the index must employ a dynamic structure that adapts to changing workload patterns and expression schemata, also often disregarded in matching algorithm designs [4], [2].

We designed BE*-Tree to specifically tackle these challenges. In particular, BE*-Tree is geared towards determining the most relevant patterns, in which top-k processing is treated as first class citizen. BE*-Tree introduces a hierarchical top-k processing scheme that differs from existing work that assumes a static and a flat structure [2]. Additionally, BE*-Tree achieves scalability by overcoming the curse of dimensionality through a non-rigid space-cutting technique while attaining expressiveness and dynamism without restricting the space to only a discrete and finite domain as prevalent in [2], [7]. Notably, with respect to scalability, we solve the two critical properties common to most high-dimensional indexing: (1) avoiding indexing non-empty space (cf. R-tree [8]) (2) minimizing overlap (cf. R*-tree [9]) and coverage [8]. These properties are tackled by proposing a bi-directional tree expansion: a top-down (data and space clustering) and a bottom-up (space clustering) growths process, which together enable indexing only non-empty continuous sub-spaces and adaptation to workload changes; and a splitting strategy to systematically produce and maintain overlap-free subspaces for holding expressions.

Our contributions in this paper are four-fold:

1) We generalize the BE matching problem to subspace matching (Section III),
2) We introduce BE*-Tree tailored to top-k processing for efficiently determining the most relevant matches among large numbers of potential matches of millions of subscriptions defined over thousands of dimensions (Sections IV,VI),
3) We develop a novel self-adjusting mechanism using bi-directional tree expansion that continuously adapts as the workload changes, and propose a novel overlap-free splitting strategy (Section V),
4) Finally, we present a comprehensive evaluation framework that benchmarks state-of-the-art matching algorithms, specifically emphasizing top-k processing: SCAN [10], BE-Tree [7], and k-index [2] (Section VII).

II. Related Work

Generally speaking the problems related to indexing Boolean expressions have been studied in different contexts including database (multi-dimensional indexing [8], [9], [11]) and top-k querying [12], [13]) and publish/subscribe (matching techniques [10], [4], [5], [6], [2], [3], [7]).

In general, the database literature focuses on much lower dimensionality, while we target a dimensionality in order of thousands, which is orders of magnitude larger than capabilities of existing database indexing structures [11]. The database top-k processing [12], [13] differs with our proposed top-k model in an important respect: our top-k model solves the reverse problem. In the database context, top-k querying means finding the most relevant tuples (events) for a given query (subscription). But in our context, top-k matching means finding the most relevant subscriptions (queries) for a given event (tuple).

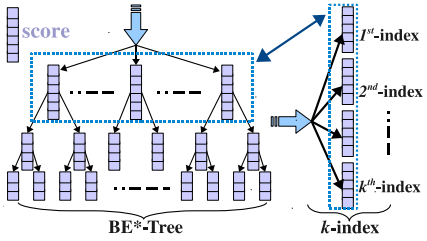


Fig. 1. Top-k Structural Comparisons

In publish/subscribe, two main categories of matching algorithms have been proposed: counting-based [10], [5], [2] and tree-based [4], [6], [7] approaches. These approaches can further be classified as either key-based, in which for each expression a set of predicates are chosen as identifier [5], [7], or as non-key based [10], [6], [2]. Alternatively, these approaches can also be classified either as rigid- [10], [5], [2], [7] and non-rigid-based (BE*-Tree) approaches. What constitutes rigidity is a pre-determined clustering of the space. For example, for a skewed distribution in which all data is clustered in a small subspace, rigid clustering continues splitting the space in half, irrespective of where the data is actually located, and is unable to avoid indexing the empty subspace.

A closer look manifests that counting-based methods primarily aim to minimize the number of predicate evaluations by constructing an inverted index over all unique predicates resulting in a rigid clustering. The two most efficient counting-based algorithms are Propagation [5], a key-based method, and the k-index [2], a non-key-based method. However, BE*-Tree, unlike Propagation, supports promoting non-equality predicates as key and supports continuous domains. Furthermore, BE*-Tree unlike k-index, is a dynamic structure that supports both discrete and continuous domains.

Likewise, tree-based methods are primarily designed to reduce predicate evaluations and to recursively divide search space by eliminating expressions on encountering unsatisfiable predicates. The first major tree-based approach, Gryphon, is a static, a non-key based method [4], which is shown to be effective only for equality predicates [4]. The latest tree-based structure is BE-Tree, a key-based approach, that introduces a two-phase space-cutting abstraction for supporting workload changes and overcoming the curse of dimensionality [7]. BE-Tree leverages an effective rigid clustering that is inherently designed for finite and discrete domains.

In comparison, BE*-Tree develops a non-rigid clustering through a self-adjusting mechanism using the bi-directional tree expansion and the overlap-free splitting strategy that together let BE*-Tree adapt to skewed workloads, cluster only non-empty subspaces, and improve search space pruning. Above all, BE*-Tree enables the top-k relevance-match-determination model. Although BE*-Tree addresses the rigidity and top-k limitations of BE-Tree, yet BE-Tree has two notable advantages over BE*-Tree: it is algorithmically easier to implement and robust w.r.t. insertion sequence (a desirable property also lacking in R-Tree family of indexes [11]).

A fresh look at enhancing existing pub/sub matching algorithms is to leverage top-k processing techniques, which not only concentrate on determining the most relevant matches, but can also improve matching performance. An early top-k model is presented in [1]; however, this model is based on a fixed and predetermined scoring function, i.e., the score for each expression is computed independent of the incoming event. In addition, this approach is an extension of the R-Tree, the interval tree, or the segment tree structure; as a result, it has difficulties scaling beyond a few dimensions [1]. In contrast, a scalable top-k model, but based on a static and flat structure, with a generic scoring function, which also takes the event into consideration, is introduced in k-index [2]. BE*-Tree's top-k model not only scales to thousands of dimensions but introduces a hierarchical top-k model. Our top-k model supports a generic scoring function, that achieves an effective pruning power for determining the most relevant matches.

BE*-Tree's hierarchical top-k structure differs from the flat structure of k-index [2]. This key structural differentiation is depicted in Figure 1. A hierarchical model achieves enhanced pruning because it refines upper bound scores as more expressions are seen (as traversing down the tree). Such pruning is not possible with a flat structure, in which the upper bound score is computed once at the top-layer because as more expressions are seen, the new information is not reflected back in the original top-layer scores.

III. Language and Data Model Formalism

In this section, we define our Boolean expression language and spatial event data model followed by our (top-k) matching semantics.

A. Notation

Given an n-dimensional space $\Re^n$, we define the projection of $\Re^n$ onto $\Re^k$ as a k-dimensional subspace, denoted by $\pi_{d_1 \cdots d_k}(\Re^n) = \Re^k$, where $\pi_{d_1 \cdots d_k} : \Re^n \rightarrow \Re^k$, $k \leq n$, and each $d_i \in \{d_1 \cdots d_k\}$ represents the i^{th} dimension in $\Re^k$ and corresponds to the j^{th} dimension in $\Re^n$; for ease of notation, we define the identity projection as $\pi_{\mathcal{I}}(\Re^n) = \Re^n$. In addition, we define a k-dimensional bounding box $\mathcal{B}^k$ over $\Re^k$ as

$$\mathcal{B}^k = [\mathtt{min}_1, \mathtt{max}_1] \times \cdots \times [\mathtt{min}_k, \mathtt{max}_k].$$

Let $\xi_j(\mathcal{B}^k) = [\mathtt{min}_i, \mathtt{max}_i]$ be the i^{th} boundary in $\mathcal{B}^k$ defined over the i^{th} dimension in $\Re^k$ which corresponds to the j^{th} dimension in $\Re^n$. Let $\chi_i(\mathcal{B}^k)$ be the center of the i^{th} boundary in $\mathcal{B}^k$. Furthermore, let $\lambda_i(\mathcal{B}^k)$ be the length of the i^{th} boundary in $\mathcal{B}^k$ given by

$$\lambda_i(\mathcal{B}^k) = \mathtt{max}_i - \mathtt{min}_i.$$

Lastly, let $\mu_i(\mathcal{B}^k) = \mathtt{min}_i$ and $M_i(\mathcal{B}^k) = \mathtt{max}_i$ represent the minimum and the maximum value of the i^{th} boundary of $\mathcal{B}^k$, respectively.

B. Expression Language and Subspace Model

We support a rich Boolean expression language that unifies the subscription language and the event data model. This generalization gives rise to more expressive matching semantics while still encompassing the traditional pub/sub matching problem.

TABLE I
Predicate Mapping

$P^{\mathtt{attr,opt,val,wt}}(x)$	$\gamma(P^{\mathtt{attr,opt,val,wt}}(x))$
$\mathtt{attr} < v_1$	$[-\infty, v_1 - \epsilon]$
$\mathtt{attr} \leq v_1$	$[-\infty, v_1]$
$\mathtt{attr} = v_1$	$[v_1, v_1]$
$\mathtt{attr} \neq v_1$	$[-\infty, \infty]$
$\mathtt{attr} \geq v_1$	$[v_1, \infty]$
$\mathtt{attr} > v_1$	$[v_1 + \epsilon, \infty]$
$\mathtt{attr} \in \{v_1, \cdots, v_m\}$	$[v_1, v_m]$
$\mathtt{attr} \notin \{v_1, \cdots, v_m\}$	$[-\infty, \infty]$
$\mathtt{attr}$ BETWEEN v_1, v_2	$[v_1, v_2]$

In our model, a Boolean expression is a conjunction of Boolean predicates. A predicate is a quadruple: an attribute that uniquely represents a dimension in $\Re^n$; an operator (e.g., relational operators ($<$,$\leq$, $=$, $\neq$, $\geq$,$>$), set operators ($\in$, $\notin$), and the SQL BETWEEN operator); a set of values (for discrete domains) or a range of values (for continuous domains); and an assigned predicate weight, denoted by $P^{\mathtt{attr,opt,val,wt}}(x)$ or more concisely as $P(x)$. A predicate either accepts or rejects an input x such that $P^{\mathtt{attr,opt,val,wt}}(x) : x \rightarrow \{\mathtt{True}, \mathtt{False}\}$, where $x \in \mathtt{Dom}(P^{\mathtt{attr}})$ and $P^{\mathtt{attr}}$ is the predicate's attribute. Formally, a Boolean expression Ω is defined over $\Re^n$ as follows:

$$\Omega = \{P_1^{\mathtt{attr,opt,val,wt}}(x) \wedge \cdots \wedge P_k^{\mathtt{attr,opt,val,wt}}(x)\},$$
$$\textit{where } k \leq n;\ \ i, j \leq k,\ P_i^{\mathtt{attr}} = P_j^{\mathtt{attr}} \ \ \textit{iff} \ \ i = j.$$

We extended the semantics of projection to a Boolean expression in order to enable projecting out predicates associated with certain dimensions

$$\pi_{d_1 \cdots d_h}(\Omega) = \{\Omega' \mid \Omega' = P_1^{\mathtt{attr,opt,val,wt}}(x) \wedge \cdots \wedge P_h^{\mathtt{attr,opt,val,wt}}(x),$$
$$\forall d_i \in \{d_1 \cdots d_h\},\ P_i^{\mathtt{attr}} = d_i,\ P_i(x) \in \Omega\}.$$

Now, we are in a position to formalize the predicate mapping[1] and, ultimately, to (approximately) represent an expression as a k-dimensional bounding box.

A predicate $P^{\mathtt{attr,opt,val,wt}}(x)$ is mapped into a 1-dimensional bounding box, denoted by $\gamma(P^{\mathtt{attr,opt,val,wt}}(x)) = \mathcal{B}^1$, as shown in Table I, where ϵ is the machine's epsilon. Therefore, a predicate $P(x)$ is covered by $\mathcal{B}^1 = [\mathtt{min}_1, \mathtt{max}_1]$ only if the permitted values defined by $P(x)$ lie in $[\mathtt{min}_1, \mathtt{max}_1]$; for brevity, we say the predicate is enclosed by $\mathcal{B}^1$. Similarly, we say an expression Ω over $\Re^n$ is partially enclosed by $\mathcal{B}^h$ w.r.t. the projection $\pi_{d_1 \cdots d_h}$, denoted by $\Gamma^{\pi}(\Omega)$.

$$\Gamma^{\pi}(\Omega) = \{\mathcal{B}^h \mid \forall P_i^{\mathtt{attr,opt,val,wt}}(x) \in \pi(\Omega),$$
$$\gamma(P_i(x)) \cap \xi_{P_i^{\mathtt{attr}}}(\mathcal{B}^h) = \gamma(P_i(x))\}.$$

Furthermore, we say an expression Ω is fully enclosed by $\mathcal{B}^k$ when the identity projection $\pi_{\mathcal{I}}$ is applied, $\Gamma^{\pi_{\mathcal{I}}}(\Omega) = \mathcal{B}^k$. Lastly, the smallest $\mathcal{B}^k$, or the minimum bounding box (MBB), that (partially) encloses an expression Ω is given by

$$\Gamma^{\pi}_{\mathtt{min}}(\Omega) = \{\arg\min_{\mathcal{B}_i^k} \ \ \Gamma^{\pi}(\Omega) = \mathcal{B}_i^k\}.$$

The Boolean expression Ω is said to have size k, denoted by $|\Omega| = k$, when having k predicates; hence, Ω is represented by $\Gamma_{\mathtt{min}}(\Omega)$ defined over a k-dimensional subspace.

[1]The mapping strategy for predicates with operator $\in, \notin, \neq$ 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.

C. Top-k Matching Semantics

Our formulation of subscriptions and events as expressions enables us to support a wide range of matching semantics, including the classical pub/sub matching: *Given an event ω and a set of subscriptions $\mathbf{\Omega}$, find all subscriptions $\Omega_i \in \mathbf{\Omega}$ that are satisfied by ω*. We refer to this problem as the *stabbing subscription*[2] $\texttt{SQ}(\omega)$ and formalize it as follows:

$$\texttt{SQ}(\omega) = \{\Omega_i \mid \forall P_q^{\texttt{attr,opt,val,wt}}(x) \in \Omega_i,\ \exists P_o^{\texttt{attr,opt,val,wt}}(x) \in \omega, \\ P_q^{\texttt{attr}} = P_o^{\texttt{attr}}, \exists x \in \texttt{Dom}(P_q^{\texttt{attr}}),\ P_q(x) \wedge P_o(x)\}.$$

Alternatively, we can (approximately) express *stabbing subscription* as subspace matching in $\Re^n$ as follows, where the approximation is due to the mapping function γ:

$$\texttt{SQ}(\omega) = \{\Omega_i \mid \forall P_q^{\texttt{attr,opt,val,wt}}(x) \in \Omega_i,\ \exists P_o^{\texttt{attr,opt,val,wt}}(x) \in \omega, \\ P_q^{\texttt{attr}} = P_o^{\texttt{attr}},\ \gamma(P_q(x)) \cap \gamma(P_o(x)) \neq \emptyset\}.$$

Finally, we adopt the popular vector space scoring used in information retrieval (IR) systems[3] for computing the score of a matched subscription Ω_i for a given event ω (to enable top-k computation), denoted by $\texttt{score}(\omega, \Omega_i)$, and defined by

$$\texttt{score}(\omega, \Omega_i) = \sum_{P_q(x) \in \Omega_i, P_o(x) \in \omega, P_q^{\texttt{attr}} = P_o^{\texttt{attr}}} P_q^{\texttt{wt}} \times P_o^{\texttt{wt}}.$$

Similarly, we compute the upper bound score for an event ω w.r.t. an upper bound weight-summary ($\texttt{sum}_{\texttt{wt}}^{\mathbf{\Omega}}$) for a set of subscriptions $\mathbf{\Omega}$ as follows

$$\texttt{uscore}(\omega, \texttt{sum}_{\texttt{wt}}^{\mathbf{\Omega}}) = \sum_{P_o(x) \in \omega} P_o^{\texttt{wt}} \times \texttt{sum}_{\texttt{wt}}^{\mathbf{\Omega}}(P_o^{\texttt{attr}}),$$

where $\texttt{sum}_{\texttt{wt}}^{\mathbf{\Omega}}(\texttt{attr})$ returns the upper bound score of $\texttt{attr}$ over the set of subscriptions $\mathbf{\Omega}$ which is given by

$$\texttt{sum}_{\texttt{wt}}^{\mathbf{\Omega}}(\texttt{attr}) = \max_{\Omega_i \in \mathbf{\Omega}, P_q(x) \in \Omega_i, P_q^{\texttt{attr}} = \texttt{attr}} P_q^{\texttt{wt}}.$$

IV. BE*-Tree Structural Properties

BE*-Tree is generic index structure for indexing a large collection of Boolean expressions (i.e., subscriptions) and for efficient retrieval of the most relevant matching expressions given a stream of incoming expressions (i.e., events). BE*-Tree supports Boolean expressions with an expressive set of operators defined over a high-dimensional continuous space. BE*-Tree copes with the curse of dimensionality challenge through a non-rigid 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 that identifies highly dense subspaces. The two phases BE*-Tree employs are: (1) *partitioning* which is the global structuring to determine the next best attribute $\texttt{attr}_i$ (i.e., the i^{th} dimension in $\Re^n$) for splitting the space and (2) *non-rigid clustering* which is the local structuring for each partition to determine the best grouping of expressions w.r.t. the expressions' range of values for $\texttt{attr}_i$. BE*-Tree not only supports dynamic insertion and deletion of expressions, but it also adapts to workload changes by incorporating top-down (data and space clustering) and bottom-up (space clustering) expansion within each clustering phase.

The data clustering aims to avoid indexing empty space and to adapt to a skewed workload while space clustering aims to avoid degeneration of the structure in the presence of frequent insertions and deletions of expressions. Conceptually, the space and the data clustering techniques are a hybrid scheme that takes the best of both worlds. On the one hand, the data clustering employs data dependent grouping of expressions to adapt to different workload distributions and to avoid indexing empty space, and on the other hand, the space clustering employs space dependent grouping of expressions to accommodate an insertion-independent mechanism.

In general, BE*-Tree is an n-ary tree structure in which a leaf node contains the actual data (expressions) and an internal node contains partial information about data (e.g., an attribute and a range of values) in its descendant leaf nodes. BE*-Tree consists of three classes of nodes: p-node (partition node) for storing the partitioning information, i.e., an attribute; c-node (cluster node) for storing the clustering information, i.e., a range of values; and l-node (leaf node), being at the lowest level of the tree, for storing the actual data. Moreover, p-nodes and c-nodes are logically organized in a special directory structure for fast tree traversal and search space pruning. Thus, a set of p-nodes are organized in a p-directory (partition directory), and a set of c-nodes are organized in a p-directory (cluster directory). The overall BE*-Tree structure together with its structural properties is depicted in Figure 2.

A. Non-rigid Space-cutting

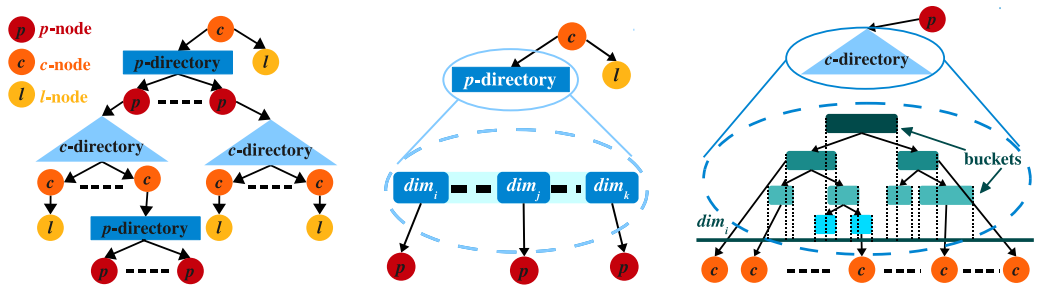
BE*-Tree's non-rigid two-phase space-cutting, the partitioning followed by the non-rigid clustering, introduces new challenges such as how to determine the right balance between the partitioning and clustering, and how to develop a robust principle to alternate between both. In BE*-Tree, we propose a splitting policy to guide the non-rigid clustering phase for establishing not only a robust principle for alternating between partitioning and clustering but also for naturally adapting to the workload distributions. We begin discussing the overall structure and the two-phase cutting dynamics of BE*-Tree before presenting the key design principles behind BE*-Tree.

In BE*-Tree, the partitioning phase, conceptually a global adjusting mechanism, is the first phase of our space-cutting technique. The partitioning phase is invoked once a leaf node overflows (initially occurs at the root level). This phase involves ranking each candidate $\texttt{attr}_i$, attributes that appear in the expressions of the overflowed l-node, in order to determine the most effective $\texttt{attr}_i$ for partitioning, as outlined in Algorithm 4. Essentially, this process identifies the highest ranking $\texttt{attr}_i$, given a scoring function (Section V-B), to spread expressions into smaller groups (leaf nodes). In short, partitioning space based on a high-ranking $\texttt{attr}_i$ enables the pruning of search space more efficiently while coping with the curse of dimensionality by considering a single $\texttt{attr}_i$ for each partitioning phase.

Upon executing the partitioning phase, the high-dimensional indexing problem is reduced to one-dimensional interval in-

[2]This is a relaxation of the stabbing query problem, in which interval cutting is generalized to subspace cutting.

[3]We are not limited to IR scoring, but support any monotonic scoring function.



(a) BE*-Tree high-level structure (b) Partition directory (c) Clustering directory

Fig. 2. The BE*-Tree data structure.

dexing, which paves the way to exploit underlying distribution of a single $\texttt{attr}_i$ at a time through BE*-Tree's clustering phase, conceptually a local adjusting mechanism. At the core of our clustering phase, and ultimately of our BE*-Tree, are (1) a non-rigid clustering policy to group overlapping expressions (into buckets) that minimizes the overlap and the coverage among these buckets and (2) a robust and well-defined policy to alternate between the partitioning and the clustering (summarized in Algorithm 5).

The absence of a well-defined policy gives rise to the dilemma of whether to further pursue the space clustering or to switch back to the partitioning. Furthermore, a partitioned bucket can no longer be split without suffering from the cascading split problem [14]. Thus, a clustering policy that cannot react and adapt to the insertion sequence is either prone to ineffective buckets that do not take advantage of the domain selectivity for effectively pruning the search space or is prone to suffering from substantial performance overhead due to the cascading split problem. Therefore, a practical space clustering must support dynamic insertion and deletion and must adapt to any workload distributions, yet satisfying the cascading-split-free property.

In the clustering phase, each group of expressions is referred to as a bucket. Formally, a *bucket* is a 1-dimensional bounding box over $\texttt{attr}_i$, denoted by $\mathcal{B}^1$, and an expression Ω with predicate $P_j^{\texttt{attr,opt,val,wt}}(x) \in \Omega$ is assigned to a bucket only if

$$\texttt{attr}_i = P_j^{\texttt{attr}} \quad \text{and} \quad \gamma(P_j(x)) \cap \xi_{P_j^{\texttt{attr}}}(\mathcal{B}^1) = \gamma(P_j(x)).$$

Furthermore, a bucket has a minimum bounding box (MBB) that partially encloses all of its expressions $\mathbf{\Omega}$ if and only if

$$\mathcal{B}^1 = \bigcup_{\Omega_j \in \mathbf{\Omega}} \Gamma_{\texttt{min}}^{\pi_{\texttt{attr}_i}}(\Omega_j).$$

Moreover, each bucket is associated with exactly one c-node in the BE*-Tree, which is responsible for storing and maintaining information about the bucket's assigned expressions.

B. Structural Adaptation

The problem of supporting dynamic insertion and deletion is further complicated due to the non-rigid and dynamic clustering introduced by BE*-Tree. In a rigid clustering that relies on deterministically dividing the space in half, a best-effort deletion strategy is adequate because the size of the clustering boundary is deterministic and fixed; thereby, removing an expression does not alter the boundary. However, the non-rigid clustering directory of BE*-Tree defines a relative (non-deterministic) bucket boundary as minimum boundary to enclose all of the bucket's expressions. Consequently, a simple best-effort approach for maintaining the boundary is inadequate because for a pathological case the boundary in BE*-Tree can unnecessarily span a large space, if an expression, defined over a large subspace, is inserted and later removed. To overcome this bucket boundary degeneration, BE*-Tree introduces a set of inactive dimensions which are used exclusively for efficiently keeping the boundary size up-to-date in presence of insertions and deletions. To incorporate the concept of inactive dimensions, we revise the bucket definition as follows:

A *bucket* at the k^{th} level of BE*-Tree is a k-dimensional bounding box, $\mathcal{B}^k$ over $\Re^k$, where $\pi_{d_1 \cdots d_k}(\Re^n) = \Re^k$, and each $d_i \in \{d_1 \cdots d_k\}$ is chosen by the partitioning phase at the i^{th} level of BE*-Tree. Also, we refer to the first $k-1$ dimensions of $\mathcal{B}^k$ as inactive while we refer to the k^{th} dimension (where $d_k \in \Re^n$) as active dimension. Furthermore, the bucket's partial enclosure property w.r.t. its expressions $\mathbf{\Omega}$ holds if and only if

$$\mathcal{B}^k = \bigcup_{\Omega_j \in \mathbf{\Omega}} \Gamma_{\texttt{min}}^{\pi_{d_1 \cdots d_k}}(\Omega_j).$$

The $k-1$ inactive dimensions correspond to $k-1$ attributes selected by the partitioning phase on the path from the root to the current clustering directory at the k_{th}-level of BE*-Tree. These inactive dimensions are utilized only during deletion while the active dimension is used for matching, insertion, and deletion. Without the notion of inactive dimension, deleting an expression results in the *cascading update problem*. The cascading update problem is an unpredictable chain reaction that propagates downwards in order to compute a new bucket boundary after an expression in one of the subtree's buckets has been removed. Thus, at every level of the tree, a brute force boundary re-computation of every bucket is required. However, with inactive dimensions in place, the boundary re-computation is required only for the bucket that is associated with the leaf node from which the expression was removed (or updated). Moreover, this information needs to be only sent upward along the path to the root. Thus, at the higher levels of BE*-Tree, the new boundary can simply be reconstructed by considering the updated boundary information together with the boundary of unmodified buckets. The maintenance of active and inactive dimensions is handled by Algorithm 3.

C. BE*-Tree Invariance

Before stating BE*-Tree's invariance and operational semantics, we must distinguish among four bucket types: *open*

bucket: a bucket that is not yet partitioned; *leaf bucket*: a bucket that has no children (or has not been split); *atomic bucket*: a bucket that is a single-valued bucket which cannot further be split; and *home bucket*: a bucket that is the smallest existing bucket that encloses the inserting expression.

The correctness of the BE*-Tree operational semantics is achieved based on the following three rules:

1) *insertion rule*: an expression is always inserted into the smallest bucket that encloses it, while respecting the descendant-repelling property (Section V-A3).
2) *forced split rule*: an overflowing non-leaf bucket is always split before switching back to the partitioning.
3) *merge rule*: an underflowing leaf bucket is merged with its parent only if the parent is an open bucket

Finally, the BE*-Tree correctness can be summarized as follows: INVARIANCE: *Every expression Ω is always inserted into the smallest bucket that encloses it and a non-atomic bucket is always split first before it is partitioned.*

V. BE*-TREE ADAPTIVENESS PROPERTIES

An essential property of any index structure is to dynamically adapt as the workload changes in order to prevent index deterioration. Supporting adaptiveness is central to the design of BE*-Tree and has shaped every aspect of BE*-Tree. The adaptive nature of BE*-Tree is reflected in its bi-direction expansion that foresees index evolution, in its splitting strategy that improves search-space pruning w.r.t. the subscription workload, and in its proposed simplified ranking function that actively refines the space partitioning based on both subscription and event workloads. The core of BE*-Tree adaptiveness is captured by its insertion operation. Also, the adaptiveness in deletion is accomplished through maintenance of inactive dimension, which was discussed in Section IV-B, and through recycling nodes [7] (i.e., reinsertion policy) whose rank have dropped below certain threshold, our ranking function is defined in Section V-B.

A. Bi-directional Expansions

The main aim of the bi-directional expansion is to determine the right balance between a rigid clustering of the grid-based approach (space dependent) and a flexible and a dynamic clustering similar to that of an R-Tree-based approach (data dependent), while avoiding the cascading split problem. Therefore, the final objective is to enable BE*-Tree to model any data distribution (e.g., skewed), and only in the worst case degenerate to a grid-based approach which is best suited for a uniform distribution.

1) Top-down Expansion: The top-down expansion is part of the clustering phase in which an overflowing bucket is recursively split while keeping the bucket capacity within a threshold, denoted by A. There are two main types of bucket splitting: a data dependent splitting strategy (data clustering), discussed in Section V-A3, and a space dependent splitting strategy (space clustering) which follows a grid-based splitting, also in Section V-A3. As a result, whenever the data clustering approach fails to split the data into two or more buckets, while satisfying a minimum capacity requirement, denoted by α, for each bucket, it relies on space clustering

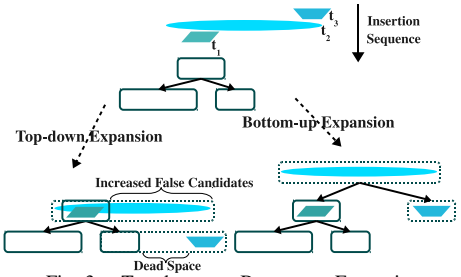


Fig. 3. Top-down vs. Bottom-up Expansion

such that the bucket is split in half, as shown in Algorithm 2, Lines 11-25.

2) Bottom-up Expansion: The bottom-up expansion reverses the conventional tree top-down expansion. It focuses on dynamically adjusting BE*-Tree's clustering directory to accommodate workload changes. Most importantly, the bottom-up expansion avoids BE*-Tree degeneration by preventing the partitioned bucket's size to expand indefinitely. Therefore, to achieve this goal, the clustering directory is re-adjusted by injecting a new bucket in the upward direction, whenever, a partitioned bucket is required to host an expression which is β times larger than the size of all expressions in its subtree. This procedure is illustrated in Figure 3 and its pseudocode is provided in Algorithm 2, Lines 6-10.

3) Descendant-repelling & Overlap-free Split: We present a novel splitting strategy for overflowing buckets, a strategy that falls in between the traditional overlapping-style splitting and grid-style rigid splitting. Our new splitting algorithm aims at systematically reducing the overlap, similar to grid-based structures, while adapting to the data distribution, similar to R-Tree-based structures. To achieve these properties, the overflowing bucket is split into three buckets: left, overlapping, and right buckets, such that the left and right buckets (descendant buckets) repel each other. Although their boundary can be dynamically changed to adapt to the data distribution, they can never overlap. Consequently, data that spans over the boundary of the descendant buckets is placed in the overlapping node. Thus, by definition the overlapping bucket is the home bucket to all of its data, and upon overflowing, the overlapping bucket is handled by switching back to the partitioning phase.

An important property of the overlap-free splitting, guaranteed under the uniform distribution, is realized when *a point event* (i.e., consisting of only equality predicates) falls within the overlapping region. This results in a substantial reduction of the search space in which only $\frac{2}{3}$ of the data must be scanned as opposed to checking all of the data when only two buckets are used and bucket-overlap is allowed. The $\frac{2}{3}$-reduction in search space is due to the proposed splitting of overflowing buckets into three buckets, a left child, a right child, and an overlapping bucket such that left and right children never intersect, which is enforced by our descendant-repelling property. This overlap-free splitting guarantees that a point-event can cut through at most one child. Thus, in the worst case only $\frac{2}{3}$ of the data is visited[4]. The resulting reduc-

[4]The $\frac{2}{3}$-reduction can simply be applied to extended events (events with non-equality predicates), if the descendant-repelling property is generalized to enforce that sibling buckets must be separated by the average extension size of events, where the extension is defined as $\gamma(P^{\texttt{attr},\texttt{opt},\texttt{val},\texttt{wt}}(x))$.

tion is substantial because an entire BE*-Tree subtree could be associate with each bucket. Finally, under the descendant-repelling property, all subscriptions under any sibling buckets are guaranteed to be disjoint on at least one dimension. Next, we present two algorithms to construct descendant-repelling and overlap-free splitting.

Exhaustive Splitting Algorithm Our first algorithm finds an effective splitting hyperplane to divide overflowing buckets into three buckets. In the splitting of the overflowing bucket within the clustering phase at the k^{th} level of BE*-Tree, first all the expressions $\Omega_i \in \boldsymbol{\Omega}$ in the overflowing bucket are sorted w.r.t. the active dimension (k^{th} dimension of the overflowing bucket, denoted by $\texttt{attr}^k$) of BE*-Tree based on both upper and lower boundary of $P^{\texttt{attr}}(x) \in \Omega_i$, where $P^{\texttt{attr}} = \texttt{attr}^k$. The cost of the exhaustive splitting algorithm is dominated by the sorting phase together with enumeration over all possible splitting choices, namely, $O((n-2\alpha)n\log(n))$, where $n = |\boldsymbol{\Omega}|$ and α is the minimum capacity of the resulting right and left buckets after splitting. In brief, the algorithm first assigns the I leftmost expressions to the left bucket (or I rightmost expressions to the right bucket); second, it distributes the remaining expressions to the right (or the left) and the overlapping buckets accordingly. The main objective of the algorithm is to find the optimal I that minimizes overlap and coverage while satisfying the descendant-repelling property and minimum bucket capacity α. The splitting is formally defined as follow:

$$\texttt{buckets}_{\texttt{opt}} = \begin{cases} \texttt{split}^{\texttt{opt}}_{\texttt{dataClustering}}(\texttt{find}^{\texttt{opt}}(\boldsymbol{\Omega}_{\texttt{low}}, \boldsymbol{\Omega}_{\texttt{up}})) & \textbf{if } {}^{|\mathcal{B}^k_R|\geq\alpha\wedge}_{|\mathcal{B}^k_L|\geq\alpha} \\ \texttt{split}_{\texttt{spaceClustering}}(\boldsymbol{\Omega}, s) & \textbf{otherwise}, \end{cases}$$

where $s = \chi_k(\mathcal{B}^k)$, $\mathcal{B}^k$ is the overflowing bucket at the k^{th} level of BE*-Tree, and $\mathcal{B}^k_R, \mathcal{B}^k_O, \mathcal{B}^k_L$ are the new right, overlapping, and left buckets, respectively, returned by either split functions. Furthermore, the most cost effective splitting, denoted by $\texttt{find}^{\texttt{opt}}$, is determined by a set of buckets (right, overlapping, left) which minimizes the total overlap, to break ties among candidate bucket sets, the set with the smallest coverage is selected. The function $\texttt{find}_{\texttt{opt}}$ is formally defined as follows:

$$(\boldsymbol{\Omega_j}, I) \leftarrow \texttt{find}^{\texttt{opt}}(\boldsymbol{\Omega}_{\texttt{low}}, \boldsymbol{\Omega}_{\texttt{up}}) \leftarrow \underset{(\boldsymbol{\Omega_j}, I)\in \texttt{min}_{\texttt{overlap}}(\boldsymbol{\Omega}_{\texttt{low}}, \boldsymbol{\Omega}_{\texttt{up}})}{\arg\min} \texttt{coverage}(\boldsymbol{\Omega_j}, I)$$

$$\{(\boldsymbol{\Omega_j}, I)\} \leftarrow \texttt{min}_{\texttt{overlap}}(\boldsymbol{\Omega}_{\texttt{low}}, \boldsymbol{\Omega}_{\texttt{up}}) \leftarrow \left\{ \underset{\substack{\boldsymbol{\Omega_j}\in\{\boldsymbol{\Omega_{low}}, \boldsymbol{\Omega_{up}}\},\\ I\in(\alpha\cdots|\boldsymbol{\Omega}|-\alpha)}}{\arg\min} \texttt{overlap}(\boldsymbol{\Omega_j}, I) \right\},$$

where Ω_{low} represents the expressions in the overflowing bucket $\mathcal{B}^k$ sorted in descending order w.r.t. the lower bound of the expressions' k^{th} dimension; similarly, Ω_{up} represents the expressions sorted in ascending order w.r.t. the upper bound on the k^{th} dimension.

The overlap and coverage are computed over k^{th} dimension of the bucket as follows:

$$\texttt{overlap}(\boldsymbol{\Omega}, I) = \Big(\lambda_k\big(\pi_k(\mathcal{B}^k_O)\big) \mid \mathcal{B}^k_L, \mathcal{B}^k_O, \mathcal{B}^k_R \leftarrow \texttt{split}^{\texttt{opt}}_{\texttt{dataClustering}}(\boldsymbol{\Omega}, I)\Big)$$

$$\texttt{coverage}(\boldsymbol{\Omega}, I) = \Big(\lambda_k\big(\pi_k(\mathcal{B}^k_L) \cup \pi_k(\mathcal{B}^k_O) \cup \pi_k(\mathcal{B}^k_R)\big) \mid \mathcal{B}^k_L, \mathcal{B}^k_O, \mathcal{B}^k_R \leftarrow \texttt{split}^{\texttt{opt}}_{\texttt{dataClustering}}(\boldsymbol{\Omega}, I)\Big).$$

The descendant-repelling and overlap-free buckets are defined as follows[5], which is the basis of our data clustering procedure:

$$\begin{aligned}(\mathcal{B}^k_L, \mathcal{B}^k_O, \mathcal{B}^k_R) \leftarrow\ & \texttt{split}^{\texttt{opt}}_{\texttt{dataClustering}}(\boldsymbol{\Omega}, I) = \\ & \bigcup_{i=1}^{I} \Big(\mathcal{B}^k_L \leftarrow \Gamma^{\pi_{d_1}\cdots d_k}_{\texttt{min}}(\Omega_i)\Big), \\ & \bigcup_{i=I+1}^{|\boldsymbol{\Omega}|} \Big(\big(\mathcal{B}^k_R \leftarrow \Gamma^{\pi_{d_1}\cdots d_k}_{\texttt{min}}(\Omega_i) \mid \pi_k(\mathcal{B}^k_L) \cap \gamma(\pi_{d_k}(\Omega_i)) = \emptyset\big), \\ & \qquad \big(\mathcal{B}^k_O \leftarrow \Gamma^{\pi_{d_1}\cdots d_k}_{\texttt{min}}(\Omega_i) \mid \pi_k(\mathcal{B}^k_L) \cap \gamma(\pi_{d_k}(\Omega_i)) \neq \emptyset\big)\Big).\end{aligned}$$

In contrast, if no data clustering exists that produces left and right buckets with at least α expressions, then we rely on space clustering, a default clustering which divides the space in half in order to avoid degeneration of the clustering directory.

$$\begin{aligned}(\mathcal{B}^k_L, \mathcal{B}^k_O, \mathcal{B}^k_R) \leftarrow\ & \texttt{split}_{\texttt{spaceClustering}}(\boldsymbol{\Omega}, s) = \\ & \bigcup_{i=1}^{|\boldsymbol{\Omega}|} \Big(\big(\mathcal{B}^k_L \leftarrow \Gamma^{\pi_{d_1}\cdots d_k}_{\texttt{min}}(\Omega_i) \mid \mu(\gamma(\pi_{d_k}(\Omega_i))) > s\big), \\ & \qquad \big(\mathcal{B}^k_R \leftarrow \Gamma^{\pi_{d_1}\cdots d_k}_{\texttt{min}}(\Omega_i) \mid M(\gamma(\pi_{d_k}(\Omega_i))) < s\big), \\ & \qquad \big(\mathcal{B}^k_O \leftarrow \Gamma^{\pi_{d_1}\cdots d_k}_{\texttt{min}}(\Omega_i) \mid s \cap \gamma(\pi_{d_k}(\Omega_i) \neq \emptyset)\big)\Big).\end{aligned}$$

Approximate Splitting Algorithm Unlike the exhaustive algorithm, the cost of our fast splitting algorithm is dominated by sorting expressions, namely, $O(n\log(n))$, where $n = |\boldsymbol{\Omega}|$. In a nutshell, the approximate algorithm assigns expressions one-by-one from the leftmost expression to the left bucket and from the rightmost expression to the right bucket while pushing expressions that span both left and right bucket into the overlap bucket. Formally the algorithm is defined as follows:

$$\texttt{buckets}_{\texttt{apx}} = \begin{cases} \texttt{split}^{\texttt{apx}}_{\texttt{dataClustering}}(\boldsymbol{\Omega}_{\texttt{low}}, \boldsymbol{\Omega}_{\texttt{up}}) & \textbf{if } {}^{|\mathcal{B}^k_R|\geq\alpha\wedge}_{|\mathcal{B}^k_L|\geq\alpha} \\ \texttt{split}_{\texttt{spaceClustering}}(\boldsymbol{\Omega}, s) & \textbf{otherwise}, \end{cases}$$

where $s = \chi_k(\mathcal{B}^k)$.

$$\begin{aligned}(\mathcal{B}^k_L, \mathcal{B}^k_O, \mathcal{B}^k_R) \leftarrow\ & \texttt{split}^{\texttt{apx}}_{\texttt{dataClustering}}(\boldsymbol{\Omega}_{\texttt{up}}, \boldsymbol{\Omega}_{\texttt{low}}) = \\ & \bigcup_{i=1}^{|\boldsymbol{\Omega}_{\texttt{up}}|} \Big(\big(\mathcal{B}^k_L \leftarrow \Gamma^{\pi_{d_1}\cdots d_k}_{\texttt{min}}(\Omega_{\texttt{up}_i}) \mid \gamma(\pi_{d_k}(\Omega_{\texttt{up}_i})) \cap \pi_k(\mathcal{B}^k_R) = \emptyset\big), \\ & \qquad \big(\mathcal{B}^k_R \leftarrow \Gamma^{\pi_{d_1}\cdots d_k}_{\texttt{min}}(\Omega_{\texttt{low}_i}) \mid \gamma(\pi_{d_k}(\Omega_{\texttt{low}_i})) \cap \pi_k(\mathcal{B}^k_L) = \emptyset\big), \\ & \bigcup_{i=1}^{|\boldsymbol{\Omega}_{\texttt{up}}|} \Big(\mathcal{B}^k_O \leftarrow \Gamma^{\pi_{d_1}\cdots d_k}_{\texttt{min}}(\Omega_{\texttt{up}_i}) \mid \Omega_{\texttt{up}_i} \notin \mathcal{B}^k_L \wedge \Omega_{\texttt{up}_i} \notin \mathcal{B}^k_R\Big).\end{aligned}$$

[5]The provided definition of $\texttt{split}_{\texttt{dataClustering}}$ is used when computing buckets over $\boldsymbol{\Omega_{up}}$; for applying it to $\boldsymbol{\Omega_{low}}$, the computation of $\mathcal{B}^k_L$ and $\mathcal{B}^k_R$ must simply be reversed.

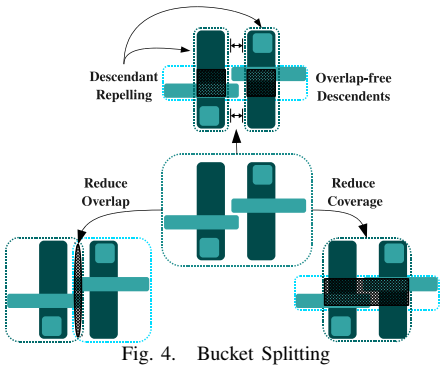


Fig. 4. Bucket Splitting

B. Adaptive Ranking Function

Our ranking function simplifies BE-Tree's ranking function while maintaining the actual index objective, namely, reducing the matching cost. The matching cost is defined as the total number of predicate evaluations: *minimizing false candidate computations* and *minimizing true candidate computations*. The former cost is the total number of predicate evaluations for each discarded search path or for each unsatisfied expression, i.e., penalizing multiple search paths or paths resulting in many false positives. The latter cost is the number of predicate evaluations before concluding a set of expressions as matched, i.e., promoting evaluation of common predicates among expressions exactly once.

Furthermore, our ranking model relies on the notion of *false candidates: the scanned expressions (subscriptions) that are not matched by an input expression (events).* Our model assigns a rank to each node n_i using the function $\texttt{rank}(n_i)$ which is a combination of the $\texttt{loss}(n_i)$ and $\texttt{gain}(l_j)$ functions. $\texttt{loss}(n_i)$ computes for each node the false candidates generated over a window of m events. $\texttt{gain}(l_j)$ computes the number of common predicates for each of its expressions. Formally, $\texttt{rank}(n_i)$, $\texttt{loss}(n_i)$, and $\texttt{gain}(l_j)$ are defined as follows:

$$\texttt{rank}(n_i) = \begin{cases} \texttt{gain}(n_i) - \texttt{loss}(n_i) & \text{if } n_i \text{ is a } l\text{-node} \\ \Big(\sum\limits_{n_j \in \texttt{des}(n_i)} \texttt{rank}(n_j) \Big) - \texttt{loss}(n_i) & \text{otherwise,} \end{cases}$$

where $\texttt{des}(n_i)$ returns n_i's immediate descendants

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

$$\texttt{gain}(l_j) = \#\ \texttt{common pred.}$$

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

$$\texttt{rank}(l_i) = \texttt{gain}(l_i) - \texttt{loss}(l_i),$$

where $\texttt{gain}(l_i)$ is approximated by the number of expressions that have a predicate on $\texttt{attr}_i$, and $\texttt{loss}(l_i)$ is estimated either using the selectivity by constructing a histogram, in which $\texttt{loss}(l_i)$ is the average bucket size in the histogram, or using popularity, an optimistic approach, by initially setting $\texttt{loss}(l_i)$ to zero to eliminate any histogram construction and to rely on the matching feedback mechanism for adjusting the ranking if necessary. In our experimental evaluation, the optimistic approach results in an improved matching and insertion time.

Algorithm 1 $\texttt{Insert}(\Omega, cnode, cdir)$

```
1:  /* Find attr with max rank not yet used for partitioning */
2:  if cnode.pdir ≠ NULL then
3:     for P_i^attr ∈ ω do
4:        if !IsUsed(P_i^attr) then
5:           pnode ← SearchPDir(P_i^attr, cnode.pdir)
6:           if pnode ≠ NULL then
7:              foundPartition = true;
8:              if maxRank < pnode.rank then
9:                 maxPnode ← pnode
10:                maxRank ← pnode.rank
    /* if no partitioning found, then insert into the l-node */
11: if !foundPartition then
12:    Insert(Ω, cnode.lnode)
13:    UpdateLNodeTopNScore(Ω, cnode.lnode.sum_wt^Ω)
       /* if c-node is the root, then do partitioning; otherwise
       clustering */
14:    if isRoot(cnode) then
15:       Partitioning(cnode)
16:    else
17:       Clustering(cdir)
18: else
19:    maxCdir ← InsertCDir(Ω, maxPnode.cdir)
20:    insert(Ω, maxCdir.cnode, maxCdir)
21:    UpdatePNodeScore(maxPnode)
22:    UpdatePNodeTopNScore(Ω, maxPnode.sum_wt^Ω)
```

Algorithm 2 $\texttt{InsertCDir}(\Omega, cdir)$

```
1:  B_L^k, B_O^k, B_R^k ← cdir.B_L^k, cdir.B_O^k, cdir.B_R^k
2:  B^k ← B_L^k ∪ B_O^k ∪ B_R^k
3:  if IsLeaf(B_O^k) then
4:     UpdateBucketInfo(Ω, B_O^k)
5:     return cdir
       /* Bottom-up: if total k^th boundary is expanded by more than
       β */
6:  else if λ_k(γ(π_{d_k}(Ω)) ∪ B^k) > β × λ_k(π_k(B^k)) then
7:     B_new^k ← Ω
8:     cdir ←insert— B_new^k ←new parent— B_L^k, B_O^k, B_R^k
9:     UpdateBucketInfo(Ω, B_new^k)
10:    return cdir
       /* Top-down: otherwise */
11: else
12:    /* If descendant-repelling property is violated by inserting
       Ω */
13:    if γ(π_{d_k}(Ω)) ∩ π_k(B_L^k) ≠ ∅ ∧ γ(π_{d_k}(Ω)) ∩ π_k(B_R^k) ≠ ∅ then
14:       UpdateBucketInfo(Ω, B_O^k)
15:       return cdir
16:    else if γ(π_{d_k}(Ω)) ∩ π_k(B_L^k) ≠ ∅ then
17:       return InsertCDir(Ω, B_L^k)
18:    else if γ(π_{d_k}(Ω)) ∩ π_k(B_R^k) ≠ ∅ then
19:       return InsertCDir(Ω, B_R^k)
20:    else
21:       /* insert into a bucket which results in minimum expan-
          sion */
22:       if λ_k(γ(π_{d_k}(Ω)) ∪ B_L^k) < λ_k(γ(π_{d_k}(Ω)) ∪ B_R^k) then
23:          return InsertCDir(Ω, B_L^k)
24:       else
25:          return InsertCDir(Ω, B_R^k)
```

Algorithm 3 `UpdateBucketInfo`(Ω, $cdir.\mathcal{B}^k$)

1: /* *Update the active dimension (the bucket k^{th} dimension)* */
2: $\mathcal{B}^k \leftarrow \gamma(\pi_{d_k}(\Omega)) \cup \pi_k(\mathcal{B}^k)$
/* *Update the inactive dimensions $(1 \cdots (k-1)^{th}$ dimensions)* */
3: $\mathcal{B}^k \leftarrow \Gamma_{\texttt{min}}^{\pi_{d_1 \cdots d_{(k-1)}}}(\Omega) \cup \pi_{1 \cdots (k-1)}(\mathcal{B}^k)$
4: `UpdateCNodeInactiveTopNScore`($\Omega_i, \mathcal{B}^k.cnode.\texttt{sum}_{\texttt{wt}}^{\Omega}$)

Algorithm 4 `Partitioning`($cnode$)

1: $lnode \leftarrow cnode.lnode$
2: **while** `IsOverflowed`($lnode$) **do**
3: $attr \leftarrow$ `GetNextHighestScoreUnusedAttr`($lnode$)
/* *Create new partition for the next highest ranked attr* */
4: $pnode \leftarrow$ `CreatePDir`($attr$, $cnode.pdir$)
/* *Move all the subscriptions with predicate on attr* */
5: **for** $\Omega_i \in lnode.\mathbf{\Omega}$ **do**
6: **if** $\exists P_j^{\text{attr}} \in \Omega_i, P_j^{\text{attr}} = attr$ **then**
7: $cdir \leftarrow$ `InsertCDir`(Ω_i, $pnode.cdir$)
8: `Move`(Ω_i, $lnode$, $cdir.cnode.lnode$)
9: `UpdatePNodeTopNScore`($\Omega_i, pnode.\texttt{sum}_{\texttt{wt}}^{\Omega}$)
10: `Clustering`($pnode.cdir$)
11: `UpdateClusterCapacity`($lnode$)

Algorithm 5 `Clustering`($cdir$)

1: $lnode \leftarrow cdir.cnode.lnode$
2: **if** !`isOverflowed`($lnode$) **then**
3: **return**
4: **if** !`IsLeaf`($cdir$) **or** `IsPartitioned`($cdir$) **then**
5: `Partitioning`($cdir.cnode$)
6: **else**
7: **if** Exhaustive Splitting Algorithm **then**
8: $cdir.\mathcal{B}_L^k, cdir.\mathcal{B}_O^k, cdir.\mathcal{B}_R^k \leftarrow \texttt{buckets}_{\texttt{opt}}$
9: **else**
10: $cdir.\mathcal{B}_L^k, cdir.\mathcal{B}_O^k, cdir.\mathcal{B}_R^k \leftarrow \texttt{buckets}_{\texttt{apx}}$
11: **for** $\Omega_i \in lnode.\mathbf{\Omega}$ **do**
12: **if** $\gamma(\pi_{d_k}(\Omega_i)) \cap \pi_k(cdir.\mathcal{B}_L^k) = \gamma(\pi_{d_k}(\Omega))$ **then**
13: `UpdateBucketInfo`($\Omega_i, cdir.\mathcal{B}_L^k$)
14: `Move`(Ω_i, $lnode, cdir.\mathcal{B}_L^k.cnode.lnode$)
15: **else if** $\gamma(\pi_{d_k}(\Omega_i)) \cap \pi_k(cdir.\mathcal{B}_R^k) = \gamma(\pi_{d_k}(\Omega))$ **then**
16: `UpdateBucketInfo`($\Omega_i, cdir.\mathcal{B}_R^k$)
17: `Move`(Ω_i, $lnode$, $cdir.\mathcal{B}_R^k.cnode.lnode$)
18: `Partitioning`($cdir.\mathcal{B}_O^k.cnode$)
19: `Clustering`($cdir.\mathcal{B}_L^k$)
20: `Clustering`($cdir.\mathcal{B}_R^k$)
21: `UpdateClusterCapacity`($lnode$)

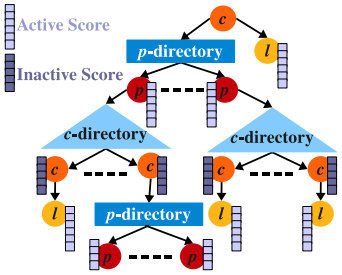


Fig. 5. BE*-Tree Top-k Structural Overview

VI. Top-k Processing

Our top-k algorithm is a two-stage process in which, on the one hand, a cost-based tree traversal is utilized to determine which p-node in BE*-Tree to visit next, and, on the other hand, along the traversal path, a top-k upper bound computation determines whether an l-node content is scanned or not. For an efficient top-k implementation of the former stage, a p-node-max-heap structured is maintained, whereas for the latter, a k-min-heap structure is maintained.

The BE*-Tree's partitioning and clustering abstraction lends itself well for top-k processing. This abstraction also provides a different level of granularity for scoring. For instance, at a p-node level, in which only the information about the attribute space (and the upper bound score for each attribute) is maintained, no maintenance of scoring w.r.t. the distribution of values within each dimension is required. As a result, the upper bound scoring information that must be stored at each p-node can be compressed. This compression results in huge space reduction. Moreover, the same compression technique is applied for storing upper bound scores within each l-node. The overall BE*-Tree top-k structure is illustrated in Figure 5.

The top-k algorithm is as follows: Upon arrival of an event expression, the score between the event and the root's relevant p_i-nodes are computed to form a key (score_{p_i}, p_i-node) which is then inserted into the p-node-max-heap (Algorithm 6 Lines 6-12). Next, the p-node with highest score value is removed from the p-node-max-heap and its subtree is visited (Algorithm 6 Lines 13-19). For the chosen p-node, all of its relevant c_j-nodes (those c_j-nodes having a non-empty intersection with the event) are visited while going through the c-directory (Algorithms 7). For each c_j-node, its l-node is scanned. This process is applied recursively until either the top-k expressions are discovered or all the relevant portions of BE*-Tree are visited. We now discuss the pruning procedures (top-k early terminations):

1) **l-node pruning** Before scanning expressions in an l-node, given an event expression, the upper bound score of an l-node is computed, if the score is larger than the minimum score of the matched expressions seen so far, stored in k-min-heap, or there are fewer than k matched expressions seen so far, then the l-node is visited; it is pruned otherwise, as outlined in Algorithm 6, Line 2.
2) **p-node pruning** Before inserting the p-node into the p-node-max-heap, the score of p-node w.r.t. the event expression is computed; only if the score is larger than the minimum score of matched expressions (or fewer than k matched expressions are gathered so far), then the p-node is inserted. Similarly, before removing the p-node with the highest score from the p-node-max-heap, the score of the highest p-node is compared with the lowest score of matched expressions, if the p-node's score is smaller, then the entire matching process is terminated, resulting in a substantial saving in the matching time, as given in Algorithm 6, Lines 15-17 results.

VII. Evaluations

We present an extensive evaluation of BE*-Tree based on synthetic and real datasets. The experiments were ran on a machine with two Intel Xeon 3.00GHz Quad-core processors having 16GB of memory and running CentOS 5.5 64bit. All algorithms are implemented in C and compiled using gcc 4.1.2.

Algorithm 6 MatchTopk(ω, $\boldsymbol{\Omega_{matched}}$, $cnode$)

```
1: lnodeScore = uscore(ω, lnode.sum_wt^Ω);
2: if lnodeScore < MinScore(k-min-heap) then
3:     /* lnode pruning, skipping an l-node */
4: else
5:     Ω_matched ← FindMatchingExpr(cnode.lnode)
   /* Iterate over event's predicates */
6: for P_i^attr ∈ ω do
7:     /* Retrieve c-node's p-directory for P_i^attr */
8:     pnode ← SearchPDir(P_i^attr, cnode.pdir)
       /* If P_i^attr exists in the p-directory */
9:     if pnode ≠ NULL then
10:        pnodeScore = uscore(ω, pnode.sum_wt^Ω);
11:        if pnodeScore > MinScore(k-min-heap) then
12:            p-node-max-heap ← (pnode, pnodeScore)
13: while !IsEmpty(k-min-heap) do
14:    maxScorePnode ← GetMax(p-node-max-heap)
15:    if IsFull(k-min-heap) then
16:        if maxScorePnode < MinScore(k-min-heap) then
17:            break /* pnode pruning, matching termination */
18:    SearchCDir(ω, Ω_matched, maxScorePnode.cdir)
19:    DeleteMax(p-node-max-heap)
```

Algorithm 7 SearchCDir(ω, $\boldsymbol{\Omega_{matched}}$, $cdir$)

```
1: if γ(π_{d_k}(ω)) ∩ π_k(cdir.B_O^k) ≠ ∅ then
2:     MatchTopk(ω, Ω_matched, cdir.B_O^k.cnode)
3: if γ(π_{d_k}(ω)) ∩ π_k(cdir.B_L^k) ≠ ∅ then
4:     SearchCDir(ω, Ω_matched, cdir.B_L^k)
5: if γ(π_{d_k}(ω)) ∩ π_k(cdir.B_R^k) ≠ ∅ then
6:     SearchCDir(ω, Ω_matched, cdir.B_R^k)
```

A. Experiment Overview

Our evaluation considers the following algorithms: (1) SCAN (a sequential scan), (2) k-ind (the conjunction algorithm implemented over k-index [2] in which all subscriptions are known in advance), (3) BE (BE-Tree [7]), (4) BE*-B (a batch-oriented construction of BE*-Tree in which all subscriptions are known in advance), (5) BE* (a fully dynamic version of BE*-Tree in which subscriptions are inserted individually). Most notably, BE*-Tree is the only structure that is designed to support continuous domains (with the exception of the naive SCAN). Therefore, to compare BE*-Tree with state-of-the-art algorithms, we emulate the effect of continuous domains using a large discrete domain. Furthermore, we primarily focus on the range operator (BETWEEN) over discrete domains. Moreover, we further restrict the size of ranges to only a handful of values, otherwise the enumeration technique of k-index [2] to support range operators would become infeasible due to exponential space explosion.

Our evaluation explores various crucial workload characteristics including *workload distribution*, *workload size*, *space dimensionality*, *average subscription and event size*, *dimension cardinality*, *number of clusters in each dimension*, *dimension cluster size*, and *event matching probability*. To generate various experimental configurations, we leverage the *BEGen*[6] framework [7] including public domain data (i.e., DBLP repository data) to generate real workloads. Various workload configurations are captured in Table III. Due to limited space, in this paper, we primarily focus on a distinguishing subset of experimental results extracted from Table III.

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

TABLE II
BE*-TREE MICRO EXPERIMENTS

(a) Max Leaf Capacity Trends

	A
Match Prob $< 1\%$	5
$1\% \leq$ Match Prob $< 10\%$	20
Match Prob $\geq 10\%$	160

(b) Construction Time

Data Sets	Time (second)
Unif (1M)	2.30
Zipf (1M)	2.45
Author (760K)	4.98
Title (250K)	2.83

B. Micro Experiments

BE*-Tree Parameters BE*-Tree internal parameters are extensively studied and the most robust configuration that resulted in the best overall performance in all experiments is presented here (the in-depth results are omitted). The values used throughout our experiments are A (maximum leaf capacity) shown in Table II(a); $\alpha = 10\%$ (minimum capacity of descendant buckets); $\beta = 4$ (maximum allowable bucket boundary increase before invoking bottom-up expansion); and the exhaustive splitting algorithm is used to handle overflowing buckets.

BE*-Tree Construction The dynamic construction time of BE*-Tree for our largest workload having up to a million subscriptions was below 5 seconds; BE*-Tree's average construction time over our main datasets are reported in Table II(b).

C. Macro Experiments

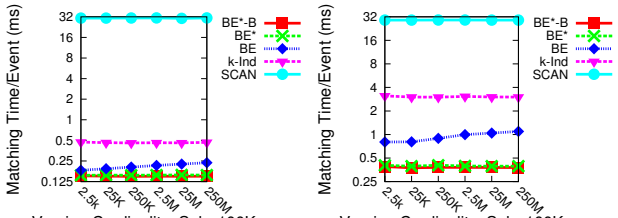
In our macro experiments, we first establish the effectiveness of BE*-Tree's internal techniques, namely, the non-rigid space-cutting, bi-directional tree expansion, and descendant-repelling and overlap-free splitting, then, we shift our focus to BE*-Tree's hierarchical top-k processing that is realized over these unique inner structures of BE*-Tree for finding the most relevant matches.

Workload Distribution A key distinguishing property that sets apart key- vs. non-key-based and rigid-based vs. non-rigid-based algorithms is the workload distribution. The effects of the distributions are shown in Figure 6-9 in which the graphs in the left column correspond to the uniform distribution (for choosing predicates' attributes) while the graphs on the right column correspond to the Zipf distribution with the exception of DBLP-driven workloads which follow a real-data distribution. The observed trend is that k-index, a non-key-based method, is highly sensitive to the underlying data distribution because few popular attributes appear in the majority of subscriptions which result in increased false candidates. The presence of popular attributes forces k-index to sort and scan through posting lists with large number of subscriptions [2]. The increased false candidate problem is reduced by BE-Tree, being a key-based approach. However, BE-Tree also needlessly indexes dead spaces. As a result, BE*-Tree on average outperforms k-index by 60% and 90% and BE-Tree by 35% and 65% for the uniform and the Zipf distributions, respectively.

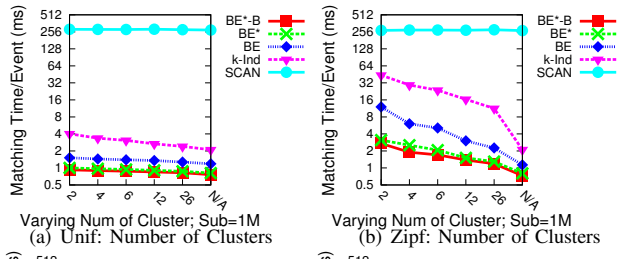
Dimension Cardinality An important factor that further differentiates rigid from non-rigid algorithms is dimension

TABLE III
SYNTHETIC AND REAL WORKLOAD CONFIGURATIONS

	Workload Size	Number of Dimensions	Dimension Cardinality	Number of Clusters	Cluster Size	Sub/Event Size	Match Prob		DBLP (Author)	DBLP (Title)	Match Prob (Author)	Match Prob (Title)
Size	100K-1M	1M	100K	100K	100K	100K	1M		100-760K	50-250K	400k	150
Number of Dim	400	50-1400	400	400	400	400	400		677	677	677	677
Cardinality	250M	250M	2.5K-250M	250M	250M	250M	250M		26^3	26^3	26^3	26^3
Number of Clusters	4	4	4	2-26,NA	4	4	4		—	—	4	4
Cluster Size	100	100	100	100	25-800	100	100		—	—	100	100
Avg. Sub Size	8	8	8	8	8	5-66	8		8	35	8	30
Avg. Event Size	15	15	15	15	15	13-81	15		8	35	16	43
Match Prob %	0.1	0.1	0.1	0.1	0.1	0.1	0.01-9		—	—	0.01-9	0.01-9



(a) Unif: Dim Cardinality (b) Zipf: Dim Cardinality
Fig. 6. Varying Cardinality



(a) Unif: Number of Clusters (b) Zipf: Number of Clusters
(c) Unif: Cluster Size (d) Zipf: Cluster Size
Fig. 7. Varying the Number of Clusters & Cluster Size

cardinality. For a rigid algorithm, BE-Tree, the larger cardinality implies larger tree depth because BE-Tree relies on a grid-based approach that recursively cuts space in half. Therefore, as the cardinality increases, BE-Tree's matching time increases by a factor of 1.29x and 1.36x for the uniform and the Zipf distributions, respectively, while BE*-Tree remains almost unaffected as the cardinality increases and continues to improve over the next best approach by up to 65% (see Figure 6a,b).

Another consequence of increasing the dimension cardinality is the effect on memory footprint. Especially for k-index, increasing dimension cardinality results in an exponential memory growth when the ratio of predicate range size to dimension cardinality is kept constant. The space blow-up occurs because k-index relies on the enumeration technique to support range predicates. For instance, 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.

Number of Clusters/Cluster Size Next, we study the effects of the number of clusters and the cluster size in each domain. If no clustering is employed, then each predicate draws its value randomly from its domain, but when clustering is employed, the drawn values are clustered in certain regions. For example, when the number of clusters is 10, then all drawn values are centered (clustered) at 10 randomly chosen points in the domain. On the other hand, the cluster size models the extent of each individual cluster. For instance, the cluster size 400 implies that all values belonging to a particular cluster are within the Euclidean distance of 400. Studying these workload characteristics are essential when the data is being generated syntactically over a large domain.

k-index is highly sensitive to both the number of clusters and the cluster size. Both fewer clusters and smaller cluster size results in larger overlap among subscriptions and increases the number of false candidates, especially for the Zipf distribution. Thus, from having no cluster, denoted by "N/A", to only two clusters, the matching time is increased by a factor of 20.84x, and from reducing the cluster size from 800 to 25, the matching time is also increased by a factor of 17.44x (Figure 7b,d).

Likewise, BE-Tree, by relying on a rigid clustering, is best suited when predicate values are uniformly drawn, namely, no data clustering, because the rigid clustering fails to fully adapt to the skewed distribution and fails to avoid indexing the dead spaces. As a result, BE*-Tree improves over BE-Tree by up to 77% when data is clustered and by up to 65% when no clustering is used (Figure 7b). Finally, BE*-Tree remains dominant as the number and size of clusters are varied, as observed in Figure 7.

Workload Size/Matching Probability (top-k) To enable top-k processing, we must first generate the predicate weights. In fact, our model is not limited to any particular weight generation technique or scoring function for top-k. Therefore, we adopt the weight generation technique proposed in [2]. In brief, for each unique attribute, we first compute its inverse frequency, denoted by $\texttt{inv(attr)}$, i.e., a popular attribute will be assigned a low weight while a rare attribute will be assigned a high weight. Second, for each expression Ω_i, we compute its predicate weight using a random Gaussian distribution (adopted from [2]) as follows:

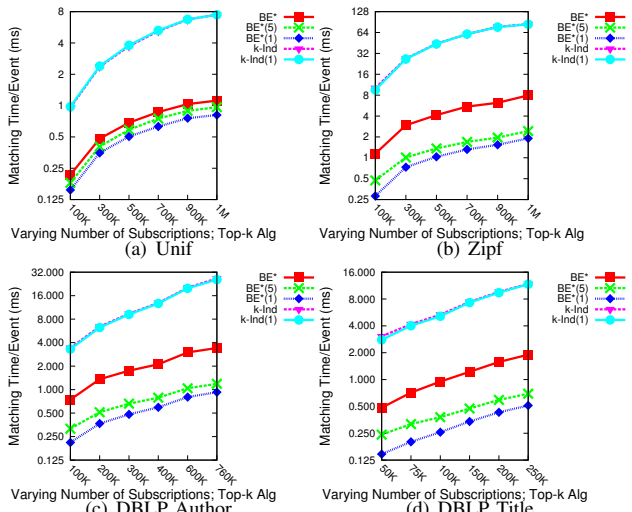


Fig. 8. Top-k vs. Workload Size (Match Prob 0.1%)

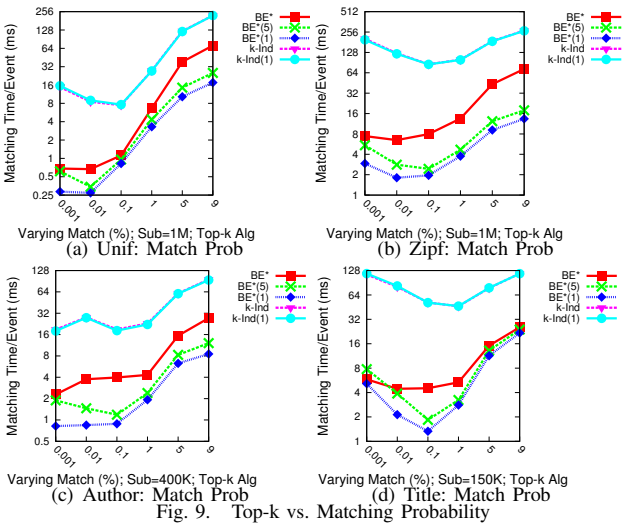


Fig. 9. Top-k vs. Matching Probability

$$\forall P_j(x) \in \Omega_i, P_j^{\text{wt}} = \max(\texttt{inv}(P_j^{\text{attr}}),\ \mathcal{N}(0.8 \times \texttt{inv}(P_j^{\text{attr}}), 0.05 \times \texttt{inv}(P_j^{\text{attr}})))$$

Next, we present BE*-Tree's top-k processing results for varying the workload size (Figure 8) and the matching probability (Figure 9). BE*-Tree hierarchical top-k processing exhibits a significant performance gain compared to k-index for finding the top-1 (denoted by k-ind(1) and BE*(1)) and top-5 (BE*(5)) most relevant expression(s). BE*-Tree's success is attributed to its hierarchical structure that refines the upper bound score as traversing down the tree; however, k-index is restricted to a flat structure that is unable to further refine the upper bound score as more expressions are seen and limited to a fix top-layer upper bound scoring. In short, as we increase the workload size (Figure 8), BE*(1) improves over BE*-Tree by up to $23\%, 77\%, 72\%, 71\%$ while k-ind(1) improves over k-index by only up to $1.3\%, 6.2\%, 3.5\%, 8.5\%$[7] for the uniform, Zipf, author, and title datasets, respectively. Similar trends that show BE*-Tree's effectiveness are also observed as we increase the matching probability (Figure 9). In general, an increase in matching probability results in an increase in the number of candidate matches. Therefore, k-index is forced to scan large posting lists with a reduced chance of pruning and an increased application of sorting to advance through posting lists. The increased matching time is also observed in BE*-Tree simply due to an increased number of matches. However, BE*-Tree scales better as matching time increases and continues to outperform k-ind.

VIII. Conclusions

To address the problem of finding the most relevant matches, we present BE*-Tree, a data structure that employs an effective hierarchical top-k pattern matching algorithm. Furthermore, BE*-Tree introduces a novel non-rigid space-cutting technique to index Boolean expressions over a high-dimensional space by developing (1) a bi-directional tree expansion technique that enables indexing only non-empty continuous sub-spaces and (2) a descendant-repelling and overlap-free splitting strategy.

Moreover, BE*-Tree is a general index structure for efficiently processing large collections of Boolean expressions, a core functionality, required by a wide range of applications including (complex) event processing, computational finance, computational advertising, selective information dissemination, and location-based services. Finally, the performance of BE*-Tree is proven through a comprehensive experimental comparison with state-of-the-art index structures for matching Boolean expressions.

[7] Similar minor improvements were also reported in [2] that showed k-index's top-k achieved only up to 11% improvement over the baseline on some datasets while resulted in up to 19% performance loss on others.

References

[1] A. Machanavajjhala, E. Vee, M. Garofalakis, and J. Shanmugasundaram, "Scalable ranked publish/subscribe," *VLDB'08*.

[2] S. Whang, C. Brower, J. Shanmugasundaram, S. Vassilvitskii, E. Vee, R. Yerneni, and H. Garcia-Molina, "Indexing boolean expressions," in *VLDB'09*.

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

[4] 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*.

[5] 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*.

[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] M. Sadoghi and H.-A. Jacobsen, "BE-Tree: An index structure to efficiently match Boolean expressions over high-dimensional discrete space," in *SIGMOD'11*.

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

[9] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger, "The R*-tree: an efficient and robust access method for points and rectangles," *SIGMOD Rec.'90*.

[10] T. Yan and H. Garcia-molina, "Index structures for selective dissemination of information under the boolean model," *ACM TODS'94*.

[11] V. Gaede and O. Günther, "Multidimensional access methods," *ACM Comput. Surv.'98*.

[12] R. Fagin, A. Lotem, and M. Naor, "Optimal aggregation algorithms for middleware," ser. PODS '01.

[13] I. F. Ilyas, G. Beskales, and M. A. Soliman, "A survey of top-k query processing techniques in relational database systems," *ACM Comput. Surv.'08*.

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