Analysis and Optimization for Boolean Expression Indexing

MOHAMMAD SADOGHI and HANS-ARNO JACOBSEN, University of Toronto

BE-Tree is a novel dynamic 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 an effective 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. Moreover, in BE-Tree, we develop two novel cache-conscious predicate evaluation techniques, namely, lazy and bitmap evaluations, that also exploit the underlying discrete and finite space to substantially reduce BE-Tree's matching time by up to 75%.

BE-Tree is a general index structure for matching Boolean expression which has a wide range of applications including (complex) event processing, publish/subscribe matching, emerging applications in cospaces, profile matching for targeted web advertising, and approximate string matching. Finally, the superiority of BE-Tree is proven through a comprehensive evaluation with state-of-the-art index structures designed for matching Boolean expressions.

Categories and Subject Descriptors: H.3.3 [**Information Storage and Retrieval**]: Inforamtion Search and Retrival—*Information filtering*

General Terms: Algorithms, Design, Measurement, Experimentation, Performance

Additional Key Words and Phrases: Boolean expressions, complex event processing, data structure, publish/subscribe

ACM Reference Format:
Sadoghi, M. and Jacobsen, H.-A. 2013. Analysis and optimization for boolean expression indexing. ACM Trans. Datab. Syst. 38, 2, Article 8 (June 2013), 47 pages.
DOI: http://dx.doi.org/10.1145/2487259.2487260

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 [Aguilera et al. 1999; Fabret et al. 2001; Campailla et al. 2001; Whang et al. 2009], for online advertising and information filtering, Boolean expressions represent advertiser profiles and filters [Machanavajjhala et al. 2008; Whang et al. 2009; Fontoura et al. 2010], and for approximate string matching they can represent string patterns [Fellegi and Sunter 1969; Chaudhuri et al. 2003; Chandel et al. 2007]. In all scenarios, key requirements are the scaling to millions of expressions and to subsecond matching latency. We use a data management scenario for cospaces as in-depth example. Cospaces are an emerging concept to model the coexistence of physical and virtual worlds touted by the Claremont Report as an area of rising interest for database researchers [Agrawal et al. 2008; Ooi et al. 2010]. Consider, for example, a mobile shopping application, where a shopper enters a physical mall and her mobile device submits

Author's address: M. Sadoghi; email: mo@cs.toronto.edu.

DOI: http://dx.doi.org/10.1145/2487259.2487260

her shopping preferences (i.e., subscriptions) to the virtual mall database, as follows: [*genre* = classics, *era* $\in \{1720s, 1730s\}$, *price* BETWEEN $[20, 40]$, *ranking* < 5, *format* $\notin$ {mass market, paperback}]. Now, assume a new promotional item (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* = "*Gulliver's Travels*", *author* = Jonathan Swift, *era* = 1730*s*, *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*. Next, we present the broad applicability of Boolean expression indexing matching; followed by the shortcoming of the existing techniques.

1.1. Motivation

Online Profile Matching. Prominent profile-driven Web applications are targeted Web advertising (e.g., Google, Microsoft, Yahoo!) and job seeker sites (e.g., Monster). For example, in Web advertising, a demographic targeting service specifies constraints such as [*age* $\in \{25, 27, 29\}$] while an incoming user's profile also includes information such as [*age* $= 27$]. Thus, only ads that match a user profile are displayed. Similarly, in online job sites, an employer submits the job detail, [*category* = 'green jobs', *hours*/*week* $>$ 15, and *rate* < 45], while a job seeker registers his profile, [*category* = 'green jobs', *hours*/*week* $= 20$, and *rate* $= 30$], in which the employer is notified of only matching applicants [Machanavajjhala et al. 2008; Whang et al. 2009; Fontoura et al. 2010]. These scenarios require the indexing of potentially millions of expressions and require event matching latency in subsecond.

(Complex) Event Processing. Event processing is gaining rising interest in industry and in academia. The common application pattern is that event processing agents publishes events while other agents subscribe to events of interest. Extensive research has been devoted to developing efficient and scalable algorithms to match events and subscriber's interests [Yan and García-Molina 1994; Aguilera et al. 1999; Fabret et al. 2001; Campailla et al. 2001; Rjaibi et al. 2002; Whang et al. 2009; Fontoura et al. 2010; Farroukh et al. 2011; Cugola and Margara 2012]. The predominant abstraction used in this context, is the content-based publish/subscribe paradigm to model an event processing application. Applications that have been referenced in this space include algorithmic trading and (financial) data dissemination [Sadoghi et al. 2010], business process management [Hull 2008], and sense-and-respond [Chandy et al. 2007].

Big Data Analytical Processing. A key challenge in processing analytical type queries in large databases is to cope with the ever increasing volume and velocity (i.e., data arrival rate in form of update, insertion, and deletion queries) of data. Recently, there has been a new paradigm shift for big data processing which is strikingly similar to event processing. In this new paradigm, disks are continuously scanned, and data is fetched in chunks and pushed to only interested queries. Essentially, in this push-based model, a traditional database is transformed into a streaming database which brings two key benefits: elimination of the need for indexing the data and relying solely on a fast sequential scan of the disk. This new model also partially boils down to efficiently identifying which queries are interested in the latest fetched data chunk (or tuple), that is, indexing queries instead of data. One way to find the interested queries is by extracting a query's selection conditions that are expressed as Boolean expressions, and in turn indexing these expressions. An example of such systems is DataPath [Arumugam et al. 2010].

Data Quality. Data quality has been an active area of research in the database community over the past decade [Fellegi and Sunter 1969; Chaudhuri et al. 2003; Chandel et al. 2007]. In general, data quality is effected by typing mistakes, lack of standards and integrity constraints, and inconsistent data mappings resulting in different representations of identical entities. Therefore, many approximate string matching algorithms have been proposed to identify similar entities [Chaudhuri et al. 2003; Chandel et al. 2007]. These algorithms are based on tokenization of a string into a set of q-grams (a sequence of q consecutive characters). For example, a 3-gram tokenization of "string" is given by {'str', 'tri', 'rin', 'ing'}. One of the main challenges for a q-gram transformation is the curse of dimensionality [Chaudhuri et al. 2003]. That is 3-grams result in at least 26^3 dimensions. For a realistic BE-Tree evaluation over real-world data, we propose a representation of a set of g-grams as Boolean expressions to significantly reduce dimensionality and leverage BE-Tree to actually solve the approximate substring matching problem. How g-grams are converted to Boolean expressions is discussed in Section A of the electronic appendix. As in the previous scenarios, scalability to large expression sets is paramount.

Applications in the Cospaces. The coexistence of virtual and physical worlds brings unique opportunities for a new generation of applications. Applications that use information gathered from the virtual world to continuously enrich a user's physical world experience while using the real-time information gathered from the physical world to refresh the virtual world in turn [Agrawal et al. 2008; Ooi et al. 2010]. Examples of cospace applications involve marketplace applications that allow virtual and physical shoppers to compete (bid on the last item) or cooperate (buy one get one free), location-based gaming that changes the gamer's environment relative to the gamer's physical location, and social networking applications that detects when virtual friends are within a close proximity and initiates a physical interaction among them [Ooi et al. 2010]. We already illustrated a detailed example of where and how expression indexing is important in this context in the introduction of this article.

1.2. Boolean Expression Matching Challenges

There are four major challenges to efficient indexing of Boolean expressions. First, the index structure 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. Fourth, the index must adapt to changing workload patterns.

However, existing techniques are inadequate to satisfy these four requirements. For instance, techniques used in expert and rule-based systems support expressive predicate languages [Forgy 1990], but are unable to scale to millions of expressions. Recent work addresses the scalability limitation, but either restricts the predicate expressiveness [Fabret et al. 2001] or assumes a static environment in which the index is constructed offline [Aguilera et al. 1999; Whang et al. 2009; Fontoura et al. 2010]. Our goal is to address scalability, expressiveness, dynamic construction, and adaptation by proposing a 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 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 [Sadoghi and Jacobsen 2011]. BE-Tree is dynamically constructed through a two-phase space-cutting (i.e., partitioning and clustering) 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.

In this article, we make the following contributions.

(1) We unify subscription and event language to enable a more expressive matching semantics (cf. Section 3), and we evaluate various matching semantics (cf. Section 8).
(2) We propose a novel data structure, BE-Tree, that supports an extensive set of operators and a dynamic schema, gracefully scales to millions of subscriptions, thousands of dimensions, and tens of predicates per subscription and event (cf. Section 4).
(3) We present formal analysis of BE-Tree's key properties and (cf. Section 4 and Sections C-E of the electronic appendix).
(4) We develop a set of novel self-adjusting policies for BE-Tree that continuously adapt to both subscription and event workload changes (cf. Section 5).
(5) We introduce a novel cache-conscious lazy and bitmap-based Boolean predicate evaluation to substantially improve BE-Tree's matching time (cf. Section 6).
(6) We present the first comprehensive evaluation framework, including both micro and macro experiments, that benchmarks state-of-the-art matching algorithms, including SCAN [Yan and García-Molina 1994], SIFT [Yan and García-Molina 1994], Gryphon [Aguilera et al. 1999], our improved Gryphon [Aguilera et al. 1999], Access Predicate Pruning (APP) [Farroukh et al. 2011], Propagation [Fabret et al. 2001], k-index [Whang et al. 2009], and GPU-based CLCB [Cugola and Margara 2012; Margara and Cugola 2013] (cf. Section 8 and Section F of the electronic appendix).

The rest of this article is organized as follows. In Section 2, we survey the related work. In Section 3, we formally define the matching problem and specify the syntax and semantics of Boolean expressions indexed by BE-Tree. Section 4 provides an in-depth description of our BE-Tree. Next, we introduce BE-Tree's self-adjustment mechanism in Section 5 followed by BE-Tree's core optimizations, that is, lazy and bitmap-based predicate evaluations and the Bloom filter optimization in Section 6. Section 7 is dedicated to describing the BE-Tree implementation. Lastly, in Section 8, we present a comprehensive experimental evaluation of BE-Tree in comparison with state-of-the-art approaches. The article is accompanied by an electronic appendix.

2. RELATED WORK

Problems related to indexing Boolean expressions have been studied in many contexts: expert systems [Giarratano and Riley 1989], active databases [Hanson et al. 1990], trigger processing [Hanson et al. 1999], publish/subscribe matching [Yan and García-Molina 1994; Aguilera et al. 1999; Fabret et al. 2001; Campailla et al. 2001; Machanavajjhala et al. 2008; Brenna et al. 2007; Whang et al. 2009; Fontoura et al. 2010; Farroukh et al. 2011; Sadoghi et al. 2011; Sadoghi and Jacobsen 2011; Cugola and Margara 2012; Sadoghi and Jacobsen 2012; Sadoghi 2012; Margara and Cugola 2013], and XPath/XML matching (e.g., [Diao et al. 2003; Chan et al. 2002; Candan et al. 2006; Sadoghi et al. 2011]). Indexing in multidimensional space has been extensively studied (e.g., [Guttman 1984; Beckmann et al. 1990; Sellis et al. 1987; Berchtold et al. 1996; Gaede and Günther 1998].)

The work on expert systems, active databases, and trigger processing [Giarratano and Riley 1989; Hanson et al. 1990; Hanson et al. 1999] as well as certain publish/subscribe (pub/sub) work [Brenna et al. 2007] focus on language expressiveness and not on scaling to thousands of dimensions and millions of expressions. XPath/XML matching [Diao et al. 2003; Chan et al. 2002; Candan et al. 2006; Sadoghi et al. 2011] is based on a completely different language from what BE-Tree supports and is not in the scope of this work. These approaches are therefore not directly applicable, and we concentrate our review on pub/sub matching [Yan and García-Molina 1994; Aguilera et al. 1999; Fabret et al. 2001; Campailla et al. 2001; Rjaibi et al. 2002; Whang et al. 2009; Farroukh et al. 2011].

2.1. Publish/Subscribe Matching

Two main categories of matching algorithms have been proposed: counting-based [Yan and García-Molina 1994; Fabret et al. 2001; Whang et al. 2009; Farroukh et al. 2011; Cugola and Margara 2012; Margara and Cugola 2013] and tree-based [Aguilera et al. 1999; Campailla et al. 2001; Sadoghi and Jacobsen 2011; 2012] approaches. Furthermore, existing work can be further classified as, either key-based in which for each expression a set of predicates are chosen as identifier [Fabret et al. 2001], or as non-key-based [Yan and García-Molina 1994; Campailla et al. 2001; Whang et al. 2009; Farroukh et al. 2011][1] 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 [Fabret et al. 2001], a key-based method, and the k-index [Whang et al. 2009], a non-key-based method. Similarly, tree-based methods are designed to reduce predicate evaluations and to recursively divide search space by eliminating subscriptions on encountering unsatisfiable predicates. Tree-based methods are proven to outperform counting-based algorithms [Kale et al. 2005]. Despite this theoretical result, only few efficient tree-based matching algorithms exist. The most prominent tree-based approach, Gryphon, is a static, non-key-based method [Aguilera et al. 1999]. Our proposed BE-Tree is a novel tree-based approach, which also employs keys, that we show to outperform existing approaches [Yan and García-Molina 1994; Aguilera et al. 1999; Fabret et al. 2001; Whang et al. 2009; Farroukh et al. 2011].

The Propagation algorithm is the state-of-the-art counting-based method with two main strengths [Fabret et al. 2001]. First, a typical counting-based inverted index is replaced by a set of multiattribute hashing schemes; each multiattribute hashing scheme is referred to as an access predicate (i.e., key). Second, keys are selected from a candidate pool using an effective cost-based optimization tuned by the workload distribution [Fabret et al. 2001]. The weaknesses of Propagation are as follows: limiting keys to only a small set of equality predicates in order to use hashing and to avoid exponential blow up in the number of candidate keys; assuming that subscriptions are uniformly distributed across keys to avoid degeneration of hashing into a sequential scan over subscriptions; and maintaining a large collection of candidate hash configurations, using histograms, based on a greedy selection. Our proposed BE-Tree structure overcomes all these shortcomings by employing a novel multilayer structure to avoid hashing degeneration and to enable on-demand creation of histograms as needed; a self-adjusting mechanism to adapt to workload changes without maintaining histograms; and, lastly, to support a rich set of operators beyond the equality predicate.

To enrich Propagation with interval predicate, a Hierarchical Clustering (HC), using a variant of the Propagation cost function, is proposed in Saita and Llirbat [2004]. However, the problem of candidate generation is worsened in HC because each cluster must now maintains a complete set of histograms for all dimensions, for instance, dimension $d = 1000$, cluster size of 100, and 5,000,000 subscriptions, roughly 50,000,000 histograms is needed [Saita and Llirbat 2004]. Furthermore, HC dynamics, merging and splitting, are only local operations between a leaf and its parent and HC global structure fails to adapt to workload changes [Saita and Llirbat 2004]. HC has also shifted its focus to a more general disk-based indexing (as opposed to main memory indexing) that scales to only tens of dimensions, but, most important, HC disregards the key observation that subscriptions tend be defined over a discrete and a finite domain [Yan and García-Molina 1994; Aguilera et al. 1999; Fabret et al. 2001; Carzaniga and Wolf 2003; Whang et al. 2009]. This key domain property introduces new challenges, yet it provides a unique opportunity for further exploitation of the inherit structure, which is fully leveraged in BE-Tree.

[1]The Access Predicate Pruning (APP) [Farroukh et al. 2011] introduces a filtering strategy using the notion of access predicate, but AAP is considered a non-key based method under our strict classification.

The latest advance in the counting-based algorithm is k-index [Whang et al. 2009], which gracefully scales to thousands of dimensions and supports equality predicates ($\in$) and nonequality predicates ($\notin$). k-index partitions subscriptions based on their number of predicates to efficiently prune subscriptions with too few matching predicates; however, k-index is static and does not support dynamic insertion and deletion. What distinguishes 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.

Other, complementary techniques to enhance pub/sub matching are event batch processing [Fischer and Kossmann 2005] and top-k matching [Machanavajjhala et al. 2008; Whang et al. 2009]. The former reduces the number of index lookups by batching similar events. The latter aims to improve matching by only returning the top-k matching subscriptions. The top-k in [Machanavajjhala et al. 2008] is based on a fixed predetermined ranking for each subscription, and this approach leverages the R-tree [Guttman 1984], the interval tree [Berg et al. 2008], or the segment tree [Berg et al. 2008] structure to answer top-k queries [Machanavajjhala et al. 2008]; however, these techniques hardly scale beyond 1-dimension and do not support updating the subscription's rank. In contrast, a scalable, but static, top-k model is introduced in the k-index [Whang et al. 2009].

Another emerging area of research is to improve language expressiveness. For example, the subscription languages in [Campailla et al. 2001; Fontoura et al. 2010] support both Conjunctive Normal Form (CNF) and Disjunctive Normal Form (DNF), while the k-index supports either CNF or DNF subscription language [Whang et al. 2009]. Finally, there is a paradigm shift (symmetric pub/sub) in which events producers are also able to impose filtering conditions on events' subscribers [Rjaibi et al. 2002], which substantially improves the expressive power of the event language; this new paradigm is supported by our BE-Tree and proposed matching semantics.

Orthogonal to matching problem is the distributed content-based routing and subscription propagation algorithms (e.g., [Triantafillou and Economides 2002; Jerzak and Fetzer 2008]). Many of these approaches employs novel techniques based on Bloom filters. However, our usage of Bloom filter is complementary to the core of BE-Tree algorithm and serves only as additional optimization layer for filtering the subscriptions stored in the leaf levels. Similar pruning technique is also explored in Cugola and Margara [2012], and Margara and Cugola [2013].

2.2. Traditional Multidimensional Indexing

An alternative approach in building pub/sub matching engine is to use the multidimensional indexing developed in the database community; the most prominent multidimensional structure is R-tree [Guttman 1984], which supports indexing spatial extended objects, theoretically a suitable index to solve the pub/sub matching problem. The R-tree introduced the idea of overlapping partitions to achieve high space utilization properties, a desirable disk-based property, at the cost of downgrading the retrieval performance [Guttman 1984]. However, this overlapping side effects further worsens as the dimensionality increases (above three) at which point a sequential scan is more efficient [Berchtold et al. 1996]. The problem of reducing overlapping partitions has been tackled from different angles: R^+-tree reduces overlapping by clipping objects, but it results in exponential space blow up [Sellis et al. 1987]; R^*-tree delays splitting and relies on reinsertion and attempts to geometrically minimize the overlap during splitting and insertion [Beckmann et al. 1990]; X-tree is a hybrid of sequential scan and R^*-tree and switches to sequential scan when no overlap-free split exist [Berchtold et al. 1996]. The X-tree is the only structure that scales well to tens of dimensions, yet takes only a passive approach to solve the overlapping problem by exploiting only the physical storage property, that is, favoring sequential vs. random access.

Among many others, interval indexing approaches such as Segment Tree [Berg et al. 2008], Interval Tree [Berg et al. 2008], and *R*-tree [Guttman 1984] have been proposed to index one-dimensional objects. The Segment Tree and Interval Tree are static structures. Although *R*-tree is a dynamic structure, it is sensitive to the insertion sequence, while BE-Tree was designed to be independent of insertion sequence.

Matching in high-dimensional space, diverges from classical database indexing in four important ways. (1) 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 [Guttman 1984; Berchtold et al. 1996; Gaede and Günther 1998]. (2) Expressions indexed by BE-Tree impose restrictions on a small subspace only and are fully defined everywhere else. As a result, there is high degree of overlap among expressions which renders current indexing techniques inapplicable. For instance, *X*-Tree [Berchtold et al. 1996], often the most suitable index for high-dimensional data, degenerates to a sequential scan in situations where all expressions overlap. (3) Much high-dimensional indexing work focuses on high space utilization and reducing random accesses, as opposed to optimize matching time (lookup); disk is assumed as storage medium and disk I/O is the bottleneck. BE-Tree, on the other hand, is a main memory structure. (4) 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 structure.

3. EXPRESSION MATCHING MODEL

In this section, we formalize our Boolean expression language and data model followed by our matching semantics.

3.1. Expression Language

Traditionally, pub/sub matching algorithms take as input a set of subscriptions (conjunction of Boolean predicates) and an event (an assignment of a value to each attribute), and return a 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 $P^{\texttt{attr,opt,val}}(x)$ or more concisely as $P(x)$. A predicate either accepts or rejects an input x such that $P^{\texttt{attr,opt,val}}(x) : x \longrightarrow \{\texttt{True}, \texttt{False}\}$, 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:

$$\begin{aligned} &be = \left\{P_1^{\texttt{attr,opt,val}}(x) \wedge \cdots \wedge P_k^{\texttt{attr,opt,val}}(x)\right\}, \\ &where\ k \leq n;\ \ i, j \leq k,\ P_i^{\texttt{attr}} = P_j^{\texttt{attr}}\ \ iff\ \ i = j. \end{aligned} \tag{1}$$

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.

3.2. Matching Semantics

Our expression subscription and event language enables a wide range of matching semantics, including *stabbing subscription*, *stabbing event*, *symmetric matching*, *containment matching*, *enclosure matching*, and *exact matching*.

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[2] $\texttt{SS}(e)$, and specify the problem as follows:

$$\texttt{SS}(e) = \big\{s_i |\ \forall P_q^{\texttt{attr,opt,val}} \in s_i,\ \exists P_o^{\texttt{attr,opt,val}} \in e, \\ P_q^{\texttt{attr}} = P_o^{\texttt{attr}}, \exists x \in \texttt{Dom}\big(P_q^{\texttt{attr}}\big),\ P_q(x) \wedge P_o(x)\big\}. \tag{2}$$

The reverse direction of stabbing subscription is defined as given an event e and a set of subscriptions, find all subscriptions s_i satisfying e, *stabbing event* $\texttt{SE}(e)$, and it is given as

$$\texttt{SE}(e) = \big\{s_i |\ \forall P_o^{\texttt{attr,opt,val}} \in e,\ \exists P_q^{\texttt{attr,opt,val}} \in s_i, \\ P_o^{\texttt{attr}} = P_q^{\texttt{attr}},\ \exists x \in \texttt{Dom}\big(P_o^{\texttt{attr}}\big),\ P_o(x) \wedge P_q(x)\big\}. \tag{3}$$

The stabbing event enables us to formalize symmetric stabbing which is necessary to model the symmetric pub/sub paradigm [Rjaibi et al. 2002]. This bidirectional matching problem is defined as given an event e and a set of subscriptions, find all subscriptions s_i satisfied by e and satisfying e, which we refer to it as *symmetric matching* $\texttt{SM}(e)$

$$\texttt{SM}(e) = \{\texttt{SS}(e) \cap \texttt{SE}(e)\}. \tag{4}$$

We can also answer much stronger matching semantics, given an event e and a set of subscriptions, find all subscriptions s_i enclosed by e, denoted by *containment matching* $\texttt{CM}(e)$

$$\texttt{CM}(e) = \big\{s_i |\ \forall P_q^{\texttt{attr,opt,val}} \in s_i,\ \exists P_o^{\texttt{attr,opt,val}} \in e, \\ P_q^{\texttt{attr}} = P_o^{\texttt{attr}},\ \forall x \in \texttt{Dom}\big(P_q^{\texttt{attr}}\big),\ P_q(x) \rightarrow P_o(x)\big\}. \tag{5}$$

The reverse direction of containment matching is defined as given an event e and a set of subscriptions, find all subscriptions s_i enclosing e, enclosure matching $\texttt{EM}(e)$

$$\texttt{EM}(e) = \big\{s_i |\ \forall P_o^{\texttt{attr,opt,val}} \in e,\ \exists P_q^{\texttt{attr,opt,val}} \in s_i, \\ P_o^{\texttt{attr}} = P_q^{\texttt{attr}},\ \forall x \in \texttt{Dom}\big(P_o^{\texttt{attr}}\big),\ P_o(x) \rightarrow P_q(x)\big\}. \tag{6}$$

Last, we define *exact matching* $\texttt{XM}(e)$: given an event e and a set of subscriptions, find all subscriptions s_i enclosed by e and enclosing e

$$\texttt{XM}(e) = \{\texttt{CM}(e) \cap \texttt{EM}(e)\}. \tag{7}$$

The matching semantics supported by BE-Tree can be summarized as follows. BE-Tree returns an approximate answer (a subset of answer) for stabbing event and enclosure matching, yet it returns an exact answer for all matching problems of immediate practical interests in Boolean expression indexing: stabbing subscription, symmetric matching, containment matching, and exact matching. Furthermore, our matching semantics can further be classified as either *forward matching* (traditional database indexing semantics, e.g., R-Tree), *reverse matching* (traditional pub/sub matching semantics, in which the role of query and objects is reversed), or *bidirectional matching* (symmetric matching). The subtle difference between forward and reverse matching is due to the fact that the database indexing semantics differs with our proposed semantics in an important respect. Our reverse matching semantics solves the reverse database matching problem. In the database context, querying (matching) means finding the relevant tuples (events) for a given query (subscription). But in our context, matching (querying) means finding the relevant subscriptions (queries) for a given event (tuple).

The three categories of the matching semantics are (1) forward matching, namely, stabbing event and enclosure matching; (2) reverse matching, namely, stabbing subscription and containment matching; and (3) bidirectional matching, namely, symmetric matching and exact matching. Alternately, BE-Tree can be characterized as to return

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

exact answers for both reverse matching and bidirectional matching and to return an approximate answers for forward matching. The distinction between forward and reverse matching semantics is yet another design principle that sets apart BE-Tree from traditional *R*-Tree family of indexes [Gaede and Günther 1998]).

In the remainder of this article, we simply refer to a Boolean expression as an "expression," and we use the term "expression" also to refer to both a subscription and an event. Without the loss of generality, whenever it is not clear from the context, we use the term subscription and event to distinguish between a set of expressions stored in the index and input expression to be matched, respectively.

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 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 attribute $\texttt{attr}_i$, that is, the i^{th} dimension (Section 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 $\texttt{attr}_i$ (Section 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 Section 7.

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 is organized in a *partition directory* (p-directory), and a set of c-nodes is 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 Figure 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-node. 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

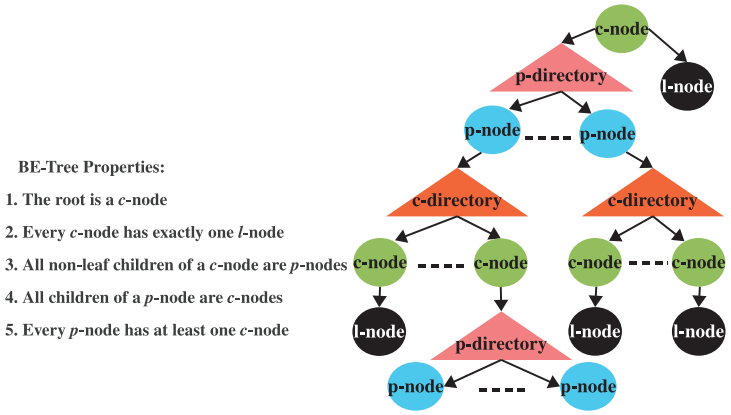


Fig. 1. The BE-Tree data structure.

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. These two phases 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 $P_i^{\texttt{attr},\texttt{opt},\texttt{val}}(x)$ (i.e., a range of permitted values), and all the expressions in its descendant l-nodes must have a predicate $P_j^{\texttt{attr},\texttt{opt},\texttt{val}}(x)$ such that

$$\left(P_i^{\texttt{attr}} = P_j^{\texttt{attr}}\right) \wedge \left(\forall x \in \texttt{Dom}\left(P_j^{\texttt{attr}}\right),\ P_j(x) \rightarrow P_i(x)\right). \tag{8}$$

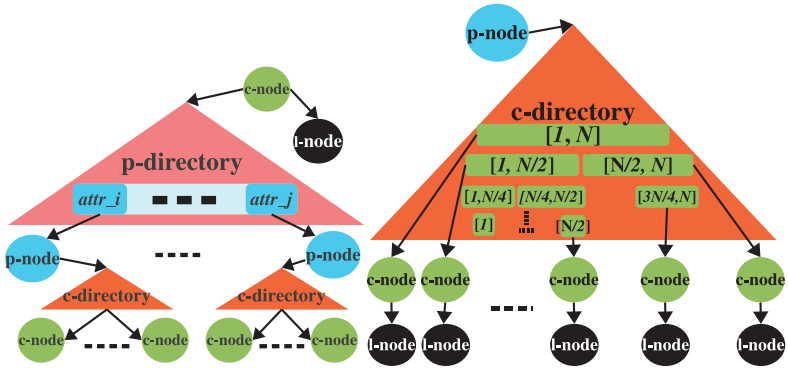
Provided that each c-node is denoted by a predicate $P_j(x)$, we can assign each l-node a key $\texttt{key}_\texttt{j}$ defined as a conjunction of all c-nodes' predicate along the path from the root to the l_j-node.

In what follows, in order to uniquely identify the same category of nodes, 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 refer to BE-Tree internal parameters as follows: $\texttt{max}_{cap}$ (system max leaf capacity), $\texttt{min}_{size}$ (system min partition size), $\texttt{max}_{cap}^{j}$, (l_j-node max capacity), $\texttt{freq}_{\texttt{window}}$, (update frequency window), θ (l-node recycling threshold), $\texttt{ratio}_{exp}$ (exploit vs. explore ratio), and $\texttt{ratio}_{ins}$ (reinsertion ratio).

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. Section 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



(a) Partition directory (b) Generic clustering directory

Fig. 2. The BE-Tree directories.

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 reevaluated as follows:

$$\texttt{max}_{cap}^j = \begin{cases} \left\lceil \frac{|l_j\text{-node}|}{\texttt{max}_{cap}} \right\rceil \times \texttt{max}_{cap}, & \text{if } |l_j\text{-node}| > 0 \\ \texttt{max}_{cap}, & \text{otherwise.} \end{cases} \tag{9}$$

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. Section 4.3). Moreover, we show in Section 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 Figure 2(a).

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 nonleaf 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 the space clustering, and ultimately of 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 [Freeston 1995]: an unpredictable chain reaction that propagates downwards, potentially effecting each node at every level of the tree including the leaf nodes. 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.

To address these challenges, we propose a 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 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 a few concepts.

Definition 4.1. A *bucket* represents an interval boundary (range of values) over $attr_i$.

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.

Definition 4.2. An *open bucket* is a bucket with a not yet partitioned c-node.

Definition 4.3. A *leaf bucket* is a bucket that has no children (a bucket that has not been split).

Definition 4.4. A *discrete bucket* is an atomic bucket that cannot further be split. A discrete bucket is also a leaf bucket, but the reverse direction is not necessarily true.

Definition 4.5. A *home bucket* is the smallest possible bucket that encloses an expression.

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.

Rule 1. An expression is always inserted into the smallest bucket that encloses it (*insertion rule*).

Rule 2. A nondiscrete bucket is always split before its c-node switches to the space partitioning (*forced split rule*).

Rule 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 nonatomic 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 pursed, these expressions are unaffected by it.

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 Section C of the electronic appendix. 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 nonatomic leaf bucket is never partitioned.*

Therefore, the key result with respect to BE-Tree's invariance property, which is proven in Section C of the electronic appendix, can be summarized as follows.

THEOREM 4.6. BE-Tree*'s two-phase space-cutting (space partitioning and clustering) is always safe and always satisfies the* BE-Tree *invariance.*

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 the clustering directory. Finally, we present a set of directory optimization including a specialized clustering and a hybrid clustering directory. The specialized clustering targets a restricted set of operators, and the hybrid clustering utilizes both the generic and the specialized clustering directories to further improve the matching time.

4.4. Space Clustering Structures

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 Figure 2(b).

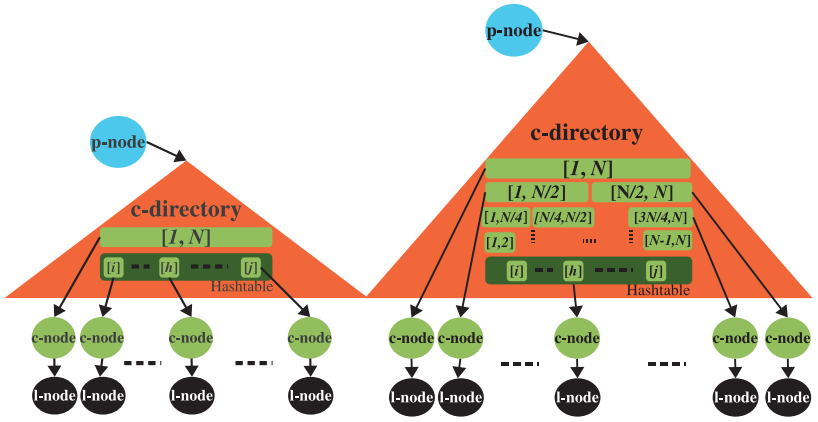
Table I. Operator Transformations

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

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), that is, 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 Section 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 I, 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 $\{v_1, \ldots, v_k\}$ 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.

Specialized Space Clustering Structure. The key observation with regard to the clustering directory is that if the predicate language is restricted to only $(=, \neq, \notin)$ operators, then the clustering directory can be replaced with a hash table and a single bucket that spans the entire domain range, as shown in Figure 3(a). The hash table and the single filtering bucket trivially satisfy the invariance and the cluster directory rules because every bucket in the hash table is a discrete bucket implying that every bucket is always the home bucket for all the expressions that it is hosting. Similarly, the single bucket stores all the expressions that are transformed into an expression



(a) Specialized clustering directory (b) Hybrid clustering directory

Fig. 3. The BE-Tree clustering directories.

that spans the entire domain, also making this bucket a home bucket for all of its expressions. Hence, the space partitioning in the 2-layer hash-based cluster directory also satisfies the BE-Tree invariance. In addition, the 2-layer clustering directory is always two. Hence, the height of BE-Tree is at most $2k$, that is, $O(k)$, where k is the maximum number of predicates in an expression.

Hybrid Space Clustering Structure. In this approach, we combine the idea of the generic and the specialized clustering structures. Therefore, if the expression contains an equality predicate, then it is pushed into a hashtable (specialized structure); otherwise, it is pushed to the generic clustering structure, Figure 3(b).

4.5. BE-Tree Theoretical Analysis

BE-Tree Operations Time Complexity. 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 Section 7.1. In contrast, for insertion, the tree traversal follows exactly one path, which is explained in-depth in Section 7.2. The deletion operation is conceptually similar to the matching operation and presented in Section B of the electronic appendix. Finally, the update operation is a composite operation consisting of a deletion followed by an insertion.

We also prove the time complexity of BE-Tree operations in Section E of the electronic appendix. The summary of these results is presented as follows.

THEOREM 4.7. *The cost of* BE-Tree *insertion is bounded by* $O(k \log N)$.

THEOREM 4.8. *The* BE-Tree *matching (searching) algorithm is a multipath traversal operation.*

THEOREM 4.9. *The deletion cost of augmented*[3] BE-Tree *is bounded by* $O(k \log N)$.

THEOREM 4.10. *The cost of* BE-Tree *update is bounded by* $O(k \log N)$.

BE-Tree Space 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 [Aguilera et al. 1999], does not grow in the number of subscriptions

[3]For detailed description of required BE-Tree augmentation please refer to Section E of the electronic appendix.

avoiding memory- and performance-detrimental tree degeneration; the complete proof is presented in Section D of the electronic appendix.

THEOREM 4.11. *The height of* BE-Tree *is bounded by* $O(k \log N)$.

5. BE-TREE SELF-ADJUSTMENT

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

5.1. Cost-Based Ranking Function

BE-Tree's ranking objective 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 [Fabret et al. 2001], which is a deviation from the actual index objective. Our BE-Tree's ranking objective, which directly reduces the matching cost, is formulated based on the notion of *false candidates*.

Definition 5.1. *False candidates* are the expressions (subscriptions) retrieved that are not matched by an input expression (event).

We formalize BE-Tree's ranking 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 the 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, that is, $\forall x \in \texttt{Dom}(P_i^{\texttt{attr}})\ P_i(x) \leftrightarrow P_l(x)$, 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 subsumed and covered predicates for each of its expressions. Formally, $Rank(n_i)$, $Loss(n_i)$, and $Gain(l_j)$ are defined as follows:

$$Rank(n_i) = \begin{cases} (1-\alpha)Gain(n_i) - \alpha Loss(n_i) & \text{if } n_i \text{ is a } l\text{-node} \\ \left(\sum_{n_j \in des(n_i)} Rank(n_j)\right) - \alpha Loss(n_i) & \text{otherwise,} \end{cases} \tag{10}$$

where $0 \leq \alpha \leq 1$.

$$Loss(n_i) = \sum_{e' \in window_m(n_i)} \frac{\#\ \texttt{discarded pred eval for}\ e'}{|window_m(n_i)|}. \tag{11}$$

$$Gain(l_j) \ = \ (1-\beta)Gain_s(l_j) + \beta Gain_c(l_j),\ 0 \leq \beta \leq 1. \tag{12}$$

$$Gain_s(l_j) \ = \ \#\ \texttt{subsumed pred},\quad Gain_c(l_j) = \#\ \texttt{covered pred}. \tag{13}$$

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

$$Rank(l_i) = (1-\alpha)Gain(l_i) - \alpha Loss(l_i), \quad where\ 0 \leq \alpha \leq 1, \tag{14}$$

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 empirical evidence, this cost model can be further simplified by setting both parameters α and β to 0.5; hence, giving equal weight to the $Gain()$ and $Loss()$ functions and to the $Gain_s()$ and $Gain_c()$ functions. This simplification is possible because in the majority of our synthetic and real experiments the rate of false candidates was sufficiently low such that altering the values of these parameters had negligible influence on the overall BE-Tree matching rate. However, in certain special settings, these two parameters could be invaluable, namely, to deal with fluctuations in event stream or to model the effectiveness of subsumed vs. covered predicates.

In general, the parameter α can play an important role in order to smoothen sudden spikes in the event stream or to adapt to a stream with high fluctuation rates. A low value of α places smaller weight on recent changes captured by our $Loss()$ function and puts heavier weight on the subscription workload characteristics that are captured by our $Gain()$ function. In contrast, a high value of α, places larger weight on an event stream and adapts to stream fluctuations. Alternatively, one can view the role of α as adapting to either event workload (a high value of α) or subscription workload (a low value of α.)

Our second parameter β is used to establish the effectiveness of subsumed and covered predicates while observing the event stream. Initially, it is assumed that both subsumed and covered predicates are equally effective (i.e., $\beta = 0.5$), meaning that a covered predicate has a low rate of generating false candidates. However, if during the matching process, it is observed that a covered predicate has a much higher rate of generating false candidates, then the parameter β is tuned accordingly. For instance, if on average, a covered predicate generates a false candidate at a rate of 80%, then β is set to 0.1 in order to favor subsumed predicates accordingly.

$$\beta = 0.5 - \frac{\phi}{2}, \tag{15}$$

where ϕ is the observed rate of false candidates for covered predicates. The effectiveness (and dynamic tunning) of the β parameter is studied in Section 8.6.12.

5.2. Adaptation Policies

In BE-Tree, three strategies are considered to utilize the cost-based ranking function and to further avoid tree degeneration: recycling l-nodes, reinserting Boolean expressions, and exploration vs. exploitation.

Recycling l-nodes is BE-Tree's main self-adjusting policy that monitors each node over a frequency window, $\texttt{freq}_{\texttt{window}}$, of the number of insertion, deletion, and matching operations. If at the end of each $\texttt{freq}_{\texttt{window}}$ interval for each node, 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 reinserted into BE-Tree.

Reinserting Boolean Expression is exercised prior to invoking the space clustering and the space partitioning. Reinsertion is geared towards adapting BE-Tree to changes in the subscription workload. This policy targets leaf nodes by randomly selecting a subset of a leaf node content, driven by the $\texttt{ratio}_{ins}$ parameter, and it reinserts the selected elements.

Exploration vs. Exploitation is a self-adjusting policy that is triggered when a new expression is inserted in BE-Tree. At every level of BE-Tree, an attribute from the expression is selected such that with probability $\texttt{ratio}_{exp}$ the selected attribute is ranked the highest in the current level (exploitation) otherwise the selected attribute is randomly chosen (exploration). This active policy enables exploring the entire space while benefiting from existing statistics and avoids making decisions based only on past selections.

6. BE-TREE OPTIMIZATIONS

In this section, we investigate BE-Tree's execution model, in order to identify key opportunities to further accelerate the matching computation. First, we present a lazy and a bitmap-based Boolean predicate evaluation. The main focus of the lazy evaluation technique is to ensure exactly-once evaluation of every distinct predicate. On the other hand, the bitmap technique, at the high-level, exploits predicate interrelationships (i.e., predicate covering) and guarantees exactly-once evaluation of distinct predicates, and at the low-level, it minimizes storage through an efficient bitmap representation and speeds up the computation using low-level bitwise operations and preserves cache locality. Second, we introduce a Bloom filtering strategy to minimize false candidate evaluation at BE-Tree's leaf level.

6.1. Lazy and Bitmap-Based Predicate Evaluations

One of the main goals of BE-Tree, in addition to search space pruning, is to minimize the true candidate computations, that is, the evaluation (and the encounter) of common predicates exactly once. BE-Tree's structure and cost-function are designed to attain this objective. We generalize the scope of this objective by also ensuring that each distinct predicate is always evaluated exactly once; however, we generalized this exactly-once objective from a different angle. Conceptually in this new paradigm, as we traverse BE-Tree for a given event, we also maintain an efficient structure (with respect to both time and space) to store the evaluation result (`True` or `False`) of each distinct predicate in our subscription workload. This structure is represented as a bit-array, in which each bit indicates whether or not a distinct predicate has been evaluated to `True` (or `False`). Exploiting a bit-array not only provides fast read/write access to predicate evaluation results, but also its compact representation, as was observed in most workloads in our experiments, can be pinned entirely in modern processor L2 cache, which significantly reduces the number of cache-misses.

To this end, we propose two different techniques for evaluating predicates using a predicate bit-array. Our fist technique, referred to as *lazy predicate evaluation*, takes a passive approach such that before evaluating a predicate, first, it determines whether or not the predicate has already been evaluated for the current event; if not, then the predicate is evaluated, and the corresponding bit in the predicate bit-array structure is flipped to reflect the predicate evaluation result.

To be precise, we actually, maintain two bit-arrays, namely, *Pass Bit-array* and *Fail Bit-array*, to maintain predicates that are evaluated to either `True` or `False`, respectively. At the outset of matching a new event, both bit-arrays are initialized to zero.[4]

[4]It is important to note that bit-array initialization can utilize the information from the last seen event in order to selectively reset only parts of the bit-arrays.

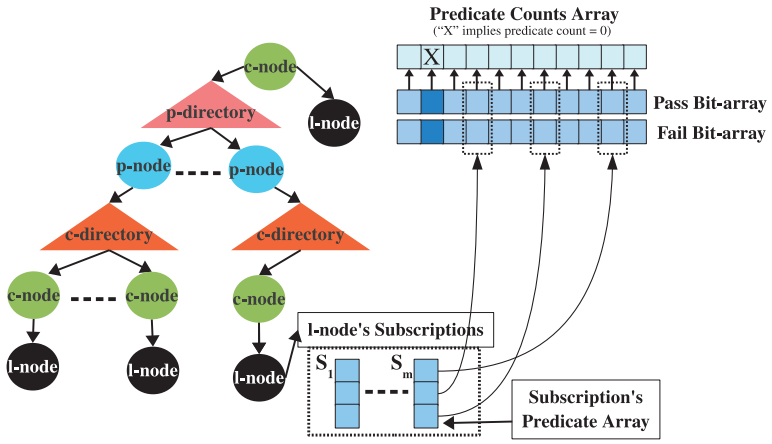


Fig. 4. BE-Tree lazy predicate evaluation technique.

Before evaluating a predicate P, we first check if the bit corresponding to P is 0 in both Pass/Fail Bit-arrays, that is, the predicate P has not been evaluated previously. If the corresponding bit in Pass Bit-array is 1, then it implies that the predicate was evaluated to `True` for the current event; similarly, if the bit in the Fail Bit-array is 1, then we can infer that the predicate P is `False`. Finally, if the predicate P has not been evaluated previously, then P is evaluated, and the result is reflected either in Pass or Fail bit-array accordingly. This procedure yields the following invariance regarding Pass/Fail bit-arrays.

INVARIANCE 1. *For each distinct predicate P, either both bits in Pass/False Bit-arrays are 0 or exactly one bit is set to 1.*

The lazy predicate evaluation technique is depicted in Figure 4, in which each subscription's predicate in BE-Tree's leaf nodes is associated with the predicate bit-arrays. In short, not only the lazy predicate evaluation achieves execution of every predicate exactly once and improves cache-locality but also supports subscription insertions, in which the new subscriptions are permitted to have distinct predicates not seen previously. The new distinct predicates are appended to the end of the bit-arrays. In addition, subscription deletions, which could potentially trigger removal of distinct predicates can be achieved by maintaining an additional supporting structure, *Predicate Count Array*, such that for each distinct predicate P, it stores the number of subscriptions that contain predicate P. Once a predicate becomes an orphan, then the predicate is marked as deleted (as shown in Figure 4). The deleted space is either used to accommodate new distinct predicates or reclaimed periodically through a standard defragmentation (or compacting) procedure. Notably, defragmentation can be done concurrently with event matching, during which the matching solely relies on BE-Tree and does not leverage the lazy predicate evaluation.

Our second predicate evaluation technique, referred to as *bitmap-based predicate evaluation*, pushes the limit of lazy predicate evaluation by also incorporating the predicate interrelationships (e.g., predicate covering) through a novel precomputation and storing of predicate coverings, which is achieved partly due to the exploitation of the discrete and finite domain properties. We propose a bitmap structure over the set of all distinct predicates such that for any given attribute-value pair, essentially an equality predicate $P^{\texttt{attribute},=,\texttt{value}}$, we precompute and store (designed for an efficient read access in mind) in our bitmap index all distinct predicates that are relevant for

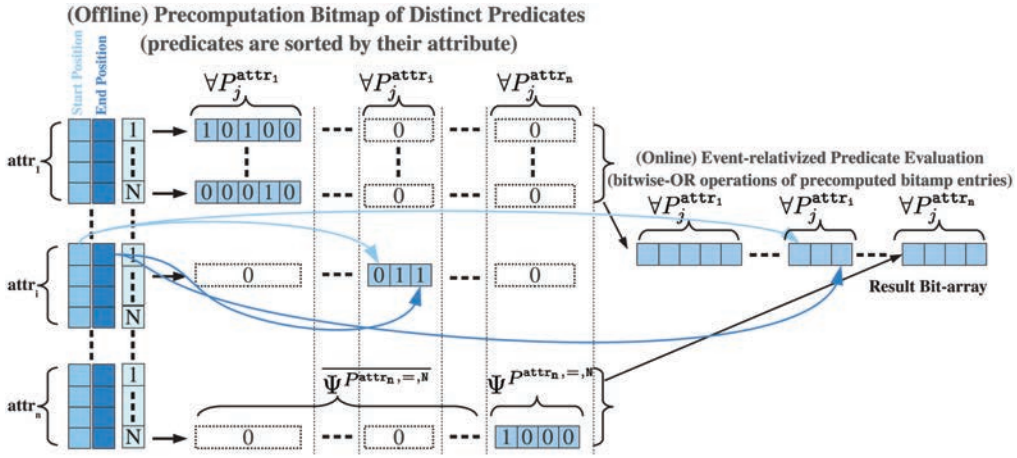


Fig. 5. BE-Tree bitmap-based predicate evaluation technique.

P[5]. Therefore, instead of individually evaluating every relevant distinct predicate for a given event's equality predicate, we precompute the evaluation results of every distinct predicate that is affected by any given event's predicate. The set of affected predicates by the equality predicate P, denoted by Ψ^P, is defined as follows

$$\Psi^P = \left\{P_i \mid \forall P_i^{\texttt{attr},\texttt{opt},\texttt{val}}(x) \in \boldsymbol{\Psi}, P^{\texttt{attr}} = P_i^{\texttt{attr}}, \exists x \in \texttt{Dom}(P^{\texttt{attr}})\right\}, \tag{16}$$

where $\boldsymbol{\Psi}$ is the set of all distinct predicates.

Consequently, the set of all distinct predicates that are not affected by P are given by $\overline{\Psi^P} = \boldsymbol{\Psi} - \Psi^P$. The bitmap is constructed by determining the sets Ψ^P and $\overline{\Psi^P}$ for each P, that is, P is formed by enumerating over the discrete values of each attribute (a dimension in the space) in order to construct an equality predicate P. Next, we evaluate each predicate in Ψ^P for a given P and store the results in the bitmap. Also the set $\overline{\Psi^P}$ is automatically filled with 0 because none of the predicates in the set $\overline{\Psi^P}$ are affected by P. The overall structure of the bitmap and its organization of Ψ^P and $\overline{\Psi^P}$ are illustrated in Figure 5.

The most striking feature of our proposed bitmap is its highly sparse matrix structure because the set of bits represented by $\overline{\Psi^P}$ are all zero, and given the high-dimensionality of our problem space, we have $|\Psi^P| \ll |\overline{\Psi^P}|$. Thus, if the corresponding bits are reordered by clustering Ψ^P and $\overline{\Psi^P}$ (as shown in Figure 5), we can achieve an effective space-reduction in our bitmap. This bitmap space saving ratio $\frac{|\overline{\Psi^P}|}{|\Psi^P|}$, through reordering of the bits in the bitmap, is directly proportional to the number of attributes n if the predicates are sorted based on their attributes and given by

$$n \propto \frac{|\overline{\Psi^P}|}{|\Psi^P|}. \tag{17}$$

This ratio is further influenced by the distribution of predicates over each attribute. For instance, for certain domain values *value* over attribute *attr*, there may be no predicate such that the *value* falls in any predicate's range of values; thereby, implying that

[5]Without the loss of generality, we focus our discussion on events consisting of only equality predicates, which can easily be extended to support events with range predicates as required in our extended matching semantics by enumerating over the domain of range predicates.

the set $\Psi^{P^{\texttt{attr},=,\texttt{x}}}$ consists of only zero bits. Such a distribution of predicates results in further space reduction because neither $\Psi^{P^{\texttt{attr},=,\texttt{x}}}$ nor $\overline{\Psi^{P^{\texttt{attr},=,\texttt{x}}}}$ need to be stored explicitly in the bitmap. This reduction in space also improves, as a byproduct, the bitwise operations during the matching process. In addition, there are potential research opportunities to develop a more effective bit reordering techniques (i.e., a predicate topological sort order) to further improve the space saving ratio. In particular, the minimum number of predicates that must be maintained for each P (a lower-bound on size of set Ψ^P) are those distinct predicates that are satisfied by P. This minimum set is defined as

$$\Psi^P_{\texttt{min}} = \left\{P_i |\ \forall P_i^{\texttt{attr},\texttt{opt},\texttt{val}}(x) \in \boldsymbol{\Psi},\ P^{\texttt{attr}} = P_i^{\texttt{attr}}, \exists x \in \texttt{Dom}(P^{\texttt{attr}}),\ P(x) \wedge P_i(x)\right\}, \quad (18)$$

During the BE-Tree matching process, upon arrival of a new event e, the precomputed bitmap index is utilized to efficiently compute all distinct predicates that are satisfied by the incoming event. This is carried out by a bit-wise OR-operation of relevant rows in our bitmap index in order to fully construct *Result Bit-array*: a bit-array in which each bit corresponds to a distinct predicate, where a bit with value 1 signifies that the corresponding predicate is `True`; otherwise `False`. The Result Bit-array is constructed as follows:

$$\textit{Result Bit-array} = \bigcup_{P_i \in e} \{\Psi^{P_i}, \overline{\Psi^{P_i}}\}, \quad (19)$$

where no actual operation is required to account for $\overline{\Psi^{P_i}}$ sets.

The Result Bit-array can entirely be pinned in cached as long as the subscription workload contains only in order of few millions of distinct predicates, for which only few mega bytes of cache is required[6]. However, the potential source of cache misses is not limited to Result Bit-array accesses, in fact, another BE-Tree's internal data structure that generates cache misses (in addition, to general pointer chasing of any tree structure) is the representation of leaf pages content. In BE-Tree with bitmap-based evaluation since each subscription in the leaf page requires only to keep an array of references to Result Bit-array, then it is feasible to store all subscriptions in the leaf page as cache-conscious block of 2-dimensional array of references. Moreover, since in BE-Tree, the number of subscriptions in each leaf page is limited to only tens or hundreds of subscription (cf. Table VI), then this *2-dimensional subscription representation* could also be fitted in the processor cache; thus, substantially reducing the number of cache misses during subscription evaluations at the leaf level and improving the overall matching time.

In summary, the lazy predicate evaluation technique is most suitable for settings in which many of the expected subscription insertions (or updates) contain unseen distinct predicates while the bitmap-based predicate evaluation technique is ideal for more stable workloads. Therefore, a hybrid mechanism can be adopted in which unseen distinct predicates can be maintained through the lazy predicate evaluation while the stable distinct predicates are maintained through the bitmap technique. Therefore, periodically, the two sets of distinct predicates are merged and the bitmap is reconstructed again. Most importantly, merging the two distinct predicate sets from both the lazy and the bitmap structures can be carried out concurrently as BE-Tree continues to match new incoming events.

6.2. Bloom Filtering Optimization

Our final optimization is designed to reduce the number of false candidates at the BE-Tree's leaf nodes level, which is motivated by a simple observation that subscription s_i is a possible candidate if at the very least, the set of attributes on which s_i has defined

[6]The processor used in our experiment has two shared 6144KB cache block size.

ALGORITHM 1: `MatchBETree`($event$, $cnode$, $matchedSub$)

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

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: else if IsEnclosed(event, cdir.rChild) then
5:    SearchCDir(event, cdir.rChild, matchedSub)
```

a predicate explicitly is a subset of attributes appearing in a given event e; formally expressed as

$$\forall P_q^{\texttt{attr,opt,val}}(x) \in s_i,\ \exists P_o^{\texttt{attr,opt,val}}(x) \in e,\ P_q^{\texttt{attr}} = P_o^{\texttt{attr}}. \tag{20}$$

This necessary condition $\mathcal{C}$ for testing set membership can efficiently be evaluated (approximately) if both the subscription s_i and the event e have a (lossy) compact encoding of all the attributes that appear in s_i and e, respectively. To attain this compact encoding, we encode the set of attributes using a Bloom filter representation as follows

$$s_i{}^{\texttt{bloom}} = \bigvee_{P_q \in s_i} \left(\texttt{sdbm\_hash}\left(P_q^{\texttt{attr}}\right) \bmod \texttt{bloom\_size}\right), \tag{21}$$

where $\vee$ is a bitwise OR-operation, `sdbm_hash` is a hash function, and `bloom_size` is the Bloom filter size. In particular, we utilize the well-known `sdbm` hash function[7], and we experimented with various choice of `bloom_size` ranging from 16-64 bits in order to investigate the false positive rate of our Bloom filter encoding (cf. Section 8). The Bloom filter for the event e is computed in a similar manner.

Therefore, during the matching process, the necessary matching condition is (approximately) satisfied *iff* the bitwise AND-operation ($\wedge$) of the subscription's and the event's Bloom filters are equal to the subscription's Bloom filter.

$$\mathcal{C}(e, s_i) = \texttt{True} \leftrightsquigarrow e^{\texttt{bloom}} \wedge s_i{}^{\texttt{bloom}} = s_i{}^{\texttt{bloom}}. \tag{22}$$

7. BE-TREE IMPLEMENTATION

Next, we provide an in-depth explanation, together with pseudocode, for the two main operations of BE-Tree, namely, matching and insertion. Furthermore, for the ease of presentation, without the loss of generality, we focus on matching with stabbing subscription semantics in which event expressions consist of only equality predicates (=).

7.1. Matching Pseudocode

Event matching consists of two routines: (1) MatchBETree (Algorithm 6.2) which checks subscriptions in a leaf node and traverses through BE-Tree's p-directory and (2) SearchCDir (Algorithm 6.2) which traverses through BE-Tree's c-directory.

[7]This algorithm was created for the `sdbm` (a public-domain reimplementation of `ndbm`) database library. The Bloom filter implementation used is found under http://en.literateprograms.org/Bloom_filter_(C).

ALGORITHM 3: `InsertBETree`(sub, $cnode$, $cdir$)

```
1: {Find attr with max score not yet used for partitioning}
2: if cnode.pdir ≠ NULL then
3:   for i = 1 to NumOfPred(sub) do
4:     if !IsUsed(sub.pred[i]) then
5:       attr ← sub.pred[i].attr
6:       pnode ← SearchPDir(attr, cnode.pdir)
7:       if pnode ≠ NULL then
8:         foundPartition = true;
9:         if maxScore < pnode.score then
10:          maxPnode ← pnode
11:          maxScore ← 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 ← InsertCDir(sub, maxPnode.cdir)
20:   InsertBETree(sub, maxCdir.cnode, maxCdir)
21:   UpdatePartitionScore(maxPnode)
```

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). In order to support more expressive matching semantics and predicate operators only the `IsEnclosed` function (in Algorithm 6.2) algorithm must be changed accordingly.

7.2. Insertion Pseudocode

Unlike matching, insertion is rather involved because it also manages the overall dynamics of BE-Tree, that is, the space partitioning and the space clustering. To insert, BE-Tree's root and a subscription is passed to the InsertBETree (Algorithm 7.2) that attempts to find an l-node with the highest score that encloses the subscription. Essentially 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 the InsertBETree. Subsequently, 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 (Algorithm 7.2). Upon choosing the smallest c-node, the

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
```

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)
```

subscription advances to the next level of BE-Tree, and the routine InsertBETree is recursively called on the new c-node. After the recursive call, the function UpdatePartitionScore (Line 21) is invoked, which implements our proposed cost-based ranking function based on Equation (10) or its simpler form given by Equation (14).

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 BE-Tree's two-phase space-cutting technique: partitioning and clustering.

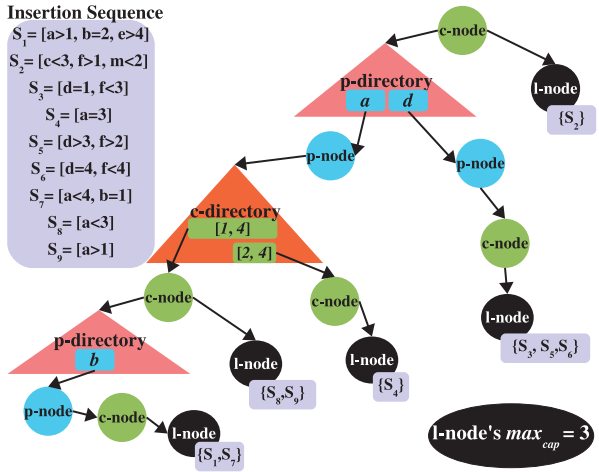


Fig. 6. A concrete example.

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 (Algorithm 7.2) 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. Lastly, by calling the function UpdateClusterCapacity (Line 10), the cluster capacity is updated based on Equation (9).

SpaceClustering (Algorithm 7.2) 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 nonleaf 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). Lastly, by calling the function UpdateClusterCapacity (Line 17), the cluster capacity is updated based on Equation (9).

7.3. Concrete Example

Next we present an example to further elucidate the insertion algorithm. The final BE-Tree is shown in Figure 6.

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 (nonleaf) 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).

8. EVALUATIONS

We present a comprehensive evaluation of BE-Tree using both synthetic and real datasets. The experiments were carried on two Quad-core Intel Xeon X5450 processors running at 3.00 GHz machine with 16GB of memory running CentOS 5.5 64bit. All algorithms are implemented in C and compiled using gcc 4.1.2 with optimizations set to $O3$.

8.1. Experiment Overview

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. In addition, we discuss the effectiveness of sequential BE-Tree in comparison of state-of-the-art GPU parallel matching algorithm in Section F of the electronic appendix.

We also ran experiments to determine optimal choices for BE-Tree internal parameters, cf. Section 8.5. The values used throughout our experiments are: $\texttt{max}_{cap}$ ranging from 5-160, $\texttt{min}_{size} = 3$, $\texttt{rank}_{\texttt{window}}$ for each node is 10% of expressions in the node's subtree, and $\theta = 0$.

Once we establish the effectiveness of BE-Tree with respect to state-of-the-art approaches, we shift our focus in the experimental evaluation towards further improving BE-Tree through lazy and bitmap-based predicate evaluations and the Bloom filter optimization.

8.2. Datasets

8.2.1. 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 workload generation framework, *BEGen*[8] 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

$$\forall P_q(x) \in e_i,\ \exists P_o(x) \in e_B,\ P_q^{\texttt{attr}} = P_o^{\texttt{attr}}, \forall x\ P_o(x) \rightarrow P_q(x). \tag{23}$$

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 II. 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

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

Table II. Inferred Predicates from $P_j^{(i,=,v_*)}$

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

Table III. Experiment Settings for Synthetic Datasets

	Workload Size	Number of Dimensions	Dimension Cardinality	Predicate Selectivity	Dimension Selectivity	Sub/Event Size	% Equality Predicate	Matching Probability	Extended Matching Semantics	Bloom Filter
Size	100K-1M	1M	100K	100K	100K	100K	1M	1M	100K-1M	100K-1M
Number of Dim	400	50-1400	400	400	400	400	400	400	400	400
Cardinality	48	48	48-150K	48	2-10	48	48	48	48	48
Avg. Sub Size	7	7	7	7	7	5-66	7	7	7	5
Avg. Event Size	15	15	15	15	15	13-81	15	15	15	7
Pred 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	0.3	0.3
Op Class	Med	Med	Med	Med	Min	Med	Med	Lo-Hi	Med	Med
Match Prob. %	1	1	1	1	—	1	1	0.01-50	—	—

Table IV. Levels of Expressiveness

OpClass	Types of Operators
Minimum	$(=)$
Low	$(=, \in)$
Medium	$(<, \leq, =, \geq, >, \in,$ BETWEEN$)$
High	$(<, \leq, =, \neq, \geq, >, \in, \notin,$ BETWEEN$)$

subscriptions is y, we choose x and y such that $\binom{x}{y}$ is large enough such that duplicate subscriptions are unlikely. Also, the probability $\Pr_{trans}$ further injects variations into the subscription workload. It is controlled by the ratio of equality vs. nonequality predicates within each workload. 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 III, in which each column corresponds to a different workload profile while each row corresponds to the actual value of the workload parameters. Lastly, Table IV captures our four levels of operator expressiveness.

8.2.2. Real Dataset. In the absence of a standard benchmark for evaluating matching algorithm with real data as part of the *BEGen* framework, we propose a generation of real workloads from extracted public domain data. We focus on the data extracted from the DBLP repository,[9] which is also commonly used as benchmark in assessing

[9]Bibliographic information on major computer science publications.

Table V. Experiment Settings for Real Datasets

	DBLP Author	DBLP Title	Matching Probability (Author)	Matching Probability (Title)	Extended Matching Semantics (Author)	Extended Matching Semantics (Title)
Size	100–760K	50–250K	400k	150	100–760K	50–250K
Number Dim	677	677	677	677	677	677
Cardinality	26	26	26	26	26	26
Avg. Sub Size	8	35	8	30	8	30
Avg. Event Size	8	35	16	43	8	30
Pred Range Size %	—	—	12	12	12	12
% Equality Pred	—	—	0.3	0.3	0.3	0.3
Op Class	Min	Min	Lo-Hi	Lo-Hi	Med	Med
Match Prob. %	—	—	0.01–50	0.01–50	—	—

algorithms used in the data quality community. In particular, we use the proceeding titles and author names as two sources of data extracted from DBLP.

We, first, use a deduplication 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 transformation to convert each string from a collection of q-grams into a Boolean expressions. 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']. This 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 in the synthetic datasets. Table V summarizes various workloads generated using the real datasets.

8.3. Matching Algorithms

The algorithms in our comparison studies are (1) SCAN (a sequential scan of the subscriptions), (2) SIFT [10] (the counting algorithm [Yan and García-Molina 1994] enhanced with the enumeration technique [Whang et al. 2009] to support range operators), (3) k-ind (the *CNF* algorithm implemented over k-index [Whang et al. 2009]), (4) GR (the Gryphon algorithm [Aguilera et al. 1999]), (5) ADGR (our Advanced Gryphon algorithm [Aguilera et al. 1999]) which is constructed after applying our operator transformation given in Table I, (6) P (the Propagation algorithm [Fabret et al. 2001]), (7) APP (the Access Predicate Pruning algorithm [Farroukh et al. 2011] also enhanced with the enumeration technique [Whang et al. 2009]), (8a) BE (our fully dynamic version of BE-Tree in which the index is constructed by individually inserting each subscription, and (8b) 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 the dynamic BE-Tree, we have constructed k-index, the two versions of Gryphon algorithms (ADGR and GR), and Propagation using a static workload in which all subscriptions are known in advance.

[10]This counting algorithm is also employed in [Cugola and Margara 2012; Margara and Cugola 2013] for a fast GPU implementation.

In addition, we have implemented the specialized GR algorithm that supports only equality predicates and the generic GR that supports arbitrary predicates [Aguilera et al. 1999]; we have included all the Gryphon optimizations as well [Aguilera et al. 1999], that is, collapsing *-edges (do not care edges) and leveraging predicate covering proposed in Aguilera et al. [1999] to build the Gryphon data structure. Moreover, after applying our proposed predicate transformation in Table I, the predicate covering in Aguilera et al. [1999] becomes substantially more effective as demonstrated in our experimental evaluations.

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, APP [Farroukh et al. 2011], SIFT [Yan and García-Molina 1994], and k-index [Whang et al. 2009] 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 [Whang et al. 2009]. 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.

8.4. Experiment Organization

In our micro experiments, we study the BE-Tree's internal 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.

In our experiment, the main metric that distinguishes between various matching algorithms is the matching time (i.e., matching response time). In particular, we compare the matching time of BE-Tree with alternative algorithms A based on the following ratio:

$$\% = \frac{\mathcal{M}^{\mathtt{A}} - \mathcal{M}^{\mathtt{BE\text{-}Tree}}}{\mathcal{M}^{\mathtt{A}}} \tag{24}$$

where $\mathcal{M}^{\mathtt{A}}$ is the matching time of algorithm A. Therefore, in our discussion, we use this ratio for comparison; for instance, BE-Tree improves over algorithm A by % or BE-Tree reduces the matching time by %.

8.5. Micro Experiments

The most important parameter of BE-Tree is the maximum l-node capacity size, $\mathtt{max}_{cap}$, which triggers our two-phase space-cutting technique. In Figure 7 (and in Section F of the electronic appendix), for various workloads with different matching probability, the effect of varying the 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 VI, which we use as a guiding principle throughout our evaluation for choosing the optimal value with respect to the degree of matching

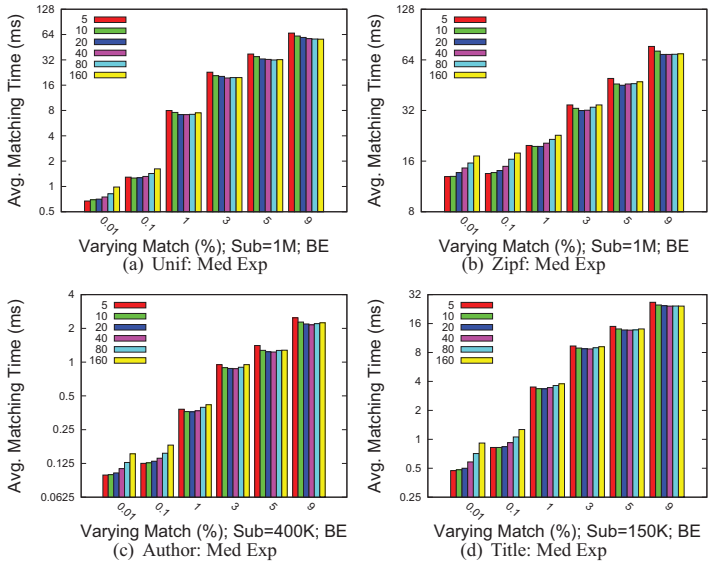


Fig. 7. Matching probability with different l-node capacities.

Table VI. Most Effective $\texttt{max}_{cap}$ for Different Matching Probabilities

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

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.

Other notable BE-Tree parameters are the base scoring function, the type of clustering directory, and reinsertion. As we described in Section 5, for the base scoring function, we used the optimistic popularity measure which resulted in a superior performance in terms of construction time of BE-Tree while reducing the matching computation by further exploiting commonality among subscriptions, the effect of the scoring function is illustrated in Figure 8. Moreover, we considered three strategies for choosing the clustering directory: *hybrid clustering directory* (*Hy*), *hybrid clustering directory* optimized with equality predicate ranking (*Hy-R*), and *generic clustering directory* (*Gen*). Our ranking optimization is an enforcement policy such that all subscriptions' equality predicates are first consumed by our two-phase space-cutting technique before consuming subscriptions' nonequality predicates. The rational behind this improvement is twofold: (1) motivated by our empirical evidence, for instance, *Hy-R* improved *Hy* and *Gen* by up to 52% for the Zipf dataset (Figure 9(b) and improved Hy and Gen by up to 72% for the title dataset (Figure 9(d) and (2) justified by utilizing fast hash access specifically designed for equality predicates in our *hybrid clustering directory*. As a result, for all experiments, we used a *hybrid clustering directory* with equality ranking except for the experiments with expressive events in which a *generic clustering directory* has proven effective. Lastly, we used the adaptive policy and reinsertion policy only for our dynamic experiment in which the reinsertion rate was 5%.

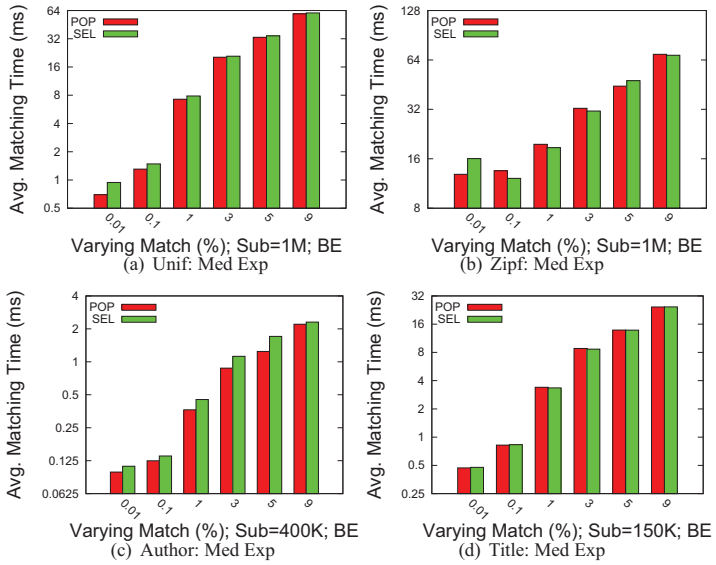


Fig. 8. Matching probability with different scoring functions.

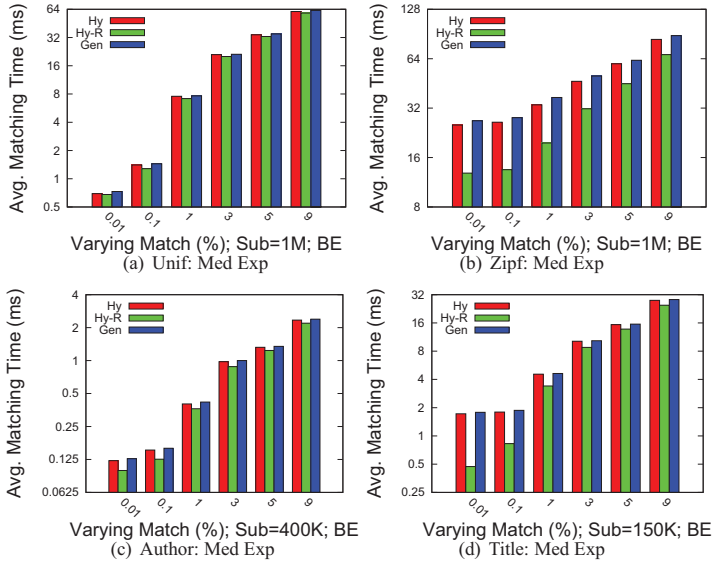


Fig. 9. Matching probability with different clustering directory types.

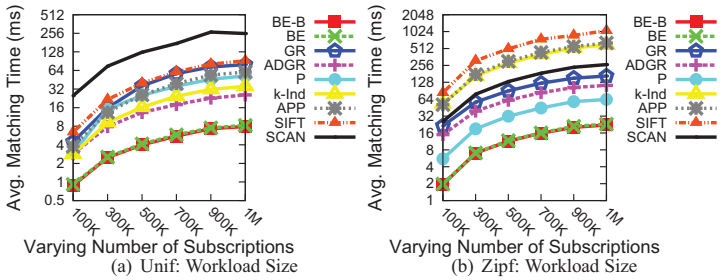
As part of our micro experiments, we also evaluate BE-Tree's construction time. In particular, the construction time of the dynamic BE-Tree, for the largest workload of up to one million subscriptions, was under 5 seconds in our experiments. BE-Tree's average construction time and index size, for the representative datasets in our framework, are summarized in Table VII.

8.6. Macro Experiments

In this section, we compare BE-Tree with several popular matching algorithms over a variety of controlled experimental conditions: workload distribution, workload size,

Table VII. BE-Tree Construction Time & Index Size

Data Sets	Construction Time (second)	Index Size (MB)
Unif (1M)	2.37	68
Zipf (1M)	2.03	67
Author (760K)	3.24	139
Title (250K)	2.18	69



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

Fig. 10. Effect of workload size.

space dimensionality, average subscription and event size, dimension cardinality, predicate selectivity, dimension selectivity, subscription expressiveness, and event matching probability.

8.6.1. 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. *k*-index, APP, 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 Figures 10–17, in which the graphs on the left column correspond to a uniform distribution while the graphs on the right column correspond to a Zipf distribution. The general trend is that under uniform distribution BE-Tree, ADGR, *k*-index, Propagation, APP, GR, and SIFT all outperform SCAN that benefits only from sequential memory access. However, under Zipf distribution both *k*-index, APP, and SIFT perform much worse than SCAN. 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 (APP and SIFT)[11] and scan through *k*-index hashtable buckets (refereed to as the posting list in [Whang et al. 2009]). 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.

8.6.2. Effect of Workload Size. Next, we consider the matching time as we increase the number of subscriptions processed. Figure 10(a)–10(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.

8.6.3. Effect of Dimensionality. Unlike the workload size, the effect of space dimensionality is more subtle; all algorithms with exception of *k*-index, APP, and SIFT are essentially

[11]However, due to the access pruning method employed in APP, APP outperforms SIFT in all experiments.

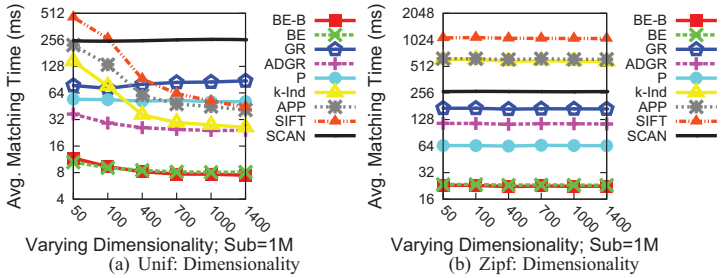


Fig. 11. Effect of dimensionality.

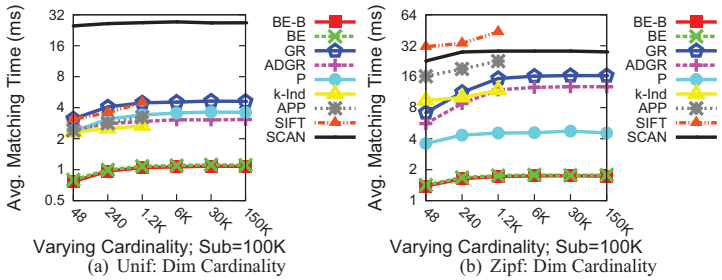


Fig. 12. Effects of dimension cardinality.

unaffected as the dimensionality varies, see Figure 11(a)–11(b). The non-key-based algorithms, namely, k-index, APP, and SIFT, substantially suffer in lower dimensionality for the uniform workload in which subscriptions tend to share many common predicates, which results in high overlap among subscriptions. Therefore, k-index, APP, 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 Zipf distribution, see Figure 11(b), k-index, APP, and SIFT matching time does not improve as the dimensionality increases because of the existence of few popular dimensions which results in a large overlap among subscriptions. Therefore, for the Zipf workload, BE-Tree improves over k-index by 97%.

8.6.4. 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, for instance, BE-Tree and Propagation (Figure 12). 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, Figure 12(a)–12(b), respectively.

However, unlike in BE-Tree, the memory footprint of k-index, APP, 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, APP, 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

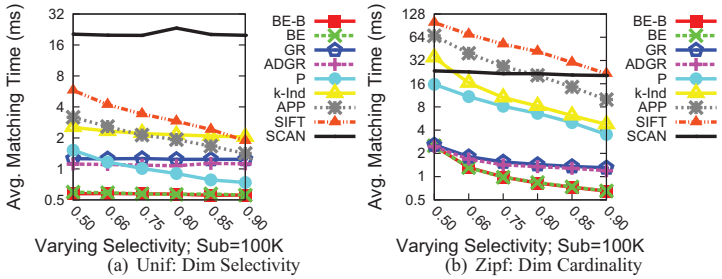


Fig. 13. Effect of dimension selectivity.

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

8.6.5. Effect of Dimension Selectivity. A notable workload characteristic is the dimension selectivity of the space, which could have a direct influence on the the ability of matching algorithm to effectively prune the search space. The result of our dimension selectivity is captured in Figure 13. For a uniform workload the robustness of BE-Tree and the two versions of 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 up to 80% and 27% is observed, respectively.

The importance of dimension selectivity is further magnified for the Zipf workload (Figure 13(b) in which a few dimensions are dominant. Therefore, for a low selectivity, a substantial overhead incurred due to a larger number of false candidates; for instance, as we decreased selectivity from 0.9 to 0.5, the response time is increased by 411% and 650% for Propagation and *k*-index, respectively. In general, the low selectivity results in a less effective pruning of the search space, and BE-Tree compensate for the low selectivity side effect by a deeper tree structure that provides a greater opportunity to prune the search space.

8.6.6. 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, APP, and SIFT, which do not utilize the predicate selectivity information. Therefore, as shown in Figure 14, 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.

In general, a low selective predicate causes a less effective pruning of the search space, and BE-Tree compensates 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. Figure 14.)

8.6.7. Effect of Subscription/Event Size. Another key workload characteristic is the average number of predicates per subscription and event. We analyzed the effect of subscription and event size with respect to three different workload characteristics: varying both subscription and event size (Figures 15(a)–15(b), varying subscription size while fixing event size (Figures 15(c)–15(d), and, varying event size while fixing the subscription size (Figures 15(e)–15(f).

In particular, *k*-index, APP, and SIFT are highly sensitive to the number of predicates: in addition to increasing the overlap among subscriptions, for *k*-index, it also translates

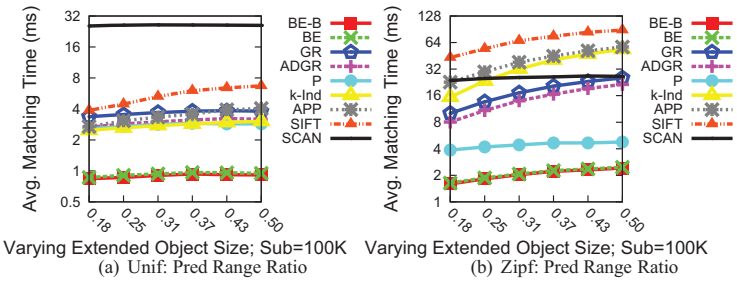


Fig. 14. Effect of predicate selectivity.

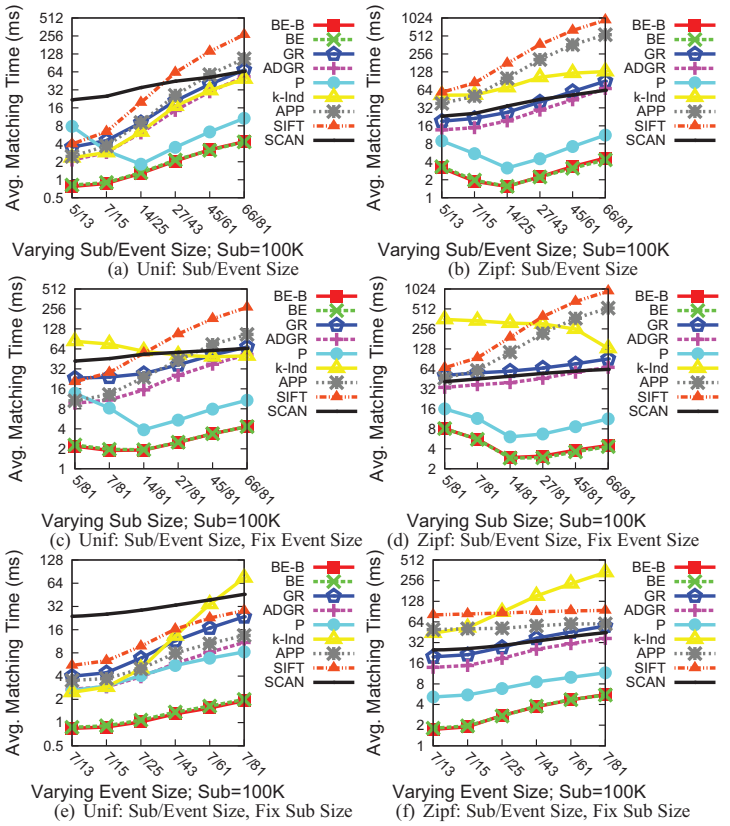


Fig. 15. Effect of subscription/event size.

into a longer sorting time, and for APP and SIFT, it translates to a larger number of retrieving and scanning hashtable buckets when considering increasing both subscription and event size, Figures 15(a)–15(b). 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

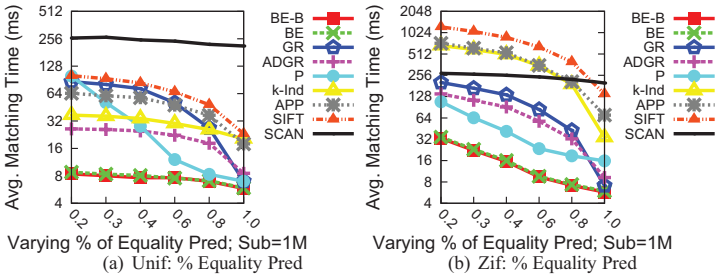


Fig. 16. Effect of percentage of equality predicates.

general, BE-Tree gracefully scales as the number of predicates increases because of its multilayer structure and improves over the next best algorithm by 63% for the uniform and by 65% for the Zipf workload, as illustrated in Figures 15(a)–15(b).

When we vary the subscription size while fixing the event size (Figures 15(c)–15(d)), the key observation is that the increase in the number of subscription's predicates has less impact on matching time as opposed to the increase in the number of event's predicates. The cost associated with increasing the number of subscription's predicates is evident because a longer time is required to check a larger number of predicates. As shown in Figure 15(c)–15(d), varying the number of subscription/event predicates from 5/81 to 5/81 results only in linear increase in matching time. In fact, for k-ind, under Zipf distribution, the matching time is even slightly reduced as the number of subscription increases because it provides a better opportunity to prune the search space. Remarkably, k-ind reaches its worse performance when the number of event's predicates is increased, when moving from Figures 15(a) to 15(c), BE-Tree improves k-ind by 72% and BE-Tree improves k-ind by 98%, respectively. This observation is more striking in Figure 15(e)–15(f), which is discussed next.

In Figure 15(e)–15(f), when increasing the number of event predicates from 13 to 81 k-ind matching time is increased by 42.6 times while BE-Tree, SCAN, P, and APP increased by less than 3 times. In conclusion, BE-Tree was dominant throughout all experiments as we varied subscription and event sizes.

8.6.8. Effect of Percentage of Equality Predicates. In this experiment, we study the effects of ratio of equality vs. nonequality predicates for each subscription. The general trend is that the matching time for all algorithms improve as the percentage of subscription equality predicates increases because the overlap among subscriptions is reduced, see Figure 16(a)–16(b). Most notably, when subscriptions consist only of equality predicates, the specialized GR (or our proposed ADGR) (for equality predicates) results in a substantial performance gain, being the best algorithm after BE-Tree, compared with the generic GR. 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, GR and Propagation (but not ADGR) are the most sensitive algorithm with respect to the percentage of equality predicates; as the percentage of equality predicate decreases, their performance substantially deteriorates. For instance, for the uniform workload, Figure 16(a), BE-Tree improves over GR matching time by 88% when subscriptions only have few equality predicates and by 20% when subscriptions restricted to only equality predicates.

8.6.9. 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

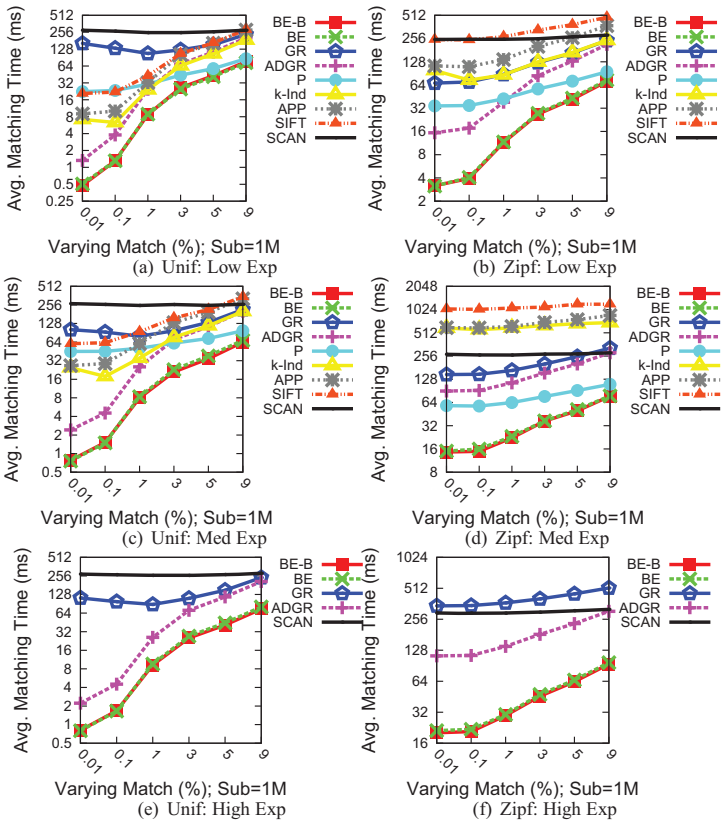


Fig. 17. Effect of percentage of matching probability.

uniform workload with low and medium expressiveness, while keeping the matching probability below 1%, k-index outperforms Propagation, APP, 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 k-index, APP, and SIFT while BE-Tree improves over ADGR and Propagation by up to 33%, even as the matching probability reaches 9%, see Figure 17(a)–17(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, as shown in Section F of the electronic appendix.

In general, an increase in matching probability results in an increase in the number of candidate matches; therefore, APP and 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, see Figure 17(b)–17(d). Furthermore, in the experiments where the highest level of expressiveness was used, BE-Tree dominates both versions of Gryphon algorithms and SCAN by orders of magnitude, see Figure 17(e)–17(f).

8.6.10. Real Datasets Experiments. In the evaluation over real datasets, extracted from DBLP, we first considered varying the workload size without controlling the matching probability and the predicate expressiveness. Therefore, subscription and event workloads were constructed by a direct translation from string data to q-gram and

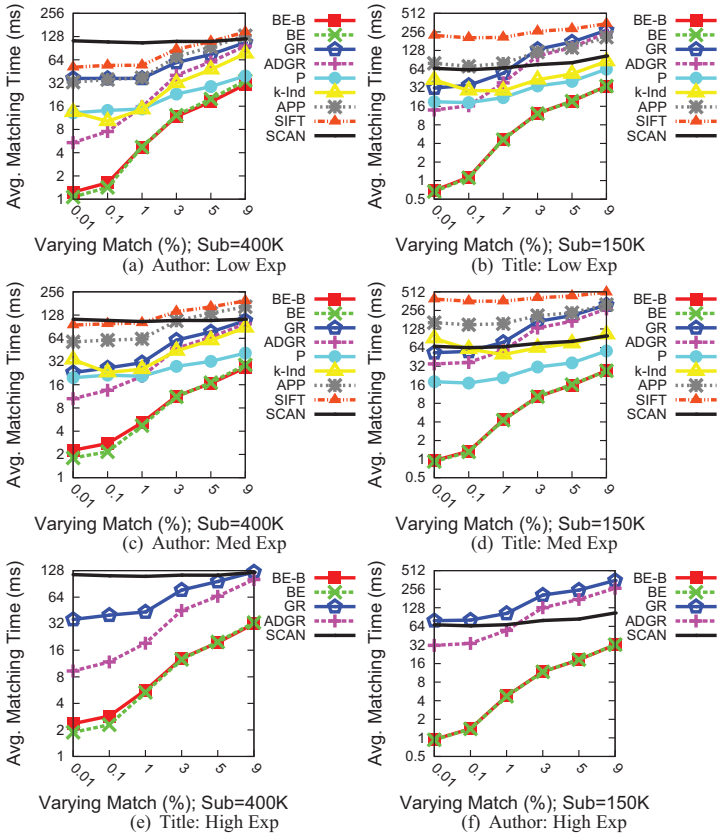


Fig. 18. Effect of percentage of matching probability (DBLP).

ultimately into a conjunction of equality predicates. Second, we considered the effects of changing the matching probability followed by event expressiveness.

In our synthetic experiments in which the percentage of equality predicates is varied, BE-Tree and the two versions of Gryphon algorithms are the top performing algorithms followed by Propagation and k-index. Similar trends were also observed for real datasets. In particular, for the author dataset, Figure 19(a), with an average of 8 predicates per subscriptions, BE-Tree improves over GR by 37% while more 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 BE-Tree's scoring that exploits interrelationships within dimensions and the multilayer structure of BE-Tree that effectively utilizes most of the subscription predicates to reduce the search space. Therefore, as demonstrated in Figure 19(b), BE-Tree improves over GR by 51% and significantly improves over Propagation by more than 99%, up to three orders of magnitude. Furthermore, we have conducted the matching probability experiments with varying degree of expressiveness which produced similar results as in our synthetic dataset, which is shown in Figure 18. Moreover, for the author dataset, as the matching probability goes beyond 50%, SCAN becomes marginally the better algorithm followed by BE-Tree and Propagation as shown in Section F of the electronic appendix. However, for the title dataset, even as the matching probability reaches 50%, BE-Tree

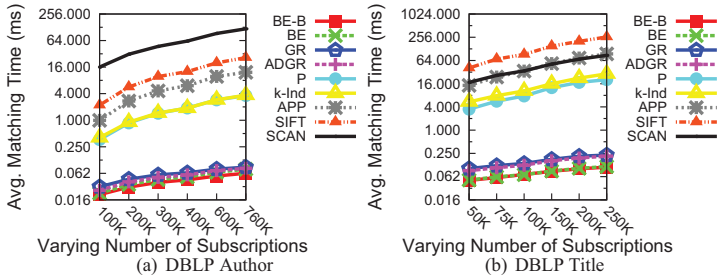


Fig. 19. Effect of real workload size.

outperforms both SCAN and Propagation by nearly 34%, as shown in Section F of the electronic appendix. This performance gain is due to a larger average expression size in the title dataset, which increases BE-Tree's benefits by pruning the search space.

8.6.11. Effect of Event Expressiveness. One of the distinct feature of our matching semantics and BE-Tree is to support an expressive event language because our proposed semantics does not differentiate between event and subscription and models both as Boolean expressions. Hitherto, we focus on evaluating stabbing subscription model in which event expressions were limited to only equality predicate, a model which is adopted by most prior work. Therefore, next, we shift the focus to stabbing subscription model in which the event is no longer limited to only equality predicate.

Notably, the only other relevant matching algorithm that naturally supports expressive event expression is SCAN. In addition, the Propagation can internally be augmented in order to support a more expressive event expression, but such augmentation is nontrivial especially for non-key-based counting approaches such as Gryphon, k-index, APP, and SIFT. Our proposed augmentation for Propagation is as follows: for each event, nonequality predicates are treated as as a set of equality predicates by enumerating over the predicate permitted range of values. For instance, during the runtime the predicate $[attr_i$ BETWEEN $[a, b]]$ is replace by

$$[attr_i = a, attr_i = a + 1, \ldots, attr_i = b]. \tag{25}$$

Therefore, we evaluated BE-Tree, Propagation, SCAN under this new paradigm while varying the workload size and the distributions using both synthetic and real datasets. The experimental results are demonstrated in Figure 20. As expected, BE-Tree continues to outperform both Propagation and SCAN, and the gap between BE-Tree and Propagation is further widen because of the natural support of BE-Tree for expressive event expressions through its two-phase space-cutting technique, and, in particular, its *generic clustering directory* structure. For the synthetic datasets, under uniform workload, BE-Tree improves over Propagation by up to 98% (Figure 20(a)) and under Zipf workload BE-Tree remains superior and improves over Propagation by up to 56% (Figure 20(b)). The smaller gap under Zipf workload is due to the nature of the workload in which not only few dimensions are dominant, but also due to allowing nonequality predicate in event expression, for those dominant dimensions, an event expression spans a large range of domain values. Consequently, there is a higher degree of matching probability, which is inherently harder to control, under the Zipf workload compared to the uniform workload. Similarly, for the real datasets, BE-Tree also significantly outperforms Propagation by 89% (Figure 20(c)) for the author dataset and by 96% (Figure 20(d)) for the title dataset.

8.6.12. Adapting to Subscription/Event Changes. In our adaptive experiments, we studied the self-adjusting mechanism and the maintenance cost (e.g., insertion, deletion,

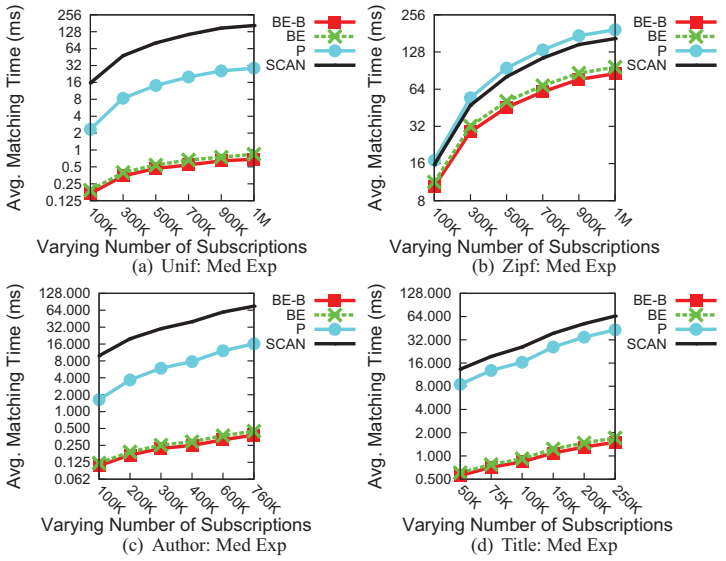


Fig. 20. Effect of events expressiveness.

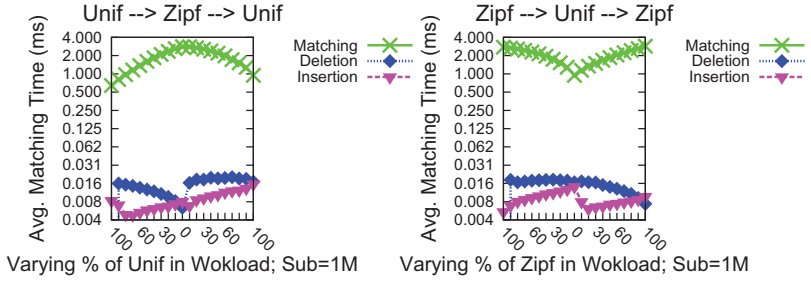


Fig. 21. Effect of dynamic workload.

update) 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, Figure 21(a). A similar adaptation was observed when we transitioned from Zipf to uniform and back again, Figure 21(b). Another important observation from these experiments is that matching cost, as expected, is the dominant compared to insertion and deletion cost. Also in BE-Tree, the update operation is simply implemented as a deletion followed by an insertion.

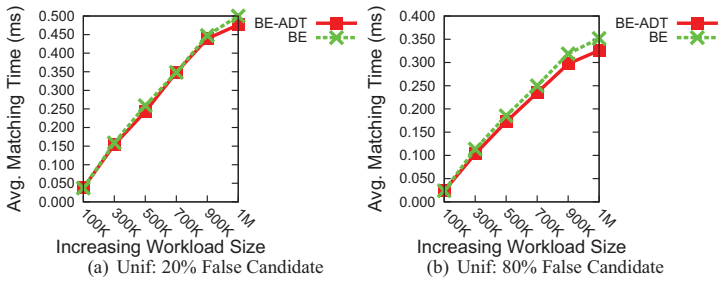


Fig. 22. Effect of dynamic workload (predicate false candidate rate).

Table VIII. BE-Tree/Bitmap Construction Time (second) & Memory Usage (MB)

		BE-Tree (batch)		Bitmap	
Data Sets	Number of Distinct Predicates	Construction Time	Index Size	Construction Time	Bitmap Size
Unif (1M)	519607	26.67	60	23.01	3.2
Zipf (1M)	266132	3.75	57	97.92	1.7
Author (760K)	10810	4.47	98	0.97	0.2
Title (250K)	11566	2.29	37	1.23	0.2

The above experiments mostly focused on adaptation to the drastic changes of the subscription workload. Next we consider the BE-Tree's adaptation with respect to event workload by utilizing BE-Tree's cost function. In these experiments, we consider two types of event workload: (1) an event stream that results in generating up to 20% false candidate through subscriptions covered predicates (Figure 22(a)) and (2) an event stream that generates up to 80% false candidate (Figure 22(b)). For both experiments, an identical subscription workload is used, and subscriptions are inserted incrementally in 100k batches followed by reexecuting the entire event stream over it. We perform the event matching using two BE-Tree variations: BE-ADT in which β in our cost function is tuned dynamically as described in Section 5, and BE in which β is set to 0.5 and is fixed. We can observe that BE-ADT can detect the high rate of false candidate generated by covered predicates; therefore, when BE-Tree is undergoing the two-phase space partitioning for the newly inserted subscriptions, it attempts to utilize subsumed predicate and avoid using covered predicates. Consequently, as more subscriptions are inserted the gap between BE-ADT and BE widens, and a matching time reduction of up to 10% is obtained (Figure 22(b)).

8.7. BE-Tree Optimization Evaluation

In this section, we study the effectiveness of key BE-Tree optimizations including bitmap evaluation and Bloom filter pruning.

8.7.1. Lazy and Bitmap-based Predicate Optimizations. We begin by investigating BE-Tree (batch version) and bitmap construction time and memory usage that are summarized in Table VIII. First, we observe that the construction time of BE-Tree with batch processing is longer compared to dynamic BE-Tree (which was shown in Table VII) because all subscriptions are known in advance; thus, statistics gathering and computations take longer, which in turn results in improved matching time. Second, for the Zipf distribution, subscriptions are densely distributed, that is, many common and overlapping predicates, which result in few dense regions of space associated to a large number predicates. On the one hand, these densely distributed regions result in a higher bitmap construction time while, on the other hand, the fewer number of distinct predicates that are densely distributed across the attribute space result in a

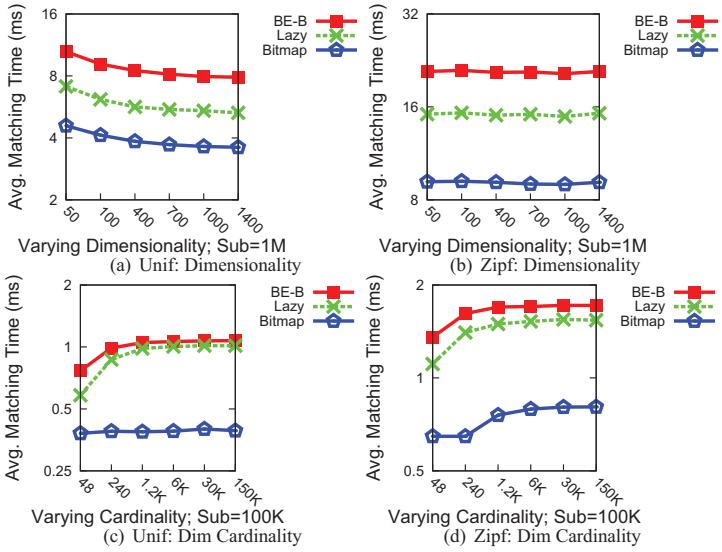


Fig. 23. Effects of dimensionality/dimension cardinality on lazy/bitmap optimizations.

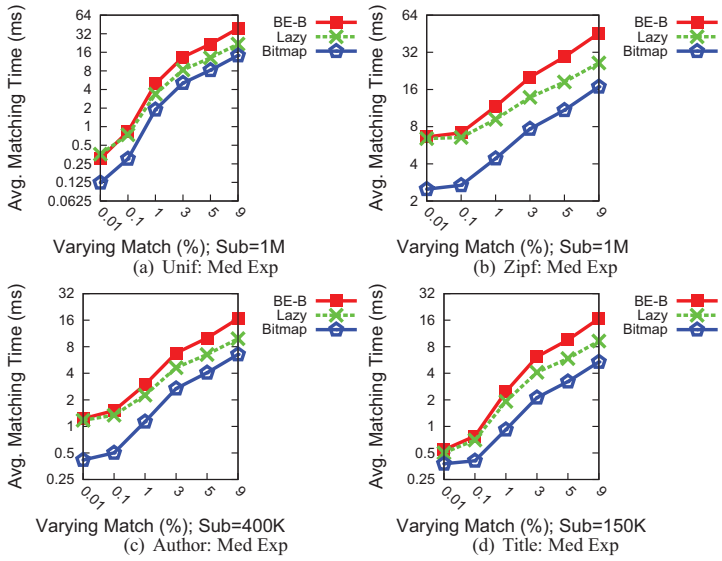


Fig. 24. Effect of percentage of matching probability on lazy/bitmap optimizations.

more effective space saving ratio and ultimately reduced memory requirements of the bitmap structure. These trends are also captured in Table VIII.

Next, we demonstrate experimentally the robustness of our proposed lazy and bitmap-based predicate evaluation techniques, with respect to key workload parameters that may affect its outcome, including workload distribution (Figures 23–24), space dimensionality (Figure 23(a)–23(b)), dimension cardinality (Figure 23(c)–23(d)), event matching probability (Figure 24,) and average subscription and event size (Figure 25.)

In order to investigate the scalability of our BE-Tree's proposed optimizations, we experiment with varying the number of dimension and dimension cardinality. As we

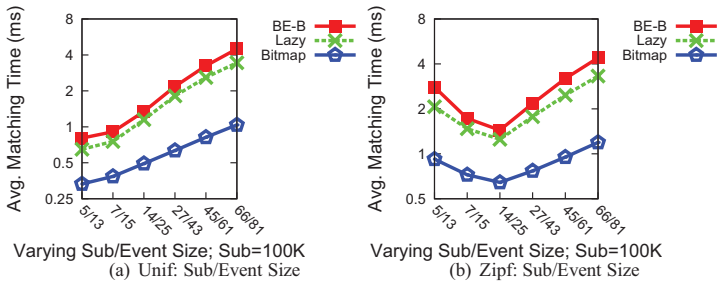


Fig. 25. Effect of sub/event size on lazy/bitmap optimizations.

scale the number of dimension from 50-1400 not only do these optimizations scale well with respect to memory use, but they also improve the matching time by up to 29% and 55% for lazy and bitmap techniques, respectively, (Figure 23(a)–23(b)). Likewise, as we increase the dimension cardinality from 48-150K, our lazy and bitmap techniques continue to outperform BE-Tree by up to 9% and 57%, respectively, (Figure 23(c)–23(d)).

In order to judge the broad applicability of BE-Tree extended with lazy and bitmap optimizations, we study the effect of varying the degree of matching probability for both synthetic and real workloads. These optimizations substantially outperform the BE-Tree structure as we increase the matching probability because a higher matching probability translates into fewer number of distinct predicates, which in turn is beneficial to both techniques. As a result, on average, with the lazy predicate evaluation, we obtain up to 43% improvement while the bitmap approach significantly reduces BE-Tree's matching time by up to 64%, as shown in Figure 24.

A key distinguishing workload parameter that demonstrates the effectiveness of our lazy and bitmap evaluations is the effect of changing subscription and event size because as we increase the number of predicates per subscription, the number of predicate evaluations also increases resulting in a larger saving from evaluating every distinct predicate exactly once. As a result, for the lazy optimization, on average, we reduce the matching time by up to 24% for both uniform and Zipf distributions while for the bitmap optimization, on average, we achieve up to 75% improvement for both workloads (Figure 25.)

The final aspect of our bitmap optimization is to demonstrate the effectiveness of our 2-dimensional subscription representation. Therefore, we consider two variations of BE-Tree with bitmap optimization: bitmap, which is the base version, and bitmap-CC, which is the cache-conscious version that incorporates 2-dimensional subscription representation. As discussed previously, with 2-dimensional representation both subscriptions and Result Bit-array could potentially fit in the processor cache entirely and eliminate all cache misses. This reduction in the number of cache misses is clearly evident by significant reduction in matching time by up to 70% as shown in Figure 26.

8.7.2. Bloom Filter Optimization. We conclude our experimental studies with an analysis of the Bloom filter optimization. In these experiments, as we increase the workload size, we vary the Bloom filter size from 16–64 bits, as shown in Figure 27. As expected, the increase in the Bloom filter size decreases the false positive rate of the Bloom filter, which in turn, results in an improved matching time by eliminating false candidate matches. However, as the Bloom filter size is increased, the overhead for checking the necessary condition $\mathcal{C}$ through the Bloom filter is also increased. Thus, in our experiment, a 32-bit Bloom filter achieved the right balance between the size and and reduction in false positive rate. For instance, when a 32-bit Bloom filter is used, the matching time of BE-Tree with lazy optimization is improved by 70% and 50% for

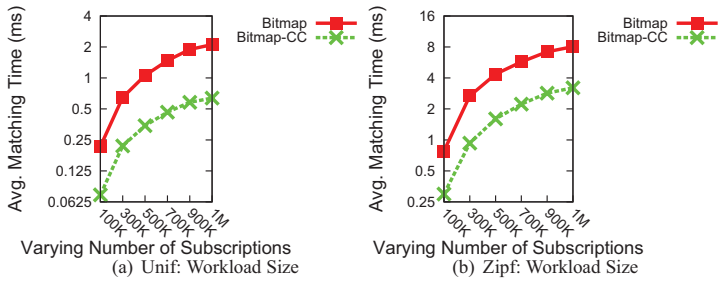


Fig. 26. Effect of cache-conscious data structure on bitmap optimization.

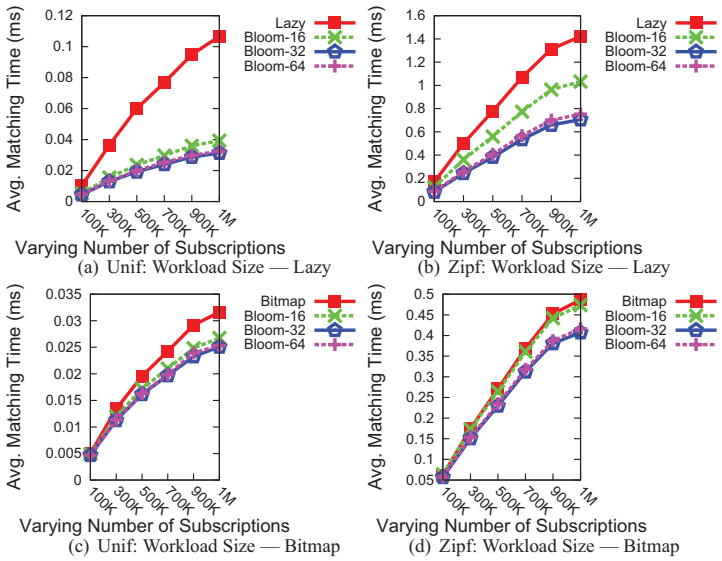


Fig. 27. Effect of workload size on bloom filter optimization.

uniform and Zipf distributions (Figure 27(a)–27(b)) while the matching time of BE-Tree with bitmap optimization is improved by 21% and 16% for uniform and Zipf distributions (Figure 27(c)–27(d)). As expected, the Bloom filter optimization is more effective for lazy optimization, in which the cost of predicate evaluation is higher compared to when the bitmap optimization is used instead; hence, the benefit of false candidate reduction of Bloom filter is more prominent in conjunction with the lazy predicate evaluation. In general, based on our findings, the Bloom filter optimization is best suited when matching probability is low, that is, event matches are rare, and, more importantly, there are only handful of predicates (or attribute-value pairs) in each event.

8.8. Experimental Summary

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, our Advanced Gryphon (ADGR), *k*-index [Whang et al. 2009], Propagation [Fabret et al. 2001], APP [Farroukh et al. 2011], and SIFT [Yan and García-Molina

1994]. In the second category (Zipf) we have: BE-Tree, Propagation, ADGR, k-index, APP, and SIFT. Lastly, in the third category we have: BE-Tree, ADGR, Gryphon [Aguilera et al. 1999], Propagation, and k-index. The general trends are that non-key-based algorithms, that is, 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, yet by incorporating our lazy and bitmap-based predicate evaluation optimizations, BE-Tree's matching time is further reduced by up to 43% and 75%, respectively. We can shape off up to an additional 70% of the matching time of BE-Tree (which is already extended with our predicate evaluation optimizations) after also applying the Bloom filter optimization when matching probability is low.

9. CONCLUSIONS

In this work, we presented BE-Tree, a novel index structure to efficiently index and match Boolean Expressions 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 appears in many application domains. Furthermore, we developed a new cost model and self-adjustment policies that enabled BE-Tree to actively adapt to workload changes. Moreover, we propose scalable and effective predicate evaluation techniques, that is, lazy and bitmap optimizations, which substantially improve BE-Tree's matching computation. Finally, through an extensive experimental evaluation, we demonstrated that BE-Tree is a generic index for Boolean expressions that supports variety of workload configurations and handles predicates with expressive set of operators.

Consequently, we presented a wide range of applications, including distributed event-based systems, applications in the cospace, targeted web advertising, and approximate string matching, that can benefit from a general purpose indexing technique for Boolean expressions.

ELECTRONIC APPENDIX

The electronic appendix for this article can be accessed via the ACM Digital Library. It includes the BE-Tree deletion algorithm, detailed correctness proofs for BE-Tree theorems, and additional experiments including a comparison with a matching algorithm running on GPUs.

REFERENCES

AGRAWAL, R., AILAMAKI, A., ET AL. 2008. The Claremont report on database research. *SIGMOD Rec. 37*, 3, 9–19.

AGUILERA, M. K., STROM, R. E., STURMAN, D. C., ASTLEY, M., AND CHANDRA, T. D. 1999. Matching events in a content-based subscription system. In *Proceedings of the 18th Annual ACM Symposium on Principles of Distributed Computing (PODC'99)*. ACM, New York, 53–61.

ARUMUGAM, S., DOBRA, A., JERMAINE, C. M., PANSARE, N., AND PEREZ, L. 2010. The DataPath system: A datacentric analytic processing engine for large data warehouses. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'10)*. ACM, New York, 519–530.

BECKMANN, N., KRIEGEL, H.-P., SCHNEIDER, R., AND SEEGER, B. 1990. The R*-tree: An efficient and robust access method for points and rectangles. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'90)*. ACM, New York, 322–331.

BERCHTOLD, S., KEIM, D. A., AND KRIEGEL, H.-P. 1996. The X-tree: An index structure for high-dimensional data. In *Proceedings of the 22th International Conference on Very Large Data Bases (VLDB'96)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, 28–39.

BERG, M. D., CHEONG, O., KREVELD, M. V., AND OVERMARS, M. 2008. *Computational Geometry: Algorithms and Applications* 3rd Ed., Springer.

Brenna, L., Demers, A., Gehrke, J., Hong, M., Ossher, J., Panda, B., Riedewald, M., Thatte, M., and White, W. 2007. Cayuga: a high-performance event processing engine. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'07)*. ACM, New York, 1100–1102.

Campailla, A., Chaki, S., Clarke, E., Jha, S., and Veith, H. 2001. Efficient filtering in publish-subscribe systems using binary decision diagrams. In *Proceedings of the 23rd International Conference on Software Engineering (ICSE'01)*. IEEE, 443–452.

Candan, K. S., Hsiung, W.-P., Chen, S., Tatemura, J., and Agrawal, D. 2006. AFilter: adaptable XML filtering with prefix-caching suffix-clustering. In *Proceedings of the 32nd International Conference on Very Large Data Bases (VLDB'06)*. VLDB Endowment, 559–570.

Carzaniga, A. and Wolf, A. L. 2003. Forwarding in a content-based network. In *Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM'03)*. ACM, New York, 163–174.

Chan, C.-Y., Felber, P., Garofalakis,M., and Rastogi, R. 2002. Efficient filtering of XML documents with XPath expressions. *VLDB J. 11*, 4, 354–379.

Chandel, A., Hassanzadeh, O., Koudas, N., Sadoghi, M., and Srivastava, D. 2007. Benchmarking declarative approximate selection predicates. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'07)*. ACM, New York, 353–364.

Chandy, K. M., Charpentier, M., and Capponi, A. 2007. Towards a theory of events. In *Proceedings of the Inaugural International Conference on Distributed Event-Based Systems (DEBS'07)*. ACM, New York, 180–187.

Chaudhuri, S., Ganjam, K., Ganti, V., and Motwani, R. 2003. Robust and efficient fuzzy match for online data cleaning. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'03)*. ACM, New York, 313–324.

Cugola, G. and Margara, A. 2012. High-performance location-aware publish-subscribe on GPUs. In *Proceedings of the ACM/IFIP/USENIX 13th International Middleware Conference*. Lecture Notes in Computer Science, vol. 7662, Springer, 312–331.

Diao, Y., Altinel, M., Franklin, M. J., Zhang, H., and Fischer, P. 2003. Path sharing and predicate evaluation for high-performance XML filtering. *ACM Trans. Datab. Syst. 28*, 4, 467–516.

Fabret, F., Jacobsen, H. A., Llirbat, F., Pereira, J., Ross, K. A., and Shasha, D. 2001. Filtering algorithms and implementation for very fast publish/subscribe systems. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'01)*. ACM, New York, 115–126.

Farroukh, A., Sadoghi, M., and Jacobsen, H.-A. 2011. Towards vulnerability-based intrusion detection with event processing. In *Proceedings of the 5th ACM International Conference on Distributed Event-Based System (DEBS'11)*. ACM, New York, 171–182.

Fellegi, I. P. and Sunter, A. B. 1969. A theory for record linkage. *J. Amer. Statist. Assoc. 64*, 328, 1183–1210.

Fischer, P. M. and Kossmann, D. 2005. Batched processing for information filters. In *Proceedings of the 21st International Conference on Data Engineering (ICDE'05)*. IEEE, 902–913.

Fontoura, M., Sadanandan, S., Shanmugasundaram, J., Vassilvitski, S., Vee, E., Venkatesan, S., and Zien, J. 2010. Efficiently evaluating complex Boolean expressions. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'10)*. ACM, New York, 3–14.

Forgy, C. L. 1990. Rete: A fast algorithm for the many pattern/many object pattern match problem. In *Expert Systems*, P. G. Raeth, Ed., IEEE, 324–341.

Freeston, M. 1995. A general solution of the n-dimensional B-tree problem. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'95)*. ACM, New York, 80–91.

Gaede, V. and Günther, O. 1998. Multidimensional access methods. *ACM Comput. Surv. 30*, 2, 170–231.

Giarratano, J. C. and Riley, G. 1989. *Expert Systems: Principles and Programming*. Brooks/Cole Publishing Co., Pacific Grove, CA.

Guttman, A. 1984. R-trees: a dynamic index structure for spatial searching. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'84)*. ACM, New York, 47–57.

Hanson, E. N., Carnes, C., Huang, L., Konyala, M., Noronha, L., Parthasarathy, S., Park, J. B., and Vernon, A. 1999. Scalable trigger processing. In *Proceedings of the 15th International Conference on Data Engineering*. M. Kitsuregawa, M. P. Papazoglou, and C. Pu, Eds., IEEE, 266–275.

Hanson, E. N., Chaabouni, M., Kim, C.-H., and Wang, Y.-W. 1990. A predicate matching algorithm for database rule systems. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'90)*. ACM, New York, 271–280.

Hull, R. 2008. Artifact-centric business process models: Brief survey of research results and challenges. In *Proceedings of the OTM Conferences, Part II*. Lecture Notes in Computer Science, vol. 5332, Springer, 1152–1163.

JERZAK, Z. AND FETZER, C. 2008. Bloom filter based routing for content-based publish/subscribe. In *Proceedings of the 2nd International Conference on Distributed Event-Based Systems (DEBS'08)*. ACM, New York, 71–81.

KALE, S., HAZAN, E., CAO, F., AND SINGH, J. P. 2005. Analysis and algorithms for content-based event matching. In *Proceedings of the 4th International Workshop on Distributed Event-Based Systems (DEBS'05)*. IEEE, 363–369.

MACHANAVAJJHALA, A., VEE, E., GAROFALAKIS, M., AND SHANMUGASUNDARAM, J. 2008. Scalable ranked publish/subscribe. *Proc. VLDB Endow. 1*, 1, 451–462.

MARGARA, A. AND CUGOLA, G. 2013. High performance publish-subscribe matching using parallel hardware. *IEEE Trans. Parallel Distrib Syst 99*, PrePrints, 1.

OOI, B. C., TAN, K. L., AND TUNG, A. 2010. Sense the physical, walkthrough the virtual, manage the co (existing) spaces: A database perspective. *SIGMOD Rec. 38*, 3, 5–10.

RJAIBI,W., DITTRICH, K. R., AND JAEPEL, D. 2002. Event matching in symmetric subscription systems. In *Proceedings of the Conference of the Centre for Advanced Studies on Collaborative Research (CASCON'02)*. IBM Press.

SADOGHI, M. 2012. Towards an extensible efficient event processing kernel. In *Proceedings of the SIGMOD/PODS PhD Symposium (PhD'12)*. ACM, 3–8.

SADOGHI, M., BURCEA, I., AND JACOBSEN, H.-A. 2011. GPX-Matcher: A generic Boolean predicate-based XPath expression matcher. In *Proceedings of the 14th International Conference on Extending Database Technology (EDBT/ICDT'11)*. ACM, New York, 45–56.

SADOGHI, M. AND JACOBSEN, H.-A. 2011. BE-Tree: an index structure to efficiently match Boolean expressions over high-dimensional discrete space. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'11)*. ACM, New York, 637–648.

SADOGHI, M. AND JACOBSEN, H.-A. 2012. Relevance matters: Capitalizing on less (top-k matching in publish/subscribe). In *Proceedings of the IEEE 28th International Conference on Data Engineering (ICDE'12)*. IEEE, 786–797.

SADOGHI, M., JACOBSEN, H.-A., LABRECQUE, M., SHUM, W., AND SINGH, H. 2010. Efficient event processing through reconfigurable hardware for algorithmic trading. *Proc. VLDB Endow. 3*, 2, 1525–1528.

SADOGHI, M., SINGH, H., AND JACOBSEN, H.-A. 2011. Towards highly parallel event processing through reconfigurable hardware. In *Proceedings of the 7th International Workshop on Data Management on New Hardware (DaMoN'11)*. ACM, New York, 27–32.

SAITA, C.-A. AND LLIRBAT, F. 2004. Clustering multidimensional extended objects to speed up execution of spatial queries. In *Proceedings of the 9th International Conference on Extending Database Technology*. Lecture Notes in Computer Science, vol. 2992, Springer, 623–624.

SELLIS, T. K., ROUSSOPOULOS, N., AND FALOUTSOS, C. 1987. The R+-tree: A dynamic index for multi-dimensional objects. In *Proceedings of the 13th International Conference on Very Large Data Bases (VLDB'87)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, 507–518.

TRIANTAFILLOU, P. AND ECONOMIDES, A. A. 2002. Subscription summaries for scalability and efficiency in publish/subscribe systems. In *Proceedings of the 22nd International Conference on Distributed Computing Systems (ICDCSW'02)*. IEEE, 619–624.

WHANG, S. E., GARCIA-MOLINA, H., BROWER, C., SHANMUGASUNDARAM, J., VASSILVITSKII, S., VEE, E., AND YERNENI, R. 2009. Indexing Boolean expressions. *Proc. VLDB Endow. 2*, 1, 37–48.

YAN, T. W. AND GARCÍA-MOLINA, H. 1994. Index structures for selective dissemination of information under the Boolean model. *ACM Trans. Datab. Syst. 19*, 2, 332–364.

Received February 2012; revised August 2012, November 2012; accepted January 2013