PS-Tree-Based Efficient Boolean Expression Matching for High-Dimensional and Dense Workloads

Shuping Ji and Hans-Arno Jacobsen
Middleware Systems Research Group
Technical University of Munich, Germany
University of Toronto, Canada
shuping.ji@tum.de

ABSTRACT

Boolean expression matching is an important function for many applications. However, existing solutions still suffer from limitations when applied to high-dimensional and dense workloads. To overcome these limitations, in this paper, we design a data structure called `PS-Tree` that can efficiently index subscriptions in one dimension. By dividing predicates into disjoint predicate spaces, `PS-Tree` achieves high matching performance and good expressiveness. Based on `PS-Tree`, we first propose a Boolean expression matching algorithm `PSTBloom`. By efficiently filtering out a large proportion of unmatching subscriptions, `PSTBloom` achieves high matching performance, especially for high-dimensional workloads. `PSTBloom` also achieves fast index construction and a small memory footprint. Compared with state-of-the-art methods, comprehensive experiments show that `PSTBloom` reduces matching time, index construction time and memory usage by up to 84%, 78% and 94%, respectively. Although `PSTBloom` is effective for many workload distributions, dense workloads represent new challenges to PSTBloom and other algorithms. To effectively handle dense workloads, we further propose the `PSTHash` algorithm, which divides subscriptions into disjoint multidimensional predicate spaces. This organization prunes partially matching subscriptions efficiently. Comprehensive experiments on both synthetic and real-world datasets show that `PSTHash` improves the matching performance by up to 92% for dense workloads.

PVLDB Reference Format:
Shuping Ji, Hans-Arno Jacobsen. PS-Tree-Based Efficient Boolean Expression Matching for High-Dimensional and Dense Workloads. *PVLDB*, 12(3): 251-264, 2018.
DOI: https://doi.org/10.14778/3291264.3291270

1. INTRODUCTION

We consider the problem of efficiently evaluating a large collection of conjunctive Boolean expressions given a set of attribute value pairs. Efficient Boolean expression matching plays an important role in a growing number of data management and Web applications, such as online advertising [27], online news dissemination [26], workflow management [36], business process execution and monitoring [24], market feed processing [9], multiplayer online gaming [7], and advertising exchanges [17]. These applications can be modeled by the content-based publish/subscribe paradigm, in which subscriptions represent users' interests and events represent messages [14, 33, 34, 39, 22, 16]. As these applications scale to larger user populations, the underlying matching algorithms must be able to handle ever larger volumes of subscriptions at increasing event rates. Taking online advertising as an example, by April 2017, the number of active advertisers on Facebook had grown to 5 million. In March 2018, Facebook had up to 1.45 billion daily active users and 2.2 billion monthly active users [2].

Designing efficient Boolean expression matching algorithms is challenging for at least five main reasons. First, the algorithm must scale to a large number of Boolean expressions (i.e., *subscriptions* in `pub/sub`) defined over a high-dimensional space. Second, the subscriptions may be unevenly distributed over the available dimensions and highly concentrated in some dimensions, which results in dense workloads; the algorithm should be able to efficiently handle dense workloads. Third, fast subscription-index construction and dynamic index updates need to be supported to accommodate changing interests. Fourth, the algorithm should be efficient at handling a high rate of arriving events on the premise of low Boolean expression matching latency. Last, the algorithm should support a rich subscription language to enable expressive modeling of user interests.

In this paper, a *dimension* refers to the value domain underlying an attribute in a subscription. When all or part of the dimensions is associated with a large number of subscriptions, we consider the workload to be *dense*.

A large number of Boolean expression matching algorithms exist [3, 14, 38, 33, 34, 35, 6, 39, 31, 30, 15, 28]. However, these solutions continue to suffer from limitations that affect performance and applicability. For example, `Propagation` [14] suffers from costs incurred while processing the predicate bit vector that tracks the matching predicates of a given event. `k-index` [38] does not support dynamic subscription updates. `OpIndex` [39] possesses high index construction costs when the arrival of subscriptions and events overlaps. Moreover, the existing solutions are not effective at handling dense workloads, which is an important property that is rarely considered by state-of-the-art algorithms.

To overcome these limitations, we first design the *Predicate Space Tree* (`PS-Tree`), a data structure used to in-

Proceedings of the VLDB Endowment, Vol. 12, No. 3
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3291264.3291270

dex subscriptions in one dimension. Then, we propose the PSTBloom and PSTHash matching algorithms for high dimensional and dense workloads, respectively. In PS-Tree, a predicate of a subscription is regarded as a space, and an attribute-value pair of an event is regarded as a point. PS-Tree divides predicates into disjoint predicate spaces and maintains a many-to-many relationship between predicate spaces and subscriptions. Through PS-Tree, the problem of matching an attribute-value pair against a set of subscriptions is transformed into the problem of locating the predicate space to which the attribute-value pair belongs; PS-Tree efficiently solves this problem.

The example in Fig. 1 illustrates the relationship that PS-Tree maintains. In this example, there are two subscriptions: $S_1\{price, in, [0,4]\}$ and $S_2\{price, in, [2,4]\}$. The two predicates involved are divided into two disjoint predicate spaces: $[0,2)$ and $[2,4]$. The first predicate space $[0,2)$ is associated with $\{S_1\}$, while the second predicate space $[2,4]$ is associated with $\{S_1, S_2\}$. For an attribute-value pair, $\langle price, 3\rangle$, after determining the predicate space $[2,4]$ to which it belongs, the matching subscriptions $\{S_1, S_2\}$ can be directly retrieved.

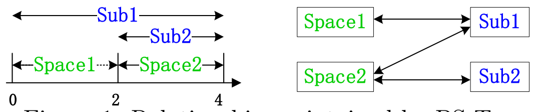


Figure 1: Relationship maintained by PS-Tree

Based on PS-Tree, we first propose the PSTBloom algorithm. For each subscription S in PSTBloom, one of its predicates is selected as the *access predicate*. Access predicates are evaluated before other predicates to retrieve candidate subscriptions, i.e., subscriptions that are likely to match the given input event. The access predicate is divided into several disjoint predicate spaces. Each predicate space corresponds to a leaf node in PS-Tree. S is associated with those leaf nodes that correspond to its access predicate. For an input event, each attribute-value pair of that event is matched against a corresponding PS-Tree to determine a set of partially matching subscriptions. These candidate subscriptions are then pruned by the Bloom filter signatures of the related leaf nodes. The signature of a leaf node is the Bloom filter created from subscription IDs. PSTBloom achieves good matching performance since a large proportion of unmatching subscriptions can be efficiently filtered out. Our comprehensive experiments show that PSTBloom outperforms state-of-the-art algorithms in terms of not only matching time but also index construction time and memory consumption.

PSTBloom is effective for many types of workloads, especially high-dimensional workloads, because PSTBloom can efficiently locate the small number of subscriptions whose access predicates are satisfied by an event. However, dense workloads present new challenges to PSTBloom and other algorithms: the number of partially matching candidate subscriptions could be large for an event. To overcome this limitation, we further design the PSTHash algorithm. PSTHash is also based on PS-Tree. However, in contrast to PSTBloom, PSTHash selects more than one predicate as *access predicates* for each subscription. These access predicates are divided into a set of disjoint multidimensional predicate spaces. PS-Hash constructs a many-to-many relation between the multidimensional predicate spaces and the subscriptions. When an event is received in PSTHash, each attribute-value pair of that event is matched against a corresponding PS-Tree to determine the predicate space to which the attribute-value pair belongs. Then, a number of multidimensional predicate spaces is constructed. Through these multidimensional predicate spaces, the candidate subscriptions can be directly obtained for that event. PSTHash identifies fewer candidate subscriptions since PSTHash guarantees that all the selected access predicates in each candidate subscription have matching attribute-value pairs in the event. Under dense workloads, our experiments show that PSTHash achieves the best matching performance among the algorithms we evaluated.

Our proposed algorithms present advantages over existing solutions, especially for high-dimensional and dense workloads. With this paper, we make the following three main contributions. (1) We propose the PS-Tree index, which achieves high matching performance, exhibits high expressiveness and supports dynamic subscription updates. (2) We propose PSTBloom, which presents advantages over existing solutions for reducing not only the Boolean expression matching latency but also the memory consumption and index construction latency. (3) We propose PSTHash, which achieves the best matching performance given dense workloads. Finally, we offer an extensive evaluation of these algorithms based on both synthetic and real-world workloads.

The remainder of this paper is organized as follows. In Section 2, we review the existing Boolean expression matching algorithms. In Section 3, we present the expression language and matching semantics. The PS-Tree index is presented in Section 4. In Section 5, we describe the design of the PSTBloom algorithm. In Section 6, we present the design of the PSTHash algorithm. Section 7 reports extensively on our experimental results, and Section 8 concludes the paper.

2. RELATED WORK

A large body of work is dedicated to studying Boolean expression matching in many contexts: trigger processing [8, 19], XPath/XML matching [13, 32, 23, 20], indexing in multidimensional space [25, 5, 18], and pub/sub matching [33, 39, 38, 14]. We concentrate on pub/sub matching since other contexts either use different languages or cannot scale to thousands of dimensions and millions of expressions. Existing solutions in pub/sub matching include BE-Tree [33, 34, 35], OpIndex [39], Propagation [14], k-index [38], SIFT [3], Gryphon [3], H-Tree [31], TAMA [40], REIN [30], GEM [15], and SCAN [3]. These solutions can be roughly classified into two classes: count-based methods and tree-based methods.

Count-based approaches [39, 14, 38, 3] usually design indexes to retrieve a set of candidate subscriptions. For these candidate subscriptions, count-based methods identify the matching subscriptions by counting the satisfied predicates of each subscription. Representative count-based approaches include Propagation, k-index, and OpIndex.

Propagation [14] creates indexes on predicates and maintains a *predicate bit vector*. Each predicate that occurs in one or more subscriptions is associated with a single entry in the predicate bit vector. When an event is received, Propagation first locates the satisfied predicates and sets related entries in the predicate bit vector to 1. Candidate subscriptions can be obtained through these predicates. Then, Propagation identifies a candidate subscription as matching if all of its predicates are satisfied. While Propagation is a novel algorithm, it suffers from the limitation that, to match each event, all bits in the predicate bit vector need to be reinitiated to 0. This operation is expensive if there are a large number of distinct predicates.

`k-index` [38] is an approach based on an inverted index. The basic idea behind `k-index` is to partition the subscriptions into subsets of predicates and to organize each predicate subset using the inverted list data structure. For each attribute-value pair in an incoming event, appropriate inverted list indexes are searched to identify predicates matching the attribute-value pair, and a counting method is used to determine the matching subscriptions for an event. `k-index` is effective in retrieving partially matching subscriptions; however, it suffers from the limitation that a range predicate must be rewritten to a disjunction of equality predicates, which increases the index's size due to the need for many inverted list entries for a single predicate.

`OpIndex` [39] is a state-of-the-art count-based method. This method adopts a two-level index structure and organizes subscriptions using inverted lists. In the first level, `OpIndex` selects an attribute for each subscription, and subscriptions with the same selected attribute are grouped together. In the second level, subscriptions are further partitioned based on their predicate operators. The predicates with the same operator are clustered together. For an incoming event, `OpIndex` retrieves all the satisfied predicates and then uses the counting method to locate the matching subscriptions. `OpIndex` has high matching performance. However, when the arrival of subscriptions and events overlaps, `OpIndex` suffers from high index construction costs since related indexes need to be reordered after each subscription is inserted. During the reordering process of an index, event matching is delayed, which results in high matching latency.

In contrast to count-based methods, tree-based methods [33, 3, 6] are designed to filter out unmatching subscriptions step by step. These methods usually recursively divide the search space by eliminating subscriptions upon encountering unsatisfied predicates. Compared to count-based methods, tree-based methods usually identify fewer candidate subscriptions, but the filtering process is more expensive.

`BE-Tree` [33], a representative tree-based method, uses a two-phase space cutting technique and organizes subscriptions in a tree index. The subscriptions are repeatedly partitioned by attribute, followed by value space partitioning. In `BE-Tree`, there are three classes of nodes: a partition node that maintains the space partitioning information (an attribute), a cluster node that maintains the space clustering information (a range of values), and a leaf node that stores the actual expressions. By configuring the maximum leaf node size, `BE-Tree` achieves a good trade-off between matching performance and memory consumption. However, `BE-Tree` indexes all subscriptions through a single tree, which constitutes a potential performance bottleneck.

`H-Tree` is also a tree-based method. `H-Tree` first selects a set of attributes to index. Then, the value domain of each attribute is divided into a number of overlapping cells. The cells of each attribute are connected level-by-level to form a tree. In this way, `H-Tree` divides the subscription space into a set of buckets. The number of buckets increases exponentially with the number of indexed attributes. As a result, `H-Tree` is not suitable for high-dimensional workloads.

`GEM` [15] and `REIN` [30] are different from the count-based and tree-based methods presented above. The idea behind these two algorithms is to filter out the unmatching subscriptions one by one for each event. `GEM` and `REIN` are not suitable for scenarios in which the proportion of unmatching subscriptions is high. `TAMA` [40] is an approximate matching algorithm, which means that it can introduce false positives into the result set.

`PS-Tree` can be interpreted to store intervals and allows efficient querying of which stored intervals contain a given point. `Segment-Tree` [11] and `Interval-Tree` [10] are two index structures that provide similar capabilities. Thus, both approaches are related to `PS-Tree`. One limitation of `Segment-Tree` is that it is a static structure, meaning that `Segment-Tree` cannot be modified after it is built. A variant [4] of `Segment-Tree` supports dynamic updates at the cost of extra memory. `Interval-Tree` does not have this limitation but presents more expensive querying. Compared to `Segment-Tree` and `Interval-Tree`, `PS-Tree` achieves good querying performance, as shown in Section 7.5. Classical string and bit-vector tree indexing structures, such as a trie [12] and Radix-, Patrica- and Suffix-tree [29, 37], present a similar structure as that of `PS-Tree`. However, they are different because they do not provide the function to match attribute-value pairs against predicates.

3. MATCHING MODEL

3.1 Expression Language

Predicate: A predicate is a triple consisting of an attribute, an operator and a set of values. A predicate is denoted as $P^{attr,op,vals}(x)$, or more concisely as $P(x)$. The attribute name P^{attr} uniquely represents a dimension. A predicate may contain more than one value. The number of values is determined by the operator. For example, if the operator is "$=$", the predicate contains a single value; if the operator is "$\in$", the predicate contains a set of values.

The expressiveness of the predicates is determined by the supported attribute value types and operators. To achieve high expressiveness, the predicates in our expression language support the standard relational operators ($<, \leq, =, \neq, >, \geq$) and set operators ($\in, \notin$) as well as SQL's BETWEEN operator (*in*) on numerical, enumeration, and string domains.

Subscription: A subscription is a conjunctive Boolean expression over predicates. Suppose that the total number of dimensions is n. Formally, a subscription S is defined over an n-dimensional space as follows:

$$S = \{P_1^{attr,op,vals}(x_1) \wedge ... \wedge P_k^{attr,op,vals}(x_k)\}, \quad k \leq n$$

Different predicates in the same subscription are required to belong to different dimensions: $P_i^{attr} \neq P_j^{attr}, \ \ if \ i \neq j$. We refer to the size of a subscription S, denoted by $|S|$, as the number of predicates in S.

Event: An event contains a set of attribute-value pairs. Formally, an event E is defined over an n-dimensional space as follows:

$$E = \{\langle attr_1, val_1\rangle, ... , \langle attr_k, val_k\rangle\}, \quad k \leq n$$

For the same event, different attribute-value pairs are required to belong to different dimensions: $attr_i \neq attr_j$, $if \ i \neq j$. We refer to the size of an event E, denoted by $|E|$, as the number of attribute-value pairs in E.

3.2 Matching Semantics

A predicate accepts an input value x and outputs a Boolean value indicating whether that predicate constraint is satisfied:

$$P^{attr,opt,vals}(x) \rightarrow \{True, False\}$$

A predicate $P^{attr,opt,vals}(x)$ matches with an attribute-value pair $\langle attr, val\rangle$, denoted by $P^{attr,opt,vals}(x) \simeq \langle attr, val\rangle$, if the following condition is satisfied:

$$\{P^{attr} = attr\} \wedge \{P^{attr,opt,vals}(val) = True\}$$

If a predicate of a subscription S matches with an attribute-value pair $\langle attr, val\rangle$, we say that the subscription matches with that attribute-value pair, denoted by $S \simeq \langle attr, val\rangle$. Furthermore, a subscription S matches with an event E, denoted by $S \simeq E$, if the following condition is met:

$$\forall P(x) \in S \rightarrow \{\exists\langle attr, val\rangle \in E\} \wedge \{P(x) \simeq \langle attr, val\rangle\}$$

Given an event E and a set of subscriptions, retrieve all the subscriptions matched by E. We refer to this problem as the Boolean expression matching problem.

3.3 Predicate Selectivity

Boolean expression matching can be seen as a process for filtering out unmatching subscriptions for a given event. We observe that different predicates have different pruning capacities, i.e., *predicate selectivity.* For a subscription S, we define the selectivity of its predicate $P(x)$ as the probability that S is identified as unmatching if $P(x)$ is used as the pruning predicate for an arbitrary event.

The selectivity of a predicate is determined by three factors: (1) the distribution of the event workload, (2) the operator of the predicate, and (3) the values of the predicate. Since the whole event workload is often unknown in advance, in our algorithms, we use statistics over historical events to calculate the selectivity of a predicate: a predicate with a higher number of matching events is considered to have lower selectivity. The advantage of our algorithms is that PS-Tree can be used to quickly obtain the number of matching historical events for a predicate. If no historical events exist, we heuristically rank the selectivity of predicates by their operators and values. The selectivity ranking of operators is $\{=\} > \{\in\} > \{in\} > \{<, \leq, >, \geq\} > \{\notin\} > \{\neq\}$. When the operators have the same selectivity, we consider predicates with wider value sets to have lower selectivity. For example, we consider the selectivity of $\{attr, in, [1, 10]\}$ to be lower than the selectivity of $\{attr, in, [1, 5]\}$.

4. PS-TREE DESIGN

PS-Tree is a novel tree index for subscriptions. The idea behind PS-Tree is to divide the set of values represented by all predicates into disjoint subsets, here referred to as *predicate spaces.* In other words, a predicate space is a value range in the predicate's dimension (i.e., value domain). Then, we construct a many-to-many relationship between predicate spaces and subscriptions. Thus, the problem of matching an attribute-value pair against a set of subscriptions is transformed into the problem of locating the predicate space to which an attribute-value pair belongs, which can be efficiently supported by PS-Tree.

4.1 PS-Tree Structure

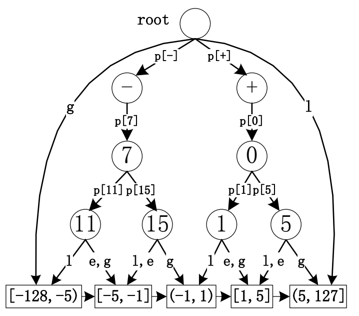
PS-Tree contains two types of nodes: leaf nodes and inner nodes. A leaf node represents a predicate space, while an inner node represents an "element" of the represented attribute values. Elements are specified differently for different value types. For example, we specify elements as digits for the integer type. The root node of PS-Tree is a special inner node that represents the starting element of the represented attribute values. The inner nodes on the path from the root node to a leaf node construct the represented attribute value and act as the boundary between two adjacent predicate spaces.

As shown in Fig. 2(a), an inner node contains three leaf node links (l, e, g) and an array of links to child inner nodes. The *length* of the inner node link array is set as the number of different elements. If an inner node corresponds to an attribute value, say, v, l of this inner node links to the leaf node whose predicate space is less than v, e links to the leaf node whose predicate space is equal to v, and g links to the leaf node whose predicate space is greater than v. As an optimization, e can link to the same leaf node as l and g. For the root node, a special consideration is that g links to the first leaf node, and l links to the last leaf node.

```
InnerNode {
   LeafNodeLink l;
   LeafNodeLink e;
   LeafNodeLink g;
   InnerNodeLink p[LENGTH];
}

LeafNode {
   LeafNodeLink next;
   Integer counter;
   list<Sub> subLinkedList;
   BloomFilter signature;
   Integer spaceId;
}
```

(a) Node design



(b) PS-Tree index example

Figure 2: PS-Tree index structure

In PS-Tree, a leaf node contains a *next link*, a *predicate counter*, an *event counter*, a *subscription linked list* and a *signature.* The next link points to another leaf node whose predicate space is adjacent and greater than that leaf node's predicate space. The predicate counter equals the number of predicates covering[1] the current leaf node's predicate space. The event counter equals the number of historical events that match with the current leaf node's predicate space. The subscription linked list contains links to subscriptions. The signature is the Bloom filter created from the subscription IDs. When a predicate of a subscription covers the predicate space of a leaf node, the ID of that subscription is inserted into the signature of that leaf node. The sub linked list and the signature are only required by PSTBloom. For PSTHash, a leaf node maintains a unique predicate space ID, which is not required by PSTBloom. As a result, we have two versions of PS-Tree, PS-Tree$_B$ and PS-Tree$_H$, which have different space complexities.

The elements are specified differently for different value types. In this paper, we use the 8-bit byte type to illustrate the design of PS-Tree. Fig. 2(b) shows a PS-Tree instance with two subscriptions, $S_1\{attr, in, [-5, -1]\}$ and S_2 $\{attr, in, [1, 5]\}$, indexed. Each of these subscriptions contains one predicate in the dimension *attr*. In this example, the value type of this dimension is an 8-bit byte. For a byte, which is stored as a binary complement, we specify the most significant bit, the next 3 bits, and the last 4 bits as an element. Take the attribute value -5 as an example; its elements are "$-$", "7" and "11". In this PS-Tree instance,

[1]In this paper, we use the term *cover* to indicate that an interval contains another interval or a point.

there are five leaf nodes. Correspondingly, the whole value domain $[-128, 127]$ of the dimension *attr* is divided into five predicate spaces: $[-128, -5)$, $[-5, -1]$, $(-1, 1)$, $[1, 5]$, and $(5, 127]$. These predicate spaces are mapped to the following five subscription sets, respectively: $\varnothing$, $\{S_1\}$, $\varnothing$, $\{S_2\}$, and $\varnothing$. Given an attribute value pair $\langle attr, 2\rangle$, the predicate space $[1, 5]$ to which it belongs can be quickly located through `PS-Tree`. Subsequently, the matching subscription set $\{S_2\}$ can be directly obtained.

4.2 Index Construction

Alg. 1 processes the inserted predicates and constructs the `PS-Tree` index. *InsertPredicate* takes two parameters, *pred* and *pstree*, as input. *pred* is the predicate to be inserted. The output is a list of leaf nodes whose predicate spaces are covered by the inserted predicate. *InsertPredicate* addresses predicates differently for different operators. Alg. 1 shows how the operator "$\geq$" is handled. Other operators are handled in a similar manner.

Algorithm 1 InsertPredicate($pred, pstree$)

```
1: if $pred.op$ $is$ $\geq$ then
2:     startNode←Partition($pred.vals[0], pred.op, pstree$)
3:     endNode←pstree.root.l
4: while $startNode \neq endNode.next$ do
5:     startNode.predCounter++
6:     leafNodes.add($startNode$)
7:     startNode←startNode.next
8: return leafNodes
```

Algorithm 2 Partition($val, op, pstree$)

```
1: currNode←pstree.root
2: for each $elem \in val$ do
3:     path.push($currNode, elem$)
4:     if $currNode.p[elem] = null$ then
5:         currNode.p[elem]←CreateInnerNode()
6:     currNode←currNode.p[elem]
7: if $currNode.e = null$ then
8:     iRNode←GetRNode($path$)
9:     iLNode←GetLNode($iRNode, pstree.root$)
10:    PartitionLeafNode($currNode, iLNode, op$)
11: else
12:    PartitionLeafNode($currNode, op$)
13: return currNode.e
```

Algorithm 3 MatchPair($pair, pstree$)

```
1: currNode←pstree.root
2: for each $elem \in pair.val$ do
3:     path.push($currNode, elem$)
4:     if $currNode.p[elem] \neq null$ then
5:         currNode←currNode.p[elem]
6:     else
7:         iRNode←GetRNode($path$)
8:         iLNode←GetLNode($currNode, pstree.root$)
9:         return iLNode.l
10: return currNode.e
```

Partition is a function invoked by *InsertPredicate* to partition a predicate space in a `PS-Tree`. The input parameters include a value, an operator, and a `PS-Tree`. The output is a leaf node whose predicate space covers the input value. As shown in Alg. 2, if not all inner nodes corresponding to that value exist, new inner nodes are created. The function *GetRNode* is used to locate the inner node adjacent to and to the right of the current inner node. *GetLNode* is used to locate the minimal left inner node. *PartitionLeafNode* is used to partition the predicate space of a leaf node and create new leaf nodes. Except for the space ID and the next link, a newly created leaf node copies other content from the leaf node to be partitioned. The detailed specifications and explanation of *GetRNode*, *GetLNode*, *CreateInnerNode* and *PartitionLeafNode* are presented in the technical report [21].

4.3 Query Processing

Alg. 3 matches an attribute-value pair against a `PS-Tree` to locate the predicate space to which it belongs. Two situations can occur: (1) all the inner nodes corresponding to the value in that attribute-value pair exist, in which case the last inner node's link e is returned, or (2) not all corresponding inner nodes exist, in which case *GetRNode* and *GetLNode* are invoked to locate the inner node whose l links to the leaf node with the predicate space covering the value.

4.4 Dynamic Index Adjustment

`PS-Tree` supports dynamic index adjustment by providing the function *DeletePredicate*, which is used to delete the outdated predicates and nodes from a `PS-Tree`. As mentioned in Sec. 4.1, every leaf node contains a *predicate counter*, which is equal to the number of predicates covering the current leaf node's predicate space. When the predicate counter is zero, the corresponding leaf node is deleted to save memory. The implementation details and complexity analysis of *DeletePredicate* and the subscription deletion operation of `PSTBloom` and `PSTHash` are presented in [21].

Here, we give an additional example that focuses on predicate space partitioning and dynamic `PS-Tree` index adjustment. Suppose, initially, that there are two subscriptions: $S1\{age, in, [20, 60]\}$ and $S2\{age, in, [30, 80]\}$. The value domain of the "age" dimension is $[1, 100]$. Therefore, at the beginning, the value domain is divided into 5 predicate spaces, $[1, 20)$, $[20, 30)$, $[30, 60]$, $(60, 80]$ and $(80, 100]$, with the counters 0, 1, 2, 1 and 0, respectively. When $S2$ is removed, the counters become 0, 1, 1, 0, 0, respectively. Then, the predicate space $[20, 30)$ is merged with $[30, 60]$, and $(60, 80]$ is merged with $(80, 100]$. As a result, there are three predicate spaces remaining: $[1, 20)$, $[20, 60]$, and $(60, 100]$.

4.5 Expressiveness

`PS-Tree` offers high expressiveness by supporting different value types and operators. To support a value type (integer, float, string, etc.), the only requirement is to specify the elements of values of that type. As shown in Fig. 2(b), we divide a byte type value into three elements. For other integer types, the elements are similarly specified as digits. For example, a 32-bit integer type value is divided into nine elements. We use characters as elements for the string type. For the float type, the most significant bit, the exponent bits and the mantissa bits are specified as elements.

`PS-Tree` supports an expressive set of operators; for numbers (e.g., integer, float and double), enumerations, and strings, the supported operations include relational operators ($<, \leq, =, \neq, >, \geq$), set operators ($\in, \notin$), and the SQL operator (*in*). The only requirement to support a specific operator in `PS-Tree` is that the predicate containing that operator is able to be divided into disjoint predicate spaces.

A further advantage of PS-Tree is that it isolates the specific value types and operators from the upper matching layer of PSTBloom and PSTHash. Thus, the upper layer works in the same way for different value types and operators.

4.6 Time and Space Analysis

Matching Time Complexity: In PS-Tree, matching an attribute-value pair against a set of subscriptions is achieved by locating the predicate space to which the attribute-value pair belongs. As shown in Alg. 3, the number of operations is linear in the number of elements of the represented attribute value. Thus, the matching complexity is $O(N_e)$, where N_e is the number of elements in an attribute value. For the integer type, N_e equals nine; therefore, the matching time complexity is only $O(1)$.

Predicate Insertion Time Complexity: In Alg. 1, two steps are needed to insert a predicate into a PS-Tree: (1) insert predicate values into the PS-Tree and (2) determine all leaf nodes covered by the predicate. The time complexity of the first step is also $O(N_e)$. For the second step, the number of operations needed equals the number of predicate spaces covered by the predicate. In the worst case, the number of predicate spaces in a PS-Tree is $2 * N_p + 1$, where N_p represents the number of predicates that have been inserted. Therefore, the predicate insertion time complexity is $O(N_e + N_p)$.

Space Complexity: PS-Tree requires space for inner nodes and leaf nodes. The number of leaf nodes is equal to the number of predicate spaces. In the worst case, the number of inner nodes is N_e times the number of leaf nodes. As analyzed above, the number of leaf nodes is $O(N_p)$. Therefore, the space complexity of PS-Tree$_H$, which is used by the PSTHash algorithm, is $O(N_e * N_p)$. For PS-Tree$_B$, which is used by the PSTBloom algorithm, additional memory is needed to store the subscription list. The space complexity of PS-Tree$_B$ is $O(N_p * (N_e + N_p))$. Note that this is the worst-case space complexity.

5. PSTBLOOM ALGORITHM

Based on PS-Tree, we first design the PSTBloom algorithm. The idea behind PSTBloom is to select a predicate with high selectivity as the *access predicate* for each subscription. Then, the subscription is attached to those leaf nodes corresponding to its access predicate. The ID of the subscription is inserted into the Bloom filter signature of the leaf nodes corresponding to its other predicates. When an event is received, it is matched against a number of PS-Trees to locate its associated leaf nodes. Then, the set of subscriptions whose access predicates match with that event is directly located. Next, we further filter out unmatching subscriptions through the signatures of those leaf nodes. The correctness of PSTBloom is based on Lemma 1.

Lemma 1 Given an event E, the matching subscriptions for E are contained in the candidate subscription set $\{S(\langle attr_i, val_i\rangle) \mid \langle attr_i,\ val_i\rangle \in E\}$, where $S(\langle attr_i, val_i\rangle)$ represents the subscriptions whose access predicates match with the *i-th* attribute-value pair $\langle attr_i, val_i\rangle$ of E.

We sketch the proof for Lemma 1 as follows. If a subscription S matches with an event E, based on the matching semantics in Sec 3.2, we know that there exists a matching attribute-value pair in E for every predicate of S. Therefore, the access predicate of S has a matching attribute-value pair $\langle attr_i, val_i\rangle$ in E. Thus, S is contained in $\{S(\langle attr_i, val_i\rangle)\}$.

5.1 PSTBloom Structure

In PSTBloom, a PS-Tree is constructed for each dimension. As shown in Fig. 2(a), each leaf node of a PS-Tree contains a *subscription linked list* and a *signature*. Each link in the subscription linked list points to a subscription whose access predicate covers the current leaf node's predicate space. The signature is the Bloom filter over a set of subscription IDs. Each such subscription has one predicate (except the access predicate) covering the current leaf node's predicate space.

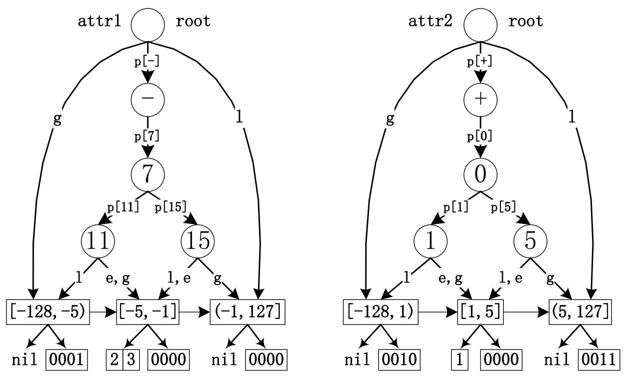


Figure 3: PSTBloom index structure

The example in Fig. 3 illustrates the PSTBloom index structure. In this example, the value type is byte. There are three subscriptions:

$$S_1 : \{attr_1, <, -5\}, \{attr_2, in, [1, 5]\}$$
$$S_2 : \{attr_1, in, [-5, -1]\}, \{attr_2, <, 1\}$$
$$S_3 : \{attr_1, in, [-5, -1]\}, \{attr_2, >, 5\}$$

PSTBloom constructs two PS-Trees, denoted as $attr_1$ and $attr_2$. For S_1, $\{attr_2, in, [1, 5]\}$ is selected as the access predicate, while for S_2 and S_3, $\{attr_1, in, [-5, -1]\}$ is selected as the access predicate. As shown in Fig. 3, a link to S_1 is inserted into the linked list associated with the leaf node with the predicate space $[1, 5]$ in the PS-Tree for $attr_2$. Links to S_2 and S_3 are inserted into the linked list associated with the leaf node with the predicate space $[-5, -1]$ of the PS-Tree for $attr_1$. The ID of S_1 is inserted into the signature of the leaf node with the predicate space $[-128, -5)$ in the PS-Tree for $attr_1$. The ID of S_2 and S_3 is inserted into the signature of the leaf nodes with the predicate space $[-128, 1)$ and $(5, 127]$ of the PS-Tree for $attr_2$, respectively.

Algorithm 4 InsertSubscription($sub, pstb$)

```
1: accPred←SelectAccPred($sub, pstree$)
2: for each $pred \in sub$ do
3:     pstree←pstb.pstrees[pred.attr]
4:     leafNodes←InsertPredicate($pred, pstree$)
5:     for each $node \in leafNodes$ do
6:         if $pred = accPred$ then
7:             node.subLinkedList.add($sub$)
8:         else
9:             node.signature.add($sub.id$)
```

5.2 Index Construction

Alg. 4 processes the inserted subscriptions. When a subscription is received, the predicate with the highest selectivity is selected as the access predicate. Then, every predicate is inserted into its corresponding PS-Tree. For each predicate, a set of leaf nodes is returned after invoking the operation *InsertPredicate* presented in Alg. 1. Based on whether that predicate is an access predicate, different operations are executed. If the predicate is an access predicate, a link to the subscription is inserted into the subscription linked list of each leaf node; otherwise, the subscription ID is inserted into the signature of each leaf node.

5.3 Event Matching

As shown in Alg. 5, PSTBloom takes three steps to match an event against subscriptions: (1) Match each attribute-value pair in the event against the PS-Trees to locate a set of leaf nodes. The subscriptions in the subscription linked lists attached to these leaf nodes are candidate subscriptions. (2) Prune a subscription if its ID is not contained in the signature of the related leaf nodes. (3) Match the event against the remaining subscriptions to further filter out false positives. In this step, the access predicate no longer needs to be checked.

Algorithm 5 MatchEvent($event, pstb$)

```
1: for each pair ∈ event do
2:     pstree←pstb.pstrees[pair.attr]
3:     leafNode←MatchPair(pair, pstree)
4:     leafNodes[pair.attr]←leafNode
5: for each node ∈ leafNodes do
6:     for each sub ∈ node.subLinkedList do
7:         isCandidate←True
8:         for each pred ∈ sub do
9:             if pred = sub.accPred then
10:                Continue
11:            nodeSign←leafNodes[pred.attr].signature
12:            if ¬nodeSign.contain(sub.id) then
13:                isCandidate←False; break
14:        if isCandidate = True then
15:            if Match(event, sub) = True then
16:                matchingSubs.add(sub.id)
17: return matchingSubs
```

5.4 Time and Space Analysis

Matching Time Complexity: As analyzed in Sec. 4.6, PS-Tree needs $O(N_e)$ time to locate the predicate space to which an attribute-value pair belongs. Therefore, for PSTBloom, the time complexity to retrieve the candidate subscriptions is $O(|E| * N_e)$, where $|E|$ is the event size. The total time complexity for event matching is $O(|E|*N_e+N_c)$, where N_c is the number of candidate subscriptions.

Index Construction Time Complexity: To insert a subscription, each of its predicates is inserted into a corresponding PS-Tree. As analyzed in Sec. 4.6, the time complexity of this operation is $O(N_e + N_p)$. Thus, the index construction time complexity for PSTBloom is $O(N_s * |S| * (N_e+N_p))$, where N_s is the number of subscriptions and $|S|$ is the subscription size.

Space Complexity: PSTBloom maintains a PS-Tree, more specifically PS-Tree$_B$, for each dimension. As analyzed in Sec. 4.6, the space complexity of PS-Tree$_B$ is $O(N_p * (N_e + N_p)$. To store all PS-Trees, PSTBloom requires $O(N_d * N_p * (N_e + N_p))$ space, where N_d is the number of dimensions and N_p is the number of predicates in a dimension.

6. PSTHASH ALGORITHM

Although PSTBloom achieves good performance with respect to event matching, index construction, and memory use, it suffers from a limitation in addressing dense workloads because PSTBloom uses only one access predicate to select candidate subscriptions. Given dense workloads, the number of candidates could potentially be large.

To overcome this limitation, we design the PSTHash algorithm, which is also based on PS-Tree. The idea behind PSTHash is to select more than one, say, N, predicates with high selectivity as *access predicates* for each subscription. These access predicates are divided into a number of disjoint *N-dim* predicate spaces. Each *N-dim* predicate space contains N predicate spaces. For an event, a subscription is identified as a candidate only when all N access predicates are matched.

PSTHash differs from PSTBloom in its method of identifying candidate subscriptions. In PSTHash, a many-to-many hash table between *N-dim* predicate spaces and subscriptions is maintained. Given an event E, each attribute-value pair of E is matched against a corresponding PS-Tree to identify the predicate spaces to which the attribute-value pair belongs. In total, $|E|$ predicate spaces can be found. Then, $\binom{|E|}{N}$ *N-dim* predicate spaces are constructed. Through these *N-dim* predicate spaces, the candidate subscriptions can be directly retrieved by querying the hash table that relates the *N-dim* predicate spaces and the subscriptions. Given dense workloads, compared with PSTBloom, PSTHash identifies fewer candidate subscriptions and achieves better matching performance. Assuming that N is not greater than the size of all events and subscriptions[2], we formulate Lemma 2.

Lemma 2 Given an event E, the matching subscriptions for E are contained in the candidate subscription set $\{S(AV_{i_1}, ..., AV_{i_N}) \mid \{AV_{i_1}, ..., AV_{i_N}\} \in E, i_x \neq i_y, x \neq y\}$, where $S(AV_{i_1}, ..., AV_{i_N})$ represents the subscriptions whose access predicates match with the i_1-*th* to the i_N-*th* attribute-value pair of E.

We sketch a proof. If a subscription S matches with an event E, based on Sec. 3.2, there exists a matching attribute-value pair in E for every predicate of S. Therefore, any access predicate of S also has a matching attribute-value pair in E, which means that S is contained in a specific $S(AV_{i_1}, ..., AV_{i_N})$.

The number of access predicates N is an important configuration parameter in PSTHash. When N is large, the access predicates of a subscription are divided into more *N-dim* predicate spaces, PSTHash consumes more memory, and index construction becomes more expensive; however, the matching performance improves, thus producing a trade-off. The most suitable value of N is dependent on workload and beyond the scope of this paper. For simplicity, in the following sections, we always set N equal to two.

6.1 PSTHash Structure

In addition to a set of PS-Trees, PSTHash utilizes two more data structures: a set of *2-dim* predicate space IDs and a hash table. A *2-dim* predicate space ID is constructed using 2 predicate space IDs. A predicate space ID is determined by a dimension ID and space ID pair. Thus, a *2-dim* predicate space ID is represented by 2 pairs of ⟨*dimension ID, space ID*⟩. The pairs are ordered in ascending order by their dimension ID values. The key of the hash table is a *2-dim* predicate space ID, and the associated value is a subscription linked list.

[2]We use the method of PSTBloom to address subscriptions and events whose size is less than N.

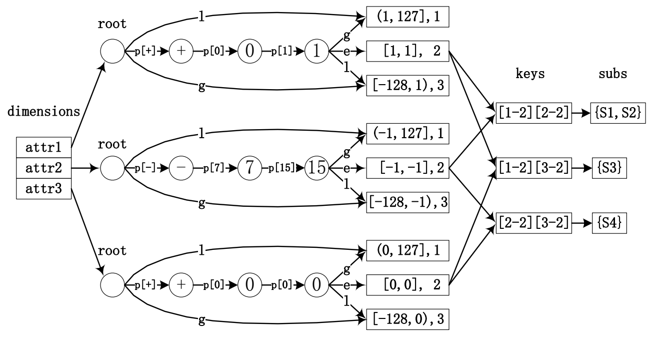


Figure 4: PSTHash index structure

Fig. 4 illustrates the index structure of `PSTHash`. Here, there are four subscriptions, all of which contain three predicates.

$$S_1: \{attr_1, =, 1\}, \{attr_2, =, -1\}, \{attr_3, <, 0\}$$
$$S_2: \{attr_1, =, 1\}, \{attr_2, =, -1\}, \{attr_3, >, 0\}$$
$$S_3: \{attr_1, =, 1\}, \{attr_2, <, -1\}, \{attr_3, =, 0\}$$
$$S_4: \{attr_1, <, 1\}, \{attr_2, =, -1\}, \{attr_3, =, 0\}$$

In this `PSTHash` index example, S_1 and S_2 are associated with the 2-dim predicate space whose ID is [1-2][2-2]. This ID indicates that the 2-dim predicate space is constructed using the second predicate space of $attr_1$ and the second predicate space of $attr_2$. Similarly, S_3 is associated with [1-2][3-2], and S_4 is associated with [2-2][3-2].

Assume that an event E $\{\langle attr_1, 1\rangle, \langle attr_2, -1\rangle, \langle attr_3, 2\rangle\}$ is received. The event is matched against these three `PS-Trees`. Three predicate spaces to which E belongs are identified: [1-2], [2-2] and [3-1]. These three predicate spaces are used to construct three 2-dim predicate spaces: [1-2][2-2], [1-2][3-1], and [2-2][3-1]. `PSTHash` uses these 2-dim predicate space IDs as keys to retrieve the candidate subscriptions S_1 and S_2. These two candidate subscriptions are evaluated, and the matching subscription S_2 is found.

6.2 Index Construction

Alg. 6 processes subscriptions for insertion. When a subscription S is received, two predicates with high selectivity are selected as access predicates. Then, each access predicate is inserted into a corresponding `PS-Tree`. A set of leaf nodes is returned after invoking *InsertPredicate*. Using the dimension IDs and space IDs of those leaf nodes, a number of *2-dim* predicate space IDs are constructed. The subscription is inserted into the hash table with the *2-dim* predicate space ID as the key. When *InsertPredicate* is invoked, predicate spaces may need to be partitioned. In this situation, the records associated with the old space ID in the hash table are copied to new records using the new space ID as the key.

Algorithm 6 InsertSubscription($sub, psth$)

```
 1: accPreds←SelectAccPreds($sub, pstree, 2$)
 2: for each pred $\in$ accPreds do
 3:     pstree←psth.pstrees[pred.attr]
 4:     leafNodes←InsertPredicate($pred, pstree$)
 5:     for each node $\in$ leafNodes do
 6:         spaceIdSets[pred.attr].add($node.spaceId$)
 7: for each spaceId1 $\in$ spaceIdSets[$attr_1$] do
 8:     for each spaceId2 $\in$ spaceIdSets[$attr_2$] do
 9:         key = [$attr_1$-spaceId1][$attr_2$-spaceId2]
10:         psth.hash[key].add($sub$)
```

6.3 Event Matching

As shown in Alg. 7, `PSTHash` takes three steps to match an event against the subscriptions: (1) Match each attribute-value pair in the event against a corresponding `PS-Tree` to obtain $|E|$ predicate space IDs. (2) Construct $\binom{|E|}{2}$ *2-dim* predicate space IDs using those predicate space IDs and retrieve the candidate subscriptions. (3) Match the event against these subscriptions to find matched subscriptions. In this step, the two access predicates no longer need to be checked.

Algorithm 7 MatchEvent($event, psth$)

```
 1: for each pair $\in$ event do
 2:     pstree←psth.pstrees[pair.attr]
 3:     leafNode←MatchPair($pair, pstree$)
 4:     predSpaces.add($pair.attr, leafNode.spaceId$)
 5: spaceIds←ConstructSpaces($predSpaces$)
 6: for each spaceId $\in$ spaceIds do
 7:     subLinkedList←psth.hash[spaceId]
 8:     for each sub $\in$ subLinkedList do
 9:         if Match($event, sub$) = $True$ then
10:             matchingSubs.add($sub.id$)
11: return matchingSubs
```

6.4 Time and Space Analysis

Matching Time Complexity: Similar to `PSTBloom`, the time complexity of locating the predicate spaces to which an event E belongs is $O(|E| * N_e)$. The number of *2-dim* predicate spaces is $\binom{|E|}{2}$. Therefore, the matching time complexity for `PSTHash` is $O(|E| * N_e + \binom{|E|}{2} + N_c)$, where N_c is the number of candidate subscriptions.

Index Construction Time Complexity: Given a subscription, `PSTHash` needs to insert its access predicates into `PS-Trees` and insert that subscription into a number of slots in the hash table. For these two operations, the time complexity is $O(N_e+N_p)$ and $O((N_p)^2)$, respectively. Therefore, the index construction complexity of `PSTHash` is $O(N_s*(N_e+(N_p)^2))$, where N_s is the number of subscriptions.

Space Complexity: `PSTHash` needs memory to store a `PS-Tree`, more specifically, `PS-Tree`$_H$, for each dimension. As analyzed in Sec. 4.6, the space complexity of `PS-Tree`$_H$ is $O(N_e + N_p)$. N_p is the number of access predicates in a dimension. `PSTHash` also needs memory for the hash table. In the worst case, the number of records in the hash table is $N_d * (N_p)^2$, where N_d is the number of dimensions. Thus, the space complexity of `PSTHash` is $O(N_d * N_e * N_p) + O(N_d * (N_p)^2) = O(N_d * N_p * (N_e + N_p))$.

7. EVALUATION

This section evaluates `PSTBloom`, `PSTHash`, and `PS-Tree` using both synthetic and real-world datasets. `BE-Tree`, `OpIndex`, `Propagation`, `k-index`, and `SCAN` (a sequential scan of the subscriptions) are selected as baselines. With the exception of `SCAN`, these algorithms have been shown to exhibit good performance in the literature[3]. `PS-Tree` is compared with `Segment-Tree` and `Interval-Tree`, which allows one to query which of the stored intervals contain a given point.

[3]We also compared `PSTBloom` and `PSTHash` with `SIFT`, `Gryphon`, `REIN`, and `GEM`. These algorithms do not exhibit comparable performance; thus, we omitted the experimental results to focus on better-performing algorithms.

All algorithms are implemented in C[4] and compiled with gcc 4.8.4 using the optimization level O3 on a Ubuntu 14.04 system. All experiments were run on an Intel 2.66 GHz machine with 512 GB of memory.

In the experiments, we consider a variety of controlled experimental conditions: workload size, workload distribution, dimension number, dimension cardinality, subscription size, event size, matching probability, and predicate selectivity.

7.1 Workloads

We first used the `BE-Gen` workload generator [33] to generate synthetic workloads. Table 1 summarizes the parameters and settings, with the default values highlighted in bold. To evaluate scalability, we vary the number of subscriptions from 300K to 100M. The attributes of the predicates were drawn from the distribution $P(r) = \frac{C}{r^{\alpha}}, r \neq 0$. When α is 0, the distribution is Uniform; otherwise, the distribution is Zipf. The number of dimensions varies from 100 to 30K. The default number of dimensions is set to 100 and 30K to represent low and high dimensionality, respectively. We vary the dimension cardinality from 3 to 1K. We vary the average subscription size from 5 to 30 and the event size from 30 to 130. The matching probability varies from 0.1% to 50%. The equality operator ratio varies from 0% to 100% to represent different predicate selectivities. Compared with the related approaches, our workloads are comprehensive, thereby exploring a richer parameter space. For example, the workloads used by `OpIndex` all follow the Uniform distribution, and the number of subscriptions increases to only 1M in `BE-Tree`.

Table 1: Parameters of the Synthetic Datasets

Subscription Number	300K, **1M**, 3M, 10M, 30M, 100M
The α in Zipf	**0**, 1, 2, 3, 4, 5
Dimension Number	**100**, 300, 1K, 3K, 10K, **30K**
Dimension Cardinality	3, 10, 30, **100**, 300, 1K
Avg. Subscription Size	**5**, 10, 15, 20, 25, 30
Avg. Event size	**30**, 42, 54, 66, 78, 90
Matching Probability	**0.001**, 0.005, 0.01, 0.05, 0.1, 0.5
Equal. Operator Ratio	0, **0.2**, 0.4, 0.6, 0.8, 1.0

The second synthetic dataset uses the query logs from the SIGMOD 2013 contest to represent keyword-based subscriptions [1]. We transform a query into a Boolean expression whereby each keyword is treated as an equality predicate. If a keyword has more than six characters, we transform it into a predicate using the first three characters as the attribute name and the next three characters as the attribute value. For example, "boolean" is transformed into $\{boo, =, \text{"}lea\text{"}\}$. Otherwise, a keyword is transformed into a predicate using the first half of its characters as the attribute name and the remaining characters as the attribute value. For example, "vldb" is transformed into $\{vl, =, \text{"}db\text{"}\}$. This transformation results in Boolean expressions in a space of 17,577 dimensions. The document dataset is transformed into events using a similar method.

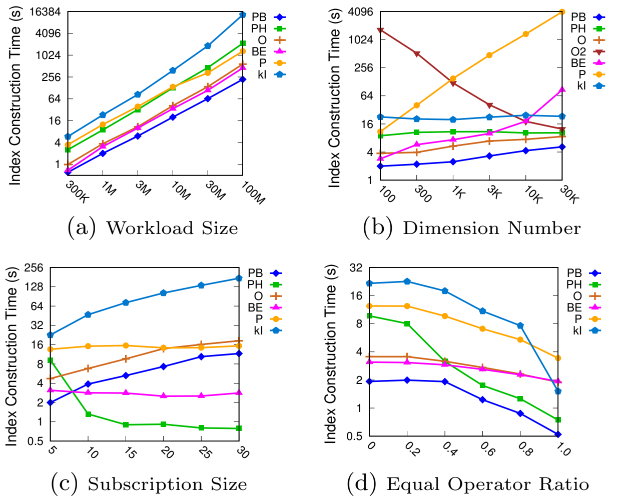
In addition to these synthetic datasets, we also derived a real-world workload based on a display ads dataset of an online shopping site for subscriptions and events. When a user visits the site, product advertisements are shown to the user. In the backend advertisement inventory, an advertisement specifies conditions to promote products to users. The conditions include channel (e.g., mobile, PC, or tablet), region (e.g., CA, DE, or CN), ad position, etc. By translating conditions into predicates, we model advertisements as subscriptions. When a user interacts with the website (e.g., surfs or logs in), the user's session is bound to a set of attributes such as the login channel, the login region and the user's profile. By translating the profile and attributes into attribute-value pairs, we model each session as an event. For example, if a user is male, the resulting event contains an attribute-value pair $\langle gender, male \rangle$.

7.2 Experiments on Synthetic Workloads

The first set of experiments was conducted on the synthetic datasets. We first report on the index construction time. Then, we evaluate the matching performance with respect to workload size, distribution, number of dimensions, etc. Finally, the memory use of each index is reported.

7.2.1 *Index Construction Time*

Our experiments show that not only workload size but also the number of dimensions, subscription size, and equality operator ratio affect the index construction time. As shown in Fig. 5(a), all the algorithms' index construction times increase with the number of subscriptions. Among the algorithms shown, `PSTBloom` exhibits the lowest index construction time: when there are up to 100M subscriptions, compared with the next-best algorithm `BE-Tree`, `PSTBloom` reduces the index construction time by 78%.



(a) Workload Size (b) Dimension Number

(c) Subscription Size (d) Equal Operator Ratio

Figure 5: Index Construction Time

Fig. 5(b) shows how the number of dimensions affects the index construction time. As shown, `BE-Tree` and `Propagation` increase quickly with the number of dimensions, whereas `PSTBloom`, `PSTHash` and `k-index` are not sensitive to the number of dimensions. For `OpIndex`, after each subscription is inserted, its index needs to be reordered before event matching can resume. An optimization adopted by `OpIndex` is to sort its index after all subscriptions are inserted. The light brown line shows the index construction time of `OpIndex` when this optimization is utilized. However, when the arrivals of subscriptions and events overlaps, `OpIndex` cannot operate in this manner. The brown line shows the index

[4]The authors of some related approaches kindly provided the source code of their implementations [33, 39, 30]. For consistency, we reimplemented `OpIndex` and `REIN` in C because the original versions were written in C++.

construction time of OpIndex when the index is kept up to date after each subscription is inserted. As shown, for dense workloads (when the number of dimensions is small), the index construction time of OpIndex is three orders of magnitude larger than that of PSTBloom, BE-Tree, and PSTHash.

Fig. 5(c) shows how the average subscription size affects the index construction time. The index construction time of PSTBloom, OpIndex and k-index increases with the subscription size. BE-Tree and Propagation are not sensitive to subscription size, whereas the index construction time of PSTHash decreases. PSTHash possesses this advantage because it builds indexes only on access predicates. When there are more predicates in each subscription, access predicates with high selectivity are more likely to be selected.

The effect of the equality operator ratio on the index construction time is shown in Fig. 5(d). All algorithms show a decrease in the index construction time when the equality operator ratio increases. PSTHash decreases most quickly because a higher equality operator ratio results in fewer predicate spaces being covered by the inserted predicate. PSTBloom achieves the lowest index construction time. For a similar reason, the advantage of PSTBloom is more obvious when the equality operator ratio is high.

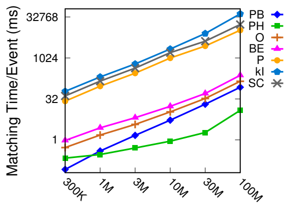
7.2.2 *Matching Time*

The matching time is among the most important metrics for Boolean expression matching algorithms. In this section, we present extensive experiments under a variety of controlled conditions. In particular, the number of dimensions is a distinguishing factor among matching algorithms. For each controlled condition, we ran two experiments: (1) with 100 dimensions and (2) with 30K dimensions. Since the default number of subscriptions is 1M, these settings represent dense and sparse workloads, respectively.

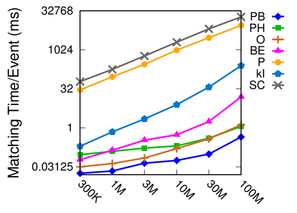
Workload Size: We consider the matching time as we increase the number of subscriptions processed. As illustrated by Fig. 6, all algorithms scale linearly with respect to the number of subscriptions. Among them, PSTHash increases slowest, especially for dense workloads. In Fig. 6(a), when there are 300K subscriptions, PSTBloom performs best. Compared with OpIndex, PSTBloom reduces the matching time by 84%. When the number of subscriptions is greater than 1M, PSTHash performs best. When there are up to 100M subscriptions, PSTHash reduces the matching time by 92% compared to OpIndex. In Fig. 6(b), PSTHash performs as well as OpIndex when the number of subscriptions increases to 30M, and PSTBloom always performs best.

Workload Distribution: Workload distribution is another distinguishing factor among matching algorithms. In Fig. 7, the workload distribution is $P(r) = \frac{C}{r^\alpha}$. When α is 0, the distribution is Uniform; when α is greater than 0, the distribution is Zipf. Given a Zipf distribution, a few popular dimensions are associated with a large number of subscriptions, resulting in dense workloads. As shown, under the Zipf distribution, PSTHash performs best regardless of the number of dimensions evaluated (here, 100 or 30K). An interesting finding is that OpIndex outperforms BE-Tree under the Uniform distribution; however, BE-Tree outperforms OpIndex under the Zipf distribution because the index retrieval time of OpIndex increases when there are a large number of subscriptions associated with a few popular dimensions. For a similar reason, under the Zipf distribution, k-index performs even worse than SCAN.

Number of Dimensions: As shown in Fig. 8(a), when the number of subscriptions is fixed and the number of dimensions increases, the matching times of all algorithms decrease, except for those of Propagation and SCAN. Compared to PSTHash and BE-Tree, the matching times of PSTBloom, OpIn- dex and k-index decrease quickly. Intuitively, these three algorithms are more suitable for high-dimensional workloads. However, the prerequisite is that the number of subscriptions does not increase simultaneously. In Fig. 8(b), we increase the number of dimensions while keeping the number of subscriptions per dimension fixed. As shown, PSTBloom, OpIndex and k-index are no longer sensitive to the number of dimensions. By combining the findings of these two experiments, we observe that PSTBloom, OpIndex and k-index are more suitable for sparse workloads. In this experiment, when the number of dimensions is greater than 300, PSTBloom always achieves the best performance.

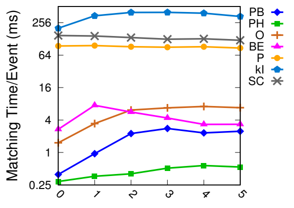


(a) 100 Dims

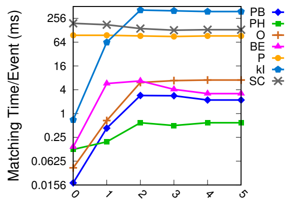


(b) 30K Dims

Figure 6: Varying Workload Size on the X-axes

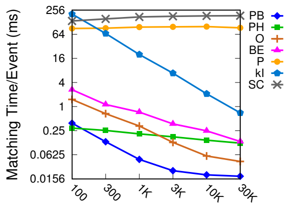


(a) 1M Subs and 100 Dims

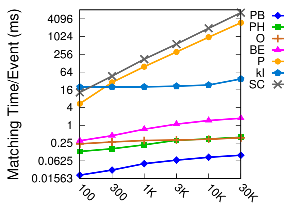


(b) 1M Subs and 30K Dims

Figure 7: Varying Workload Distribution on the X-axes



(a) 1M Subs

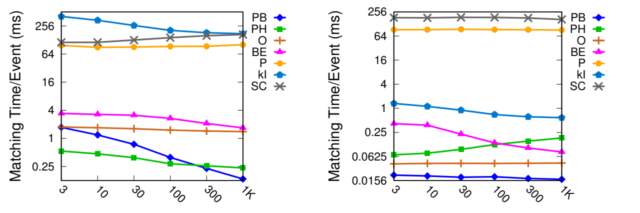


(b) 1K Subs/Dim

Figure 8: Varying Number of Dimensions on the X-axes

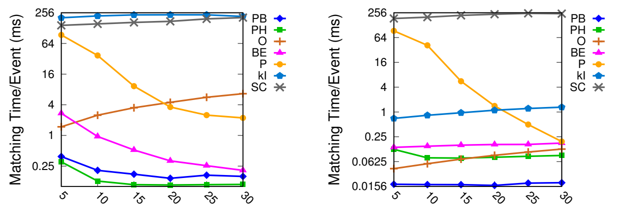
Dimension Cardinality: Fig. 9 shows how the dimension cardinality affects the matching time. As shown in Fig. 9(a), PSTHash performs best when the dimension cardinality is less than 300, while PSTBloom performs best for higher-dimension cardinalities. For example, when the dimension cardinality is 1K, compared to that of OpIndex, the matching time of PSTBloom is reduced by up to 91%.

Subscription Size: Another important workload characteristic is the subscription size. As shown in Fig. 10(a), given dense workloads, PSTBloom, PSTHash, BE-Tree, and Propagation all present lower event matching times as the average subscription size increases because these algorithms select predicates with high selectivity to prune subscriptions. When there are more predicates in a subscription, predicates with high selectivity are more likely to be found. Under sparse workloads, the effect of subscription size is not obvious, except for `Propagation`. In these experiments, only `OpIndex` performs worse as the average subscription size increases. When the average subscription size increases from 5 to 30, the matching time of `OpIndex` increases by 350% and 200% for dense and sparse workloads, respectively, because the index size of `OpIndex` increases linearly with the total number of predicates. Larger subscription sizes result in larger index scanning costs.



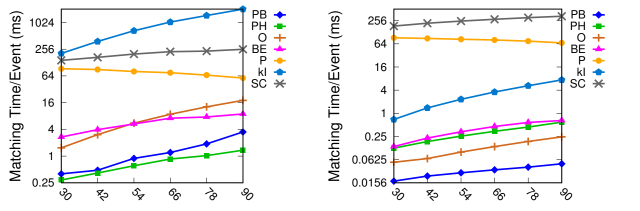
(a) 1M Subs and 100 Dims (b) 1M Subs and 30K Dims

Figure 9: Varying Dimension Cardinality on the X-axes



(a) 1M Subs and 100 Dims (b) 1M Subs and 30K Dims

Figure 10: Varying Subscription Size on the X-axes



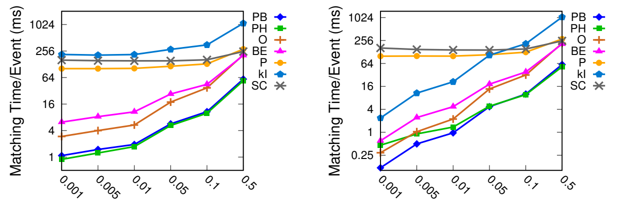
(a) 1M Subs and 100 Dims (b) 1M Subs and 30K Dims

Figure 11: Varying Event Size on the X-axes

Event Size: When the average number of attribute-value pairs of an event increases, there are more candidate subscriptions. As shown in Fig. 11, under both dense and sparse workloads, all algorithms, except for `Propagation` and `SCAN` show higher matching times as the event size increases. Given dense workloads, when the average event size increases to 90, `PSTHash` performs best. Compared to `BE-Tree`, `PSTHash` reduces the matching time by 85%. Given sparse workloads, when the average event size increases to 90, `PSTBloom` performs best. Compared to `OpIndex`, `PSTBloom` reduces the matching time by 80%.

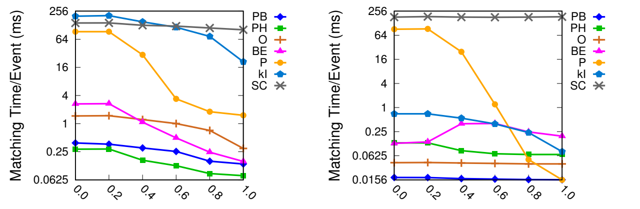
Matching Probability: We refer to the *matching probability* as the expected ratio of subscriptions that match for a given event. A higher matching probability means that more subscriptions match. In Fig. 12, with the exception of `Propagation` and `SCAN`, the algorithms' matching times increase with the matching probability under both dense and sparse workloads. When the matching probability increases to 50%, `BE-Tree`, `OpIndex` and `Propagation` show similar matching times as that of `SCAN`. In contrast, both `PSTBloom` and `PSTHash` need only approximately 25% of this matching time for both dense and sparse workloads.

Equality Operator Ratio: In this experiment, we study the effect of the ratio of equality vs. nonequality predicates per subscription. Fig. 13 shows that the general trend is that the matching time of all algorithms decreases as the percentage of equality predicates increases. Most notably, when subscriptions consist of only equality predicates, `Propagation` achieves a substantial performance gain and is as good as `PSTBloom` under sparse workloads. Given dense workloads, the matching performance of `PSTHash` always ranks first. Moreover, when the equality operator ratio is higher, the advantage of `PSTHash` becomes more obvious.



(a) 1M Subs and 100 Dims (b) 1M Subs and 30K Dims

Figure 12: Varying Matching Probability on the X-axes



(a) 1M Subs and 100 Dims (b) 1M Subs and 30K Dims

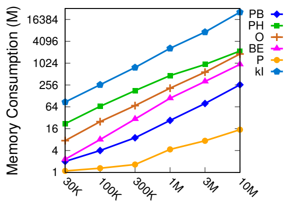
Figure 13: Varying Equal Operator Ratio on the X-axes

7.2.3 *Memory Consumption*

All indexes considered in this paper are memory resident. Here, we evaluate memory use. To accurately report the memory consumption of each index, we calculate the memory use of the runtime processes before and after all subscriptions are inserted into each index. Fig. 14(a) shows the memory use as the number of subscriptions increases. Unsurprisingly, all algorithms require more memory when there are more subscriptions. However, the memory use of `PSTBloom` and `PSTHash` increases slower than that of `OpIndex` and `BE-Tree` because the number of predicate spaces maintained in `PS-Trees` increases slower than the number of subscriptions. Fig. 14(b) shows the memory use as the average subscription size increases: `OpIndex` and `k-index` require more memory, `BE-Tree` and `Propagation` remain stable, and `PSTBloom` and `PSTHash` need less memory. In these two experiments, `Propagation` needs the least amount of memory. Compared with `BE-Tree` and `OpIndex`, `PSTBloom` reduces memory use by up to 94% and 99%, respectively.

7.3 Experiments on Query Logs

In this synthetic dataset, which stems from query logs, there are 2.1M subscriptions in a space of 17,577 dimensions. On average, each subscription contains 2.78 predicates, and each event contains 93.07 attribute-value pairs. As shown in Fig. 15(a), under this high-dimensional workload, the matching performance ranking is in the following order: `PSTBloom`, `OpIndex`, `PSTHash`, `BE-Tree`, `k-index`, `Propagation`, and `SCAN`. `OpIndex`, `PSTHash`, and `BE-Tree` have similar matching latencies, and PSTBloom performs considerably better than these three algorithms. Fig. 15(b) shows the index construction time. The ranking of the index construction time is the same as that of the matching time. The difference is that PSTBloom and BE-Tree have similar index construction times as OpIndex and k-index, respectively.


(a) Varying Workload Size

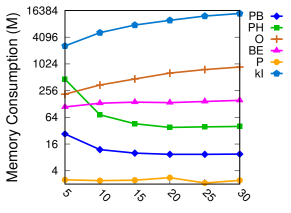
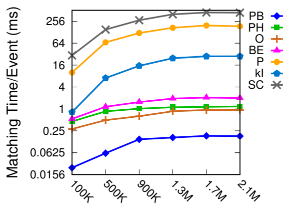

(b) Varying Sub Size

Figure 14: Memory Consumption


(a) Matching Time

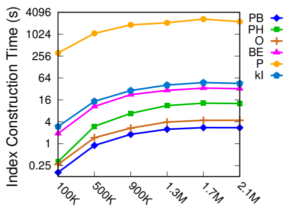

(b) Construction Time

Figure 15: Varying Workload Size on the X-axes

7.4 Experiments on the Ads Dataset

By transforming advertisements into subscriptions, in this workload, we obtain 3M subscriptions. The number of predicates in a subscription ranges from 1 to 56. On average, each subscription contains 8 predicates, and each event contains 20 attribute-value pairs. The number of dimensions is 122, which means that this workload is a dense workload.

As shown in Table 2, under this workload, PSTHash achieves the best matching performance, followed by PSTBloom. The matching performance of BE-Tree is a little better than that of OpIndex. Compared with BE-Tree, PSTHash reduces the matching time by 89%. PSTBloom achieves the shortest index construction time. Moreover, compared with BE-Tree and OpIndex, PSTBloom needs less memory. This experimental result is roughly consistent with the experimental results on synthetic workloads.

Table 2: Experiments on the Ads Dataset

Index	Matching	Construction	Memory
SCAN	435.27 ms	-	-
k-index	616.42 ms	140.76 s	7.32 GB
Propagation	168.73 ms	62.43 s	14.41 MB
OpIndex	3.21 ms	24.92 s	574.38 MB
BE-Tree	2.26 ms	13.70 s	393.21 MB
PSTBloom	0.82 ms	7.81 s	55.93 MB
PSTHash	0.24 ms	8.22 s	759.72 MB

7.5 Interval-Tree, Segment-Tree, and PS-Tree

PS-Tree, more precisely, PS-Tree$_B$, can be interpreted to store intervals and allows querying the stored intervals that contain a given point. Interval-Tree [10] and Segment-Tree [11] are two index structures that provide similar capabilities. Thus, both represent approaches related to PS-Tree. Here, we conduct the following comparative evaluations.

Table 3: PSTree Querying Performance

Index	Querying	Construction	Memory
SCAN	23.11 s	-	-
Interval-Tree	2.52 s	16.02 ms	1.61 MB
Segment-Tree	6.27 ms	29.71 ms	6.86 MB
PS-Tree	0.71 ms	91.88 ms	54.13 MB

In this group of experiments, we compare the query time, index construction time and memory consumption of PS-Tree, Segment-Tree and Interval-Tree. SCAN, which represents the naive scanning method, is used as a baseline. Table 3 shows the experimental results when there are 100K intervals and 100K query points. As shown in this table, although PS-Tree exhibits higher index construction time and memory use, its query performance is the best. Compared to Interval-Tree and Segment-Tree, PS-Tree reduces the matching time by 99.97% and 89%, respectively, which suggests that PS-Tree is more suitable for Boolean expression matching, where the number of queries is much higher than the number of intervals. Moreover, PS-Tree supports more operators, such as "$\in$" and "$>$", which are not supported by Segment-Tree and Interval-Tree. Another advantage of PS-Tree over Segment-Tree and Interval-Tree is that the PSTHash algorithm can only be supported by PS-Tree.

8. CONCLUSIONS

In this paper, we proposed a new index, PS-Tree, which efficiently constructs a many-to-many relationship between predicate spaces and subscriptions. Through PS-Tree, the problem of predicate matching is transformed into a problem of locating the predicate space to which an attribute-value pair belongs. PS-Tree offers excellent query performance and good expressiveness.

Based on PS-Tree, we first propose the PSTBloom algorithm. PSTBloom selects a predicate with high selectivity as the access predicate for each subscription. Then, the subscription is associated with its corresponding leaf nodes. Through PS-Tree, PSTBloom can efficiently filter out all the subscriptions whose access predicates do not match with a received event. Then, Bloom filter signatures are used to further filter out most unmatching subscriptions. PSTBloom is efficient at handling many workload distributions, especially high-dimensional workloads. PSTBloom achieves fast index construction and requires a small amount of memory. However, PSTBloom and other algorithms do not meet the challenge presented by dense workloads. To overcome this limitation, we further propose the PSTHash algorithm. PSTHash selects more than one access predicate for each subscription and constructs a many-to-many relationship between multidimensional predicate spaces and subscriptions. Only when an event matches with all access predicates of a subscription is the subscription identified as a candidate subscription. Compared with PSTBloom and other existing algorithms, PSTHash achieves the best matching performance for dense workloads.

We conducted extensive experiments using both synthetic and real-world datasets. The results show that our algorithms outperform state-of-the-art approaches for both high-dimensional and dense workloads.

9. ACKNOWLEDGMENTS

This work was in part supported by the Alexander von Humboldt foundation and NSERC.

10. REFERENCES

[1] http://sigmod.kaust.edu.sa, Jan. 2013.

[2] https://newsroom.fb.com/company-info, march 2018.

[3] M. K. Aguilera, R. E. Strom, D. C. Sturman, M. Astley, and T. D. Chandra. Matching events in a content-based subscription system. In *ACM PODC*, pages 53–61, 1999.

[4] L. Arge and J. S. Vitter. Optimal dynamic interval management in external memory. In *IEEE FOCS*, pages 560–569, 1996.

[5] K. Beyer, J. Goldstein, R. Ramakrishnan, and U. Shaft. When is "nearest neighbor" meaningful? In *Springer ICDT*, pages 217–235, 1999.

[6] A. Campailla, S. Chaki, E. Clarke, S. Jha, and H. Veith. Efficient filtering in publish-subscribe systems using binary decision diagrams. In *IEEE ICSE*, pages 443–452, 2001.

[7] C. Cañas, K. Zhang, B. Kemme, J. Kienzle, and H.-A. Jacobsen. Publish/subscribe network designs for multiplayer games. In *ACM Middleware*, pages 241–252, 2014.

[8] S. Ceri, R. Cochrane, and J. Widom. Practical applications of triggers and constraints: Success and lingering issues. In *PVLDB*, pages 254–262, 2000.

[9] B. Chandramouli and J. Yang. End-to-end support for joins in large-scale publish/subscribe systems. *PVLDB*, 1(1):434–450, 2008.

[10] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to algorithms*. MIT Press, 2009.

[11] M. De Berg, M. Van Kreveld, M. Overmars, and O. C. Schwarzkopf. Computational geometry. In *Computational geometry*, pages 1–17. Springer, 2000.

[12] R. De La Briandais. File searching using variable length keys. In *Western joint computer conference*, pages 295–298. ACM, 1959.

[13] Y. Diao, M. Altinel, M. J. Franklin, H. Zhang, and P. Fischer. Path sharing and predicate evaluation for high-performance xml filtering. *ACM TODS*, pages 467–516, 2003.

[14] F. Fabret, H. A. Jacobsen, F. Llirbat, J. Pereira, K. A. Ross, and D. Shasha. Filtering algorithms and implementation for very fast publish/subscribe systems. In *ACM SIGMOD*, pages 115–126, 2001.

[15] W. Fan, Y. Liu, and B. Tang. Gem: An analytic geometrical approach to fast event matching for multi-dimensional content-based publish/subscribe services. In *IEEE INFOCOM*, pages 1–9, 2016.

[16] E. Fidler, H.-A. Jacobsen, G. Li, and S. Mankovski. The padres distributed publish/subscribe system. In *FIW*, pages 12–30, 2005.

[17] M. Fontoura, S. Sadanandan, J. Shanmugasundaram, S. Vassilvitski, E. Vee, S. Venkatesan, and J. Zien. Efficiently evaluating complex boolean expressions. In *ACM SIGMOD*, pages 3–14, 2010.

[18] A. Guttman. R-trees: A dynamic index structure for spatial searching. In *ACM SIGMOD*, 1984.

[19] E. N. Hanson, C. Carnes, L. Huang, M. Konyala, L. Noronha, S. Parthasarathy, J. Park, and A. Vernon. Scalable trigger processing. In *IEEE ICDE*, pages 266–275, 1999.

[20] S. Hou and H.-A. Jacobsen. Predicate-based filtering of xpath expressions. In *IEEE ICDE*, page 53, 2006.

[21] S. Ji and H.-A. Jacobsen. PS-Tree-Based efficient boolean expression matching for high-dimensional and dense workloads. In *Technical Report MSRG (Extended PVLDB paper version)*. http://www.msrg.org/papers/PS-Tree, November, 2018.

[22] S. Ji, C. Ye, J. Wei, and H.-A. Jacobsen. MERC: Match at edge and route intra–cluster for content-based publish/subscribe systems. In *ACM Middleware*, pages 13–24, 2015.

[23] G. Li, S. Hou, and H.-A. Jacobsen. Routing of xml and xpath queries in data dissemination networks. In *IEEE ICDCS*, pages 627–638, 2008.

[24] G. Li, V. Muthusamy, and H.-A. Jacobsen. A distributed service-oriented architecture for business process execution. *ACM TWEB*, page 2, 2010.

[25] K.-I. Lin, H. V. Jagadish, and C. Faloutsos. The tv-tree: An index structure for high-dimensional data. *The VLDB Journal*, 3(4):517–542, 1994.

[26] H. Liu, V. Ramasubramanian, and E. G. Sirer. Client behavior and feed characteristics of rss, a publish-subscribe system for web micronews. In *Proceedings of the 5th ACM SIGCOMM conference on Internet Measurement*, pages 3–3. USENIX Association, 2005.

[27] A. Machanavajjhala, E. Vee, M. Garofalakis, and J. Shanmugasundaram. Scalable ranked publish/subscribe. *PVLDB*, 1(1):451–462, 2008.

[28] A. Margara and G. Cugola. High-performance publish-subscribe matching using parallel hardware. *IEEE TPDS*, pages 126–135, 2014.

[29] D. R. Morrison. Patricia—practical algorithm to retrieve information coded in alphanumeric. *Journal of the ACM*, pages 514–534, 1968.

[30] S. Qian, J. Cao, Y. Zhu, and M. Li. Rein: A fast event matching approach for content-based publish/subscribe systems. In *IEEE INFOCOM*, pages 2058–2066, 2014.

[31] S. Qian, J. Cao, Y. Zhu, M. Li, and J. Wang. H-tree: An efficient index structurefor event matching in content-basedpublish/subscribe systems. *IEEE TPDS*, pages 1622–1632, 2015.

[32] M. Sadoghi, I. Burcea, and H.-A. Jacobsen. Gpx-matcher: a generic boolean predicate-based xpath expression matcher. In *ACM EDBT*, pages 45–56, 2011.

[33] M. Sadoghi and H.-A. Jacobsen. Be-tree: an index structure to efficiently match boolean expressions over high-dimensional discrete space. In *ACM SIGMOD*, pages 637–648, 2011.

[34] M. Sadoghi and H.-A. Jacobsen. Analysis and optimization for boolean expression indexing. *ACM TODS*, page 8, 2013.

[35] M. Sadoghi and H.-A. Jacobsen. Adaptive parallel compressed event matching. In *IEEE ICDE*, pages 364–375, 2014.

[36] M. Sadoghi, M. Jergler, H.-A. Jacobsen, R. Hull, and R. Vaculin. Safe distribution and parallel execution of data-centric workflows over the publish/subscribe

abstraction. *IEEE TKDE*, pages 2824–2838, 2015.

[37] P. Weiner. Linear pattern matching algorithms. In *Switching and Automata Theory, IEEE Annual Symposium on*, pages 1–11, 1973.

[38] S. E. Whang, H. Garcia-Molina, C. Brower, J. Shanmugasundaram, S. Vassilvitskii, E. Vee, and R. Yerneni. Indexing boolean expressions. *PVLDB*, 2(1):37–48, 2009.

[39] D. Zhang, C.-Y. Chan, and K.-L. Tan. An efficient publish/subscribe index for e-commerce databases. *PVLDB*, 7(8):613–624, 2014.

[40] Y. Zhao and J. Wu. Towards approximate event processing in a large-scale content-based network. In *IEEE ICDCS*, pages 790–799, 2011.