



# A-Tree: A Dynamic Data Structure for Efficiently Indexing Arbitrary Boolean Expressions

Shuping Ji

State Key Lab of Computer Science, ISCAS  
sji@otcaix.iscas.ac.cn

Hans-Arno Jacobsen

University of Toronto  
jacobsen@eecg.toronto.edu

## ABSTRACT

Efficiently evaluating a large number of arbitrary Boolean expressions is needed in many applications such as advertising exchanges, complex event processing, and publish/subscribe systems. However, most solutions can support only conjunctive Boolean expression matching. The limited number of solutions that can directly work on arbitrary Boolean expressions present performance and flexibility limitations. Moreover, normalizing arbitrary Boolean expressions into conjunctive forms and then using existing methods for evaluating such expressions is not effective because of the potential exponential increase in the size of the expressions. Therefore, we propose the A-Tree data structure to efficiently index arbitrary Boolean expressions. A-Tree is a multirooted tree, in which predicates and subexpressions from different arbitrary Boolean expressions are aggregated and shared. A-Tree employs dynamic self-adjustment policies to adapt itself as the workload changes. Moreover, A-Tree adopts different event matching optimizations. Our comprehensive experiments show that A-Tree-based matching outperforms existing arbitrary Boolean expression matching algorithms in terms of memory use, matching time, and index construction time by up to 71%, 99% and 75%, respectively. Even on conjunctive expression workloads, A-Tree achieves a lower matching time than *state-of-the-art* conjunctive expression matching algorithms.

## ACM Reference Format:

Shuping Ji and Hans-Arno Jacobsen. 2021. A-Tree: A Dynamic Data Structure for Efficiently Indexing Arbitrary Boolean Expressions. In *Proceedings of the 2021 International Conference on Management of Data (SIGMOD '21)*, June 20–25, 2021, Virtual Event, China. ACM, New York, NY, USA, 13 pages. <https://doi.org/10.1145/3448016.3457266>

## 1 INTRODUCTION

In this paper, we study the problem of matching events, a set of attribute-value pairs, against a very large number of arbitrary Boolean expressions. This matching problem is the core of many applications, such as online advertising [22, 36, 57], online news dissemination [35], automatic user targeting [19], complex event processing (CEP) [21, 53], and content-based routing and publish/subscribe [16, 18, 31, 34]. The matching problem is also vital to systems and infrastructures developed in industry, such as those

developed at Yahoo! [19, 20], Microsoft [9] Google [54], IBM [1], Oracle [13, 48] and Amazon [4].

Developing solutions to the Boolean expression matching problem, which is often also referred to as *publish/subscribe matching* or *event filtering*, is of great relevance. The bulk of the effort over the past 25 years has focused on matching conjunctive Boolean expressions (for example, see [1, 3, 16, 27, 30, 47, 52, 55, 56]). However, many applications also require support for arbitrarily complex Boolean expressions, as they allow greater expressiveness. Advertising exchanges [19], CEP [7, 50] and predictive search [20, 54] constitute examples. We offer two scenarios to further illustrate these applications.

**Ad Exchange** – An advertising exchange is an electronic hub that connects online publishers to advertisers via intermediaries. An ad exchange can be represented as a directed graph, where the nodes are publishers, advertisers and intermediaries, and edges represent ad sourcing relationships. Each edge is annotated with a Boolean expression that restricts the set of user visits that can be selected via the edge. When a user visits a site, the visit is represented as an attribute-value assignment. The goal is to identify all ad campaigns that are satisfied by the visit such that the best ad to serve can be determined. That is, the exchange must evaluate a very large set of arbitrary Boolean expressions to determine which expression satisfies the given assignment of attributes to values (events).

**System Monitoring** – In online system monitoring, an operator is interested in errors or critical log messages created by the infrastructure. The operator would issue many filtering expressions, such as *source="mobile"* and *(type="error" or level="critical")*.

Both scenarios could be tackled by CEP engines as they do perform event matching. However, CEP engines are more general and perform many other operations that have no role in pure event matching. Today, CEP engines are designed to support the general case but are not optimized for the event matching case, which is predominantly required for stateless pub/sub and event matching.

Designing an efficient arbitrary Boolean expression (ABE) matching solution is challenging for at least four main reasons. First, the solution must scale to a large set of ABEs. Consider online display advertising as an example; there can be tens of millions of display advertisements, a.k.a., ABEs. Second, the solution should be efficient at handling a high rate of arriving events on the premise of a low matching time. Third, compared to conjunctive Boolean expression matching, ABE matching is more challenging because an arbitrary expression may contain not only *and* but also *or*, *not*, and other logical operators, resulting in a complexly structured expression. Fourth, fast index construction and dynamic index updates must be supported to accommodate changing customer interests and event workloads.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from [permissions@acm.org](mailto:permissions@acm.org).

SIGMOD '21, June 20–25, 2021, Virtual Event, China

© 2021 Association for Computing Machinery.

ACM ISBN 978-1-4503-8343-1/21/06...\$15.00

<https://doi.org/10.1145/3448016.3457266>

To the best of our knowledge, currently, there are only four algorithms that can directly work on ABEs: Dewey ID [19], Interval ID [19], BoP [6] and BDD [9]. These solutions exhibit limitations that affect their performance and flexibility. The Dewey ID and Interval ID approaches separately encode and evaluate different Boolean expressions. Although these approaches are efficient at evaluating a single expression, they are not efficient at concurrently evaluating a large number of expressions. The BoP method is an extension of the count-based conjunctive Boolean expression matching algorithms (see for example [3, 16].) However, BoP does not achieve a comparable matching time since filtering non-matching expressions based on the minimum number of matching predicates is inefficient for ABEs. BoP also separately encodes different expressions and does not exploit sharing. In principle, BDD could support ABEs. However, this support was neither discussed nor verified [9]. In this paper, we are the first researchers to adapt the BDD approach to match arbitrary expressions. Unfortunately, our experiments show that BDD consumes a substantial amount of memory and exhibits a high matching time. A BDD is a versatile data structure with flexibility that is not required for event matching; a large number of distinct predicates increase its representational complexity.

Many conjunctive Boolean expression matching algorithms have been proposed [1, 3, 9, 16, 30, 44–46, 52, 56]. A simple matching approach would consist of translating each ABE into a set of conjunctive Boolean expressions and then resort to conjunctive Boolean expression matching. However, this approach may result in an exponential increase in the size of the resulting conjunctive expressions due to normalization [51]. In addition, the translation introduces memory, matching, and maintenance costs. Our experiments on both synthetic workloads and real-world workloads show that this method is not effective.

**Table 1: A-Tree vis-a-vis Alternatives**

Alg.	Share	Index	Adjust	Filter	Blowup
Dewey	No	Compacted	No	No	No
Interval	No	Compacted	No	No	No
BoP	No	Compacted	No	Count	No
BDD	Yes	Binary Tree	No	No	Yes
Trans.	N/A	N/A	N/A	N/A	Yes
SCAN	No	No	No	No	No
A-Tree	Yes	Multitree	Yes	Access	No

To overcome these limitations, we propose the *aggregate tree* (A-Tree) data structure to efficiently index ABEs. The basic idea of A-Tree is to represent an ABE as an  $n$ -ary tree and then combine these  $n$ -ary trees into an aggregated tree with multiple roots, which can also be regarded as a directed acyclic graph. We observe that different expressions often contain many common predicates and subexpressions. Moreover, entirely duplicated expressions could be issued by different subscribers. For simplicity of presentation, we refer to *subexpression sharing* to represent the sharing of predicates, subexpressions and expressions. The four arbitrary expression matching approaches do not effectively identify and share common subexpressions [6, 9, 19, 19]. In contrast, A-Tree is explicitly designed to share common subexpressions to reduce

memory and matching time. Moreover, A-Tree supports dynamic self-adjustment, it does not need to be built in advance, and it supports expression insertion and deletion, which are properties not shared by most alternatives.

Table 1 summarizes the advantages of A-Tree over alternative approaches. In this table, Scan represents the naive one-by-one scan method adopted by many CEP engines. A-Tree exhibits several advantages over prominent alternatives: (1) A-Tree efficiently identifies and shares common subexpressions; (2) instead of simply encoding each expression into a compacted format, A-Tree employs a multirooted tree data structure, which is more flexible; (3) only A-Tree supports dynamic index adjustment; (4) A-Tree filters unmatching expressions by access predicates and subexpressions in a bottom-to-top manner, and thus, identifies much fewer expressions as potentially matching candidates per event; and (5) in contrast to the Translation and BDD approaches, A-Tree is not susceptible to an exponential increase in size.

In summary, our contributions in this paper are three-fold: (1) We introduce the A-Tree data structure to efficiently index ABEs: A-Tree uniquely represents shared subexpression by a single node to reduce memory and improve matching performance. A-Tree supports expression reorganization and index self-adjustment to optimize the index. (2) We introduce the A-Tree-based event matching algorithm together with *zero suppression filter* and *propagation on demand* optimizations that reduce matching time. (3) We provide a comprehensive experimental analysis: Compared to existing ABE matching algorithms, A-Tree reduces memory, matching time, and index construction time by a maximum of 71%, 99% and 75%, respectively.

The remainder of this paper is organized as follows. In Sec. 2, we review related work. Sec. 3 presents our expression language and matching semantics. The A-Tree data structure, the index construction and an analysis of its properties are given in Sec. 4. Event matching and optimizations are presented in Sec. 5. Our experimental results are discussed in Sec. 6.

## 2 RELATED WORK

Problems related to indexing Boolean expressions have been widely investigated in many contexts, such as query compilers [2, 15, 29], active databases [26, 40], trigger processing [11, 25], stream processing and CEP [21, 23, 38, 42, 53], XPath/XML matching [14, 28, 32, 33, 43], and pub/sub-style event matching [16, 44, 45, 52, 56].

We concentrate on event matching since other contexts either use different languages, different processing models, or do not aim to scale to tens of millions of simultaneously evaluated expressions. For example, in the context of CEP, efficient matching against a large number of expressions has not been the design goal. For example, FlinkCEP [10] even requires programmers to write a class to define a filtering expression, and then it processes these expressions one-by-one which is not scalable.

Here, we categorize related event matching work into four categories: scan-based solutions, count-based solutions, tree-based solutions and translation-based solutions.

Given an event, the naive matching method scans every ABE and matches it against the event. This method is inefficient when there are a large number of expressions. The Dewey ID [19] and Interval ID [19] methods are variants of this scan-based method with

two optimizations. First, an ABE is represented as a set of Dewey IDs [39, 49] or Interval IDs to reduce memory usage and improve evaluation performance. Second, for each event, the matching ABEs are retrieved by the matching Dewey IDs or Interval IDs. If matching Dewey IDs or Interval IDs do not exist, then the corresponding ABE is not evaluated.

The Dewey ID and Interval ID methods work in two phases. The first phase annotates each ABE as a set of Dewey IDs or Interval IDs offline. Each ID corresponds to a leaf node. As shown in Figure 1, the expression  $(P_1 \vee P_2 \vee P_3) \wedge P_4 \wedge (P_5 \vee P_6)$  is annotated as the Dewey IDs 1.1, 1.2, 1.3, 2, 3\*.1, and 3\*.2 or Interval IDs  $\langle 1,4 \rangle$ ,  $\langle 1,4 \rangle$ ,  $\langle 1,4 \rangle$ ,  $\langle 5,5 \rangle$ ,  $\langle 6,10 \rangle$ , and  $\langle 6,10 \rangle$ , and these IDs correspond to the leaf nodes annotated by  $P_1, P_2, P_3, P_4, P_5$  and  $P_6$ . The second phase is the expression evaluation phase. Existing predicate matching solutions are used to retrieve the matching predicates. In this example, assume that predicates  $P_1, P_4$  and  $P_5$  are matching. The corresponding Dewey IDs 1.1, 2, and 3\*.1 or Interval IDs  $\langle 1,4 \rangle$ ,  $\langle 5,5 \rangle$ , and  $\langle 6,10 \rangle$  are applied to compute whether the ABE is matching.

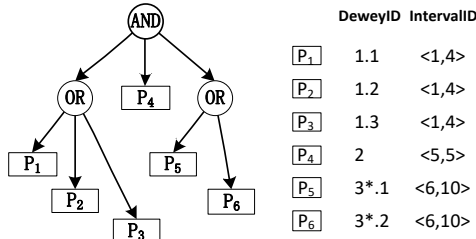


Figure 1: Dewey ID and Interval ID Example

The Dewey ID and Interval ID methods encode Boolean expressions. However, different expressions are stored and evaluated separately. Thus, shared subexpressions are evaluated multiple times for a single event, which limits performance. Moreover, both methods are not efficient at pruning nonmatching expressions to narrow down the matching candidates.

The BoP [6] algorithm is a count-based method. Similar to the count-based conjunctive expression matching algorithms [3, 16, 52, 56], the basic idea of BoP is to compute the minimum number of predicates that need to be satisfied for each ABE. Only when the number of satisfied predicates is greater than the minimal number is the expression further evaluated. Another optimization of the BoP algorithm is that an ABE is encoded into a compressed format to reduce the memory cost. For example, the expression  $(P_1 \vee P_2 \vee P_3) \wedge P_4 \wedge (P_5 \vee P_6)$  is encoded in the format in Figure 2. In this figure, *Conj* and *Disj* possess the operators  $\wedge$  and  $\vee$ , respectively. The field *Child* possesses an operator argument count.

Conj	Child	Disj	Child	Leaf	ID	Leaf	ID	Leaf	ID	Leaf	ID	Disj	Child	Leaf	ID	Leaf	ID
1	3	2	3	4	1	4	2	4	3	4	4	2	2	4	5	4	6

Figure 2: BoP Encoding Example

A limitation of BoP is that a nonmatching ABE may be incorrectly identified as a candidate when the minimum number of predicates is satisfied. Consider the expression  $(P_1 \vee P_2 \vee P_3) \wedge P_4 \wedge (P_5 \vee P_6)$  as an example. The minimum number of satisfied predicates is 3. When  $P_1, P_2$ , and  $P_3$  are matching, this expression is identified as a candidate. However, this expression is not actually matching.

Another limitation of BoP is that different candidate expressions are evaluated separately. The shared subexpressions may be evaluated several times for a single event.

A binary decision diagram (BDD) is a data structure that is utilized to represent a Boolean function [8]. For example, BDDs are employed in verification and model checking. Campailla et al. [9] applied BDDs to represent Boolean expressions by assigning a Boolean variable to each unique predicate of an expression. Figure 3 shows the BDD for the expression  $(P_1 \vee P_2 \vee P_3) \wedge P_4 \wedge (P_5 \vee P_6)$ .

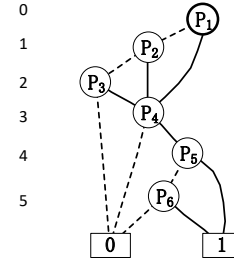


Figure 3: BDD Structure Example

A BDD tries to exploit the commonality among different expressions. As each distinct predicate is represented by a variable, when the number of variables increases, the BDD structure becomes complex. A BDD may increase exponentially with regard to the number of variables. The evaluation of an event with BDDs involves the traversal of the entire BDD in a top-to-bottom manner. Consequently, BDDs can exhibit high memory use and high matching time.

Many conjunctive Boolean expression matching algorithms have been proposed, such as PSTHash [30], OpIndex [56], BE-Tree [44, 45], Propagation [16], Gryphon [1], SIFT [1], TAMA [58], k-index[52], REIN [41] and GEM [17]. The idea of translation-based methods is to translate each ABE into a set of conjunctive Boolean expressions and then utilize conjunctive Boolean expression matching algorithms. Consider the expression  $(P_1 \vee P_2 \vee P_3) \wedge P_4 \wedge (P_5 \vee P_6)$  as an example; it is translated into six conjunctive Boolean expressions:  $P_1 \wedge P_4 \wedge P_5$ ,  $P_1 \wedge P_4 \wedge P_6$ ,  $P_2 \wedge P_4 \wedge P_5$ ,  $P_2 \wedge P_4 \wedge P_6$ ,  $P_3 \wedge P_4 \wedge P_5$ , and  $P_3 \wedge P_4 \wedge P_6$ . When any of these six expressions matches, the ABE matches. This method is simple; however, it introduces translation and maintenance costs. Also, given the potential exponential increase in expression size [51], both memory use and matching time can significantly increase. To the best of our knowledge, we are the first researchers to implement and evaluate this translation-based method. However, our experiments show that this method is not effective for ABE matching.

### 3 MATCHING MODEL

In this section, first, we present the definitions of predicates, ABEs, and events. Second, we specify the matching semantics.

#### 3.1 Expression Language

A *predicate* is a triple that consists of an attribute, an operator and a set of attribute values:  $P = \langle attr, op, vals \rangle$ . For example, in the predicate “ $age \in [16, 18]$ ”, the attribute is “*age*”, the operator is “ $\in$ ”, and the attribute values are “16” and “18”. If the attributes of two predicates are the same, we state that these two predicates belong to the same *dimension*. Predicates in our expression language support the standard relational operators ( $<, \leq, =, \neq, >, \geq$ ), set operators ( $\in$

,  $\neq$ ), and SQL's BETWEEN operator (*in*). As we will show in Sec. 3.2, given an event, a predicate can have an evaluation result of *true*, *false* or *undefined*.

An ABE is a Boolean function over a set of predicates. Formally, an ABE is expressed as  $f(P_1, \dots, P_m)$ . The logical operators connecting predicates in  $f$  are *and*, *or*, *not*, *xor*, and *xnor*, furnishing A-Tree with a highly expressive expression language.

Moreover, the same predicate can appear in an ABE more than once, which is generally not supported by a conjunctive Boolean expression. For example, in  $(P_1 \wedge P_2) \vee (\neg P_2 \wedge P_3)$ , the predicate  $P_2$  appears twice. This feature further improves the flexibility of our expression language.

An event contains a set of attribute-value pairs. Formally, an event  $E$  is defined as follows:

$$E = \{\langle attr_1, val_1 \rangle, \dots, \langle attr_n, val_n \rangle\}$$

For the same event, different attribute-value pairs cannot have the same attribute:  $attr_i \neq attr_j$ , if  $i \neq j$ .

### 3.2 Matching Semantics

We define the relation  $\Rightarrow$  to denote the evaluation result of a predicate  $P$  when its attribute receives the binding from event  $E$ :

$$E \vdash P \Rightarrow V$$

In this formula,  $E \vdash P$  means using the event  $E = \{\langle attr_1, val_1 \rangle, \dots, \langle attr_n, val_n \rangle\}$  as an environment to evaluate the predicate  $P = \langle attr, op, vals \rangle$ . The evaluation result  $V$  is *true* if  $\exists i : attr_i = attr \wedge \langle val_i, op, vals \rangle = true$ ;  $V$  is *false* if  $\exists i : attr_i = attr \wedge \langle val_i, op, vals \rangle = false$ ; and  $V$  is *undefined* if  $\forall i : attr_i \neq attr$ .

Similarly, when an ABE  $f$  receives a binding from event  $E$ , the evaluation result is:

$$E \vdash f = E \vdash f(P_1, \dots, P_m) \Rightarrow f(V_1, \dots, V_m)$$

We state that an ABE  $f$  matches event  $E$ , denoted as  $f \simeq E$ , if the evaluation result of  $f$  is *true* given the event  $E$ :

$$E \vdash f \Rightarrow f(V_1, \dots, V_m) = true$$

The matching problem can now be stated as follows: *Given event  $E$  and a set of arbitrary Boolean expressions, retrieve all the expressions matched by  $E$ . We refer to this problem as arbitrary Boolean expression matching. The objective is to minimize the matching time while maintaining a small index construction time and minimal memory use.*

## 4 A-TREE ORGANIZATION

In this section, first, we present the A-Tree data structure. Second, we present how an A-Tree index is dynamically constructed with different optimizations. Last, we analyze space use and insertion cost of A-Tree.

### 4.1 A-Tree Structure

Instead of representing an ABE as a binary decision diagram or separately storing the expressions, A-Tree represents each ABE as an n-ary tree drawing on overlap among shared subexpressions.

In A-Tree, we distinguish among three classes of nodes: a leaf node (*l-node*), which corresponds to a predicate; an inner node (*i-node*), which corresponds to a subexpression, and a root node (*r-node*), which corresponds to an ABE. To achieve a low memory cost and short matching time, A-Tree guarantees that the same

predicate, subexpression, and expression correspond to a unique *l-node*, *i-node*, and *r-node*, respectively.

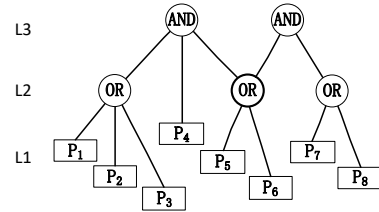


Figure 4: A-Tree Example

Assume that we have two ABEs. The first expression is  $(P_1 \vee P_2 \vee P_3) \wedge P_4 \wedge (P_5 \vee P_6)$ , and the second expression is  $(P_5 \vee P_6) \wedge (P_7 \vee P_8)$ . Figure 4 shows the A-Tree index constructed from these two expressions, where the subexpression  $(P_5 \vee P_6)$  corresponds to an *i-node*, which is shared by the two n-ary trees of the two ABEs.

In A-Tree, each *l-node* and *i-node* can have any number of parent nodes, while each *i-node* and *r-node* can have any number of child nodes. An *i-node* and *r-node* store a logical operator, including *and*, *or*, *not*, *xor* and *xnor*. In the event matching process, each of these nodes needs its child nodes' evaluation results to compute its evaluation result based on the logical operator. The evaluation results flow in a bottom-to-top approach.

To aid this matching process, each node of the A-Tree index keeps an integer referred to as *level* to track this node's distance from the farthest *l-node* among all nodes within its subtrees. The *level* of an *l-node* is set to 1. An edge between two nodes increases this value by 1. For an *i-node* or a *r-node*  $N$ , the *level* value is set according to the following formula:

$$level(N) = 1 + \max\{level(C_i) | C_i \text{ is a child of } N\}$$

### 4.2 Index Construction

The index construction of A-Tree which controls the degree of overlap achieved among different expressions, is important. We indicate that an A-Tree index instance is "good" if it needs only a small number of nodes and edges to index a fixed set of expressions. Furthermore, to match an event, only a small number of nodes and edges needs to be accessed. A good index structure has a key role in achieving low memory consumption and a low matching time.

To efficiently construct a good A-Tree index, we propose solutions in the sequel to achieve the following goals: First, every shared subexpression is guaranteed to be uniquely represented by a single node of A-Tree. Second, the organization of an incoming ABE is dynamically changed based on the current index structure to reuse a larger number of existing subexpressions. Third, the current A-Tree index can also be dynamically adjusted based on the newly incoming expressions. Fourth, expired expressions can be efficiently removed, and the A-Tree index will be adjusted accordingly.

Exploiting common subexpressions is an important design idea for A-Tree. Similar problems have been investigated in the compiler area [2, 15, 29]. However, query compilers focus mainly on optimizing the execution of a single query by reusing the intermediate results of an earlier query with common subexpressions. The method cannot be easily employed for our problem, which is to efficiently index tens of millions of ABEs.

**4.2.1 Node Uniqueness.** To ensure that every shared subexpression is uniquely represented by a single node in A-Tree, we propose a simple but efficient method: a unique ID is generated for each unique predicate, subexpression and expression, and an *expression-to-node hash table* ( $H_{en}$ ) is maintained. The key in this map is the generated ID; the value is the address of a node.

For a predicate, we simply generate its ID by running a hashing algorithm on this predicate's literal with a hashing collision handling mechanism. However, this method does not work for expressions because the same expression may have a large number of different literals. For example, *Expr1*:  $(P_1 \vee P_2 \vee P_3) \wedge P_4 \wedge (P_5 \vee P_6)$  and *Expr2*:  $P_4 \wedge (P_3 \vee P_2 \vee P_1) \wedge (P_6 \vee P_5)$  are actually the same.

To solve this problem, we generate an ID for an ABE based on the IDs of its predicates. The idea is to translate the logical operators *and*, *or*, *not*, *xor* and *xnor* into the arithmetic operators *add* and *multiply* and the bitwise operators *not*, *xor*, and *xnor*, respectively. For example, the ID of both *Expr1* and *Expr2* will be  $(Id_1 * Id_2 * Id_3) + Id_4 + (Id_5 * Id_6)$ , where  $Id_x$  represents the ID of the predicate  $P_x$ . Thus, expression equivalence determination in A-Tree accounts for different order of subexpressions.

To process an incoming ABE, a necessary step is to identify its subexpressions, i.e., child expressions. We determine the child expressions of an expression based on its literals. Consider the ABE  $(P_1 \vee P_2 \vee P_3) \wedge P_4 \wedge (P_5 \vee P_6)$  as an example; we state that it has three child expressions:  $P_1 \vee P_2 \vee P_3$ ,  $P_4$  and  $P_5 \vee P_6$ .

Alg. 1 shows how the expression-to-node hash table,  $H_{en}$ , is employed during the process of ABE insertion. If the expression already has a corresponding node in the A-Tree index, this node will be directly returned. Otherwise, nodes that correspond to its child expressions are obtained, and then a new node is created in the A-Tree index. In Alg. 1, predicates, child expressions and the incoming ABE are represented by the symbol *expr*.

---

**Algorithm 1** Insert(*expr*,  $H_{en}$ , *atree*)

---

```

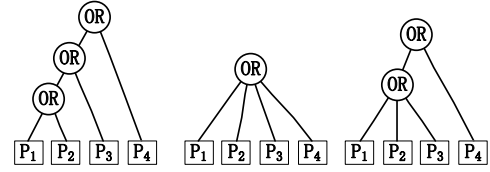
1: ID ← generateID(expr)
2: if  $H_{en}[id] \neq \text{null}$  then
3:   return  $H_{en}[id]$ 
4: else
5:   for childExpr ∈ expr.childExprs do
6:     childNode ← Insert(childExpr,  $H_{en}$ , atree)
7:     childNodes.add(childNode)
8:    $H_{en}[id] \leftarrow \text{createNewNode}(\text{expr}, \text{childNodes}, \text{atree})$ 
9:   return  $H_{en}[id]$ 

```

---

**4.2.2 Expression Reorganization.** For the A-Tree in Figure 4, assume that a new expression  $P_1 \vee P_2 \vee P_3 \vee P_4$  is received. Based on Alg. 1, a new node is created with four child nodes that correspond to the predicates  $P_1$ ,  $P_2$ ,  $P_3$  and  $P_4$ . However, there already is an *i*-node that corresponds to  $P_1 \vee P_2 \vee P_3$ . If this node is reused, the number of newly introduced nodes and edges will be reduced.

Based on the associative law of the logical operator *or*, for example, this incoming expression can be organized differently:  $((P_1 \vee P_2) \vee P_3) \vee P_4$ ,  $P_1 \vee P_2 \vee P_3 \vee P_4$  and  $(P_1 \vee P_2 \vee P_3) \vee P_4$ . Figure 5 exemplifies these structures. Although the second option has the fewest nodes and edges, the third option is more suitable for the A-Tree indexed of Figure 4 because the node that corresponds to



**Figure 5: Different Expression Structures**

the subexpression  $P_1 \vee P_2 \vee P_3$  can be reused. Reusing and sharing existing nodes in A-Tree helps to further reduce the memory consumption and improve the matching performance. Given the current index, by reorganizing an ABE, we can determine a representation that reuses more of the existing nodes. Expression ID generation already accounts for this aspect (cf. Sec. 4.2.1).

A further optimization is due to the commutativity of certain operators. For example, if several subexpressions are connected by the logical operator *and* or *or*, their order does not change the evaluation result of the expression. For example,  $P_1 \vee P_2 \vee P_3$  and  $P_2 \vee P_3 \vee P_1$  match the same events. Also, the same subexpression can be reused more than once. For example, if the expression  $P_1 \wedge P_2 \wedge P_3 \wedge P_4$  is inserted into an A-Tree index with two existing subexpressions  $P_1 \wedge P_2 \wedge P_3$  and  $P_3 \wedge P_4$ , then the inserted expression can be reorganized as  $(P_1 \wedge P_2 \wedge P_3) \wedge (P_3 \wedge P_4)$  such that these two existing subexpressions can be reused.

---

**Algorithm 2** Reorganize(*expr*, *atree*)

---

```

1:  $U \leftarrow \text{expr.childExprs}$ 
2:  $C \leftarrow \emptyset$ 
3: while  $U \neq \emptyset$  do
4:   select an  $S \in \text{atree}$  that maximizes  $|S \cap U|$ 
5:   if  $S = \emptyset$  then
6:     break
7:    $U \leftarrow U - S$ 
8:    $C \leftarrow C \cup \{S\}$ 
9:  $\text{expr.childExprs} \leftarrow C \cup U$ 

```

---

More generally speaking, by considering an expression as the set of its child expressions, this expression reorganization problem can be translated into a set cover problem: given a set of child expressions (referred to as the universe) and a collection  $S$  of  $m$  sets of expressions existing in A-Tree, we attempt to identify the smallest subcollection of  $S$  whose union equals the universe. The set cover problem is *NP-hard*. We use the greedy algorithm shown in Alg. 2 to solve this problem with an approximation. The strategy is to choose the existing expression in A-Tree that contains the most uncovered child expressions. Note that Line 5 to Line 6 are used to handle the case where not all child expressions are covered in the current A-Tree index.

**4.2.3 Index Self-adjustment.** An advantage of A-Tree is that it does not need to be built offline in advance, which means that the newly incoming ABEs are automatically handled and the index structure is dynamically adapted.

In this section, we proposed a method to dynamically change the organization of an incoming ABE based on the current A-Tree index structure. However, this approach is not sufficient in some situations. For example, assume that the current A-Tree index is expressed as shown in Figure 4. When a new expression  $(P_1 \vee P_2 \vee$

$P_3) \wedge P_4$  is received, the existing nodes corresponding to  $P_1 \vee P_2 \vee P_3$  and  $P_4$  can be reused. Based on Alg. 1, a new node corresponding to the incoming expression  $(P_1 \vee P_2 \vee P_3) \wedge P_4$  is created. In this situation, it would be ideal if the new A-Tree index could be further optimized because the newly created node can be reused as the child of the existing node corresponding to the expression  $(P_1 \vee P_2 \vee P_3) \wedge P_4 \wedge (P_5 \vee P_6)$ . Therefore, we propose another optimization to dynamically adjust the A-Tree index structure based on the newly incoming ABE. The motivation for A-Tree index self-adjustment is to ensure that the A-Tree index remains optimized regardless of the arrival order of the expressions.

Alg. 3 shows the self-adjustment of A-Tree when a new node is created. Via the new node's child nodes, the candidate nodes whose corresponding expression covers the new node's corresponding expression can be located. The index is then updated to reuse the new node as a child node of these candidate nodes.

---

**Algorithm 3** SelfAdjust(*newNode*)

---

```

1: for childNode  $\in$  newNode.childNodes do
2:   for parentNode  $\in$  childNode.parentNodes do
3:     if newNode.expr  $\subset$  parentNode.expr then
4:       update(childNode, parentNode, newNode)

```

---

The complete ABE insertion process is shown in Alg. 4. In the node structure of A-Tree, an integer field named *useCount* is kept, which indicates the total number of predicates, subexpressions and expressions that use this node. If an existing expression is inserted, then the *useCount* of the corresponding node is increased by 1. Otherwise, this expression is reorganized, a new node is created, and the A-Tree index is self-adjusted.

---

**Algorithm 4** Insert(*expr*, *H<sub>en</sub>*, *atree*)

---

```

1: ID  $\leftarrow$  generateID(expr)
2: if Hen[id]  $\neq$  null then
3:   Hen[id].useCount += 1
4:   return Hen[id]
5: else
6:   Reorganize(expr, atree)
7:   for childExpr  $\in$  expr.childExprs do
8:     childNode  $\leftarrow$  Insert(childExpr, Hen, atree)
9:     childNodes.add(childNode)
10:  node  $\leftarrow$  createNewNode(expr, childNodes, atree)
11:  node.useCount  $\leftarrow$  1
12:  SelfAdjust(node)
13:  Hen[id]  $\leftarrow$  node
14:  return node

```

---

**4.2.4 Expression Deletion.** Deleting an expression is a straightforward and fast operation. When an expression is deleted, the *useCount* of its corresponding node *N* is decremented by 1. If the *useCount* becomes 0, *N* can be safely removed because it is already not required for any expression. However, it is still possible that *N* is used as the child node of another node *P*. In this case, *P* will be changed to consume the child nodes of *N* as its child nodes. Subsequently, *N* continues to be removed from the A-Tree index. The corresponding record in the *expression-to-node* hash table is

also removed. In this way, the A-Tree index achieves dynamic self-adjustment during expression deletion. As shown in Alg. 5, when an expression is processed, all its child expressions are recursively processed in the same way.

---

**Algorithm 5** Delete(*expr*, *H<sub>en</sub>*, *atree*)

---

```

1: ID  $\leftarrow$  generateID(expr)
2: node  $\leftarrow$  Hen[id]
3: node.useCount -= 1
4: if node.useCount = 0 then
5:   Remove(node, atree)
6:   Hen[id]  $\leftarrow$  null
7:   for childExpr  $\in$  expr.childExprs do
8:     Delete(childExpr, Hen, atree)

```

---

### 4.3 Complexity Analysis

**Space Consumption:** For a single ABE with  $N_p$  predicates, both the number of required nodes and the number of links is  $O(N_p)$ . For an A-Tree index constructed from  $N_{exp}$  expressions, in the worst case, no subexpressions are shared. In this case, the space complexity of A-Tree is  $O(N_{exp} * N_p)$ . However, in practice, as we illustrate in Sec. 6.2, subexpressions are generally shared many times. For example, if *gender = female* is one of the attributes of interest, the predicate “*gender = female*” may be shared thousands of times. Thus, the average space complexity of A-Tree is  $O((N_{exp}/N_{se}) * (N_p/N_{sp}))$ , where  $N_{se}$  and  $N_{sp}$  are the average number of times an expression and a predicate are shared, respectively. This section proposed several index construction optimizations with the goal of increasing the value of  $N_{se}$  and  $N_{sp}$ .

**Expression Insertion:** As shown in Alg. 4, if an incoming expression or predicate already exists in the A-Tree index, the cost of insertion is  $O(1)$ . Otherwise, the cost of insertion comprises three parts: reorganization cost, new node creation cost, and index self-adjustment cost; the time complexities of these three operations are  $O(N_p^2)$ ,  $O(1)$  and  $O(N_p)$ , respectively. Thus, the average expression insertion time complexity is  $O((1/N_{se}) * (N_p/N_{sp})^2)$ .

**Expression Deletion:** As shown in Alg. 5, to delete an expression or predicate, we simply decrease the count of the corresponding node by one. Only when the count becomes zero does this node need to be removed and the delete operation is executed on its child nodes. Thus, the average expression deletion time complexity is  $O((1/N_{se}) * (N_p/N_{sp}))$ .

## 5 EVENT MATCHING

An ABE may contain not only *and* but also *or*, *not*, *xor*, and even *xnor* logical operators, resulting in a complex hierarchical expression organization. Compared to conjunctive Boolean expression matching, this flexibility increases the expressiveness for applications that employ expression matching. However, it also makes ABE matching more complex and challenging. For example, for conjunctive Boolean expression matching, only the satisfied predicates must be identified because a conjunctive expression is matching only when all its predicates are satisfied. However, for ABE matching, both the satisfied predicates and unsatisfied predicates need to be identified since an ABE may be matching based on unsatisfied predicates. For example, the ABE  $\neg(P_5 \vee P_6)$  is matching when both of the predicates  $P_5$  and  $P_6$  are unsatisfied.

A-Tree-based event matching proceeds in two phases: predicate matching and expression matching. Given an event, the predicate matching phase determines all satisfied and unsatisfied predicates. As we discussed in Sec. 3, an unsatisfied predicate means that the evaluation result of the predicate is *false*. A predicate's evaluation result is *undefined* if the event does not contain the predicate's attribute. The expression matching phase locates all satisfied ABEs. Several predicate matching algorithms have been proposed, such as Segment-Tree [5], table-based schemes [3], Interval-Skip [24], Interval-Tree [37], and PS-Tree [30]. A-Tree is compatible with all of these algorithms.

To run event matching, in addition to the A-Tree index, a set of queues are needed. In A-Tree, to verify whether an *i*-node's corresponding expression is satisfied, the evaluation results of its child nodes must be known in advance. The evaluation sequence in A-Tree is from a low *level* node to a high *level* node. Thus, we keep a queue for each *level* to ensure that the nodes are traversed in the correct order.

---

**Algorithm 6** Match(*preds*, *H<sub>en</sub>*)

---

```

1: for pred ∈ preds do
2:   ID ← generateID(pred)
3:   l-node ← Hen[id]
4:   l-node.result ← pred.result
5:   Q1.add(l-node)
6: for level = 1 → M do
7:   while Qlevel is not empty do
8:     node ← Qlevel.dequeue()
9:     result ← node.evaluate()
10:    node.clean()
11:    if result = undefined then
12:      continue
13:    for all parent ∈ node.parents do
14:      if parent.operands.empty() then
15:        plevel ← parent.level
16:        Qplevel.add(parent)
17:        parent.operands.add(result)
18:      if result = true then
19:        matchingExprs.add(node.exprs)
20: return matchingExprs

```

---

### 5.1 Matching Algorithm

As shown in Alg. 6, for each incoming event, after all satisfied and unsatisfied predicates are identified, their corresponding *l*-nodes in A-Tree are located by the *expression-to-node hash table* (*H<sub>en</sub>*). These *l*-nodes are queued into *Q*<sub>1</sub>, which corresponds to the first level. After this initial step is finished, the matching algorithm processes the lowest level unfinished queue. After finishing a queue *Q<sub>i</sub>*, it starts to process the next higher level queue *Q<sub>i+1</sub>* until the highest level queue *Q<sub>M</sub>* is processed. For any queue *Q<sub>i</sub>*, visiting a node *N* in *Q<sub>i</sub>* involves the following two basic tasks:

**Evaluation of Result:** Evaluation of node *N* is performed according to the node type, operator and operands. If *N* is an *l*-node, the evaluation result of *N* is copied from the predicate matching phase. For an *i*-node and *r*-node, the evaluation result is determined by its operator and the operands from its child nodes. When none

of the operands is *undefined*, the result is evaluated normally. Otherwise, the result is determined by the logic given in Table 2. Similar methods to reduce a 3-valued logic to a 2-valued one have been used in the literature (e.g., see [12]).

The operands must be available immediately at the point of visiting *N*. This availability is ensured by the level-ordered bottom-to-top traversal of the A-Tree. An event matches the ABE that corresponds to *N* if the evaluation result of *N* is *true*.

**Propagation of Result:** If the evaluation result of *N* is *undefined*, this result does not need to be propagated to the parent nodes of *N* since the default value of any operand is *undefined*. Otherwise, the result is propagated to its parent nodes. For parent node *P*, we check whether its operands are empty. If the answer is yes, then *P* is inserted into the queue *Q<sub>i</sub>*, where *i* is the level of *P*. The evaluation result of *N* is added to the operands of *P*.

**Table 2: Evaluation on the Operand Undefined**

Operator	Operand1	Operand2	Result
<i>and</i>	<i>undefined</i>	<i>true</i>	<i>undefined</i>
<i>and</i>	<i>undefined</i>	<i>false</i>	<i>false</i>
<i>or</i>	<i>undefined</i>	<i>true</i>	<i>true</i>
<i>or</i>	<i>undefined</i>	<i>false</i>	<i>undefined</i>
<i>not, xor, xnor</i>	<i>undefined</i>	<i>any</i>	<i>undefined</i>

### 5.2 Optimizations for Event Matching

The main cost of A-Tree-based event matching is propagation of the matching result to a node's parent nodes and the subsequent evaluation of the parent nodes. Two factors increase the event matching time. First, when the evaluation result of a node is *false*, it still must be propagated to its parents. This process is expensive considering the potentially large number of unsatisfied predicates and subexpressions. Second, for node *N* with the logical operator *and*, every child node's matching result is propagated to *N*. This step is not necessary because if any child node is not satisfied, *N* will not be satisfied. In this section, we propose optimizations to overcome these two limitations.

**5.2.1 Zero Suppression Filter.** Because of the existence of the logical operators *not*, *xor* and *xnor* in ABEs, we must distinguish between the *false* and *undefined* evaluation results. The *false* evaluation result needs to be propagated, which can be very expensive. To remove the cost to propagate *false* results, we use the *zero suppression filter* optimization. Its basic idea is to remove the logical operators *not*, *xor* and *xnor* from an incoming ABE by applying the following laws: (1)  $\neg(E_1 \wedge E_2) = \neg E_1 \vee \neg E_2$ , (2)  $\neg(E_1 \vee E_2) = \neg E_1 \wedge \neg E_2$ , (3)  $E_1 \oplus E_2 = (E_1 \wedge \neg E_2) \vee (\neg E_1 \wedge E_2)$ , and (4)  $E_1 \otimes E_2 = (E_1 \wedge E_2) \vee (\neg E_1 \wedge \neg E_2)$

In this formula,  $\oplus$  and  $\otimes$  are the symbols of *xor* and *xnor*, respectively. In negation removal, by applying De Morgan's laws, all negations are pushed downward level-by-level onto the predicates. Negations are then integrated into predicates (i.e., applied to the predicates) by using the inverse of the predicates' relational operators: replacing *greater than* with *less than or equal*, *equality* with *inequality*, etc. Consider the expression  $\neg((age > 60) \vee (weight > 100))$  as an example; it changes to  $(age \leq 60) \wedge (weight \leq 100)$ .

With this optimization, when the evaluation result of any node is *false*, it is no longer propagated to any of its parents. Conversely,



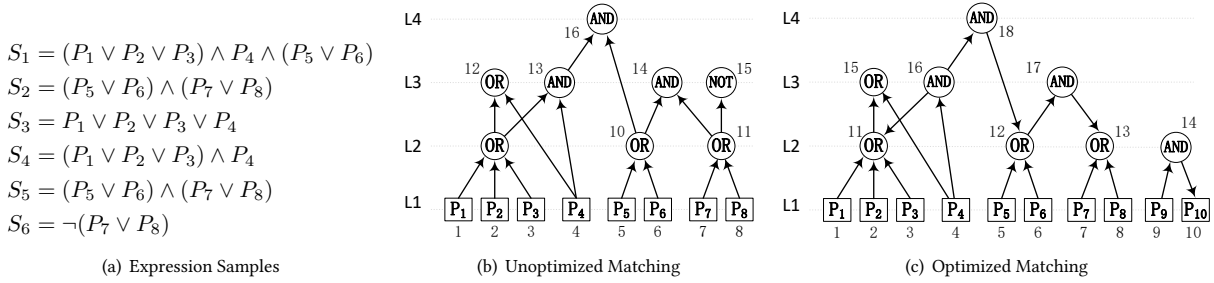


Figure 6: Event Matching Example

when computing the result of any node, if one of its operands is *undefined*, it is assumed to be *false*, and the result is computed accordingly. This optimization is promising because a *false* result is often obtained for a substantial number of leaf and inner nodes in the A-Tree index while matching an event. Our experiments show that the event matching time can be reduced by a maximum of 85% with this optimization.

**5.2.2 Propagation On Demand.** In A-Tree, for node  $N$  with the logical operator *and*, only when all its child nodes are satisfied, is  $N$  satisfied. Based on this property, we propose an optimization referred to as *propagation on demand*. As the name suggests, we propagate the matching results of  $N$ 's child nodes to  $N$  only when needed. The basic idea of *propagation on demand* is expressed as follows: If  $N$  is associated with the logical operator *and*, we randomly select one child node of  $N$  as the *access* child. In the A-Tree index, only this *access* child has a parent link to  $N$ . Furthermore,  $N$  has links to its other child nodes.

During event matching, only when the *access* child of  $N$  is satisfied, is the matching result propagated to  $N$  and is  $N$  evaluated. During the evaluation process of  $N$ , we propagate the matching results of  $N$ 's other child nodes to  $N$ . However, if the *access* child of  $N$  is unsatisfied, no matching results will be propagated to  $N$ , and  $N$  will not be evaluated.

To support this optimization, we need to store the matching results of  $N$ 's child nodes because the matching results may be needed for the evaluation of  $N$ . The question is when to clean up the stored matching results. Without cleaning, the matching of the next event will be affected. To efficiently solve this problem, we associate every matching result of a node to the signature of the current event. If the event signature of a matching result is not the same as the current processing event, this matching result will be simply discarded. This is a lazy status cleaning mechanism.

By this *propagation on demand* optimization, we can filter a large number of unnecessary propagations: only when the *access* child is satisfied are the matching results propagated. This optimization is very effective for workloads with many *and* expressions. For example, for some conjunctive expression workloads, our experiments show that this optimization reduces the matching time by a maximum of 92%.

### 5.3 Optimized Event Matching Example

In this section, we describe the execution steps of event matching using an example. In this example, there are six ABEs, as shown in Figure 6(a). We have two versions of the A-Tree index built on these expressions: the unoptimized version shown in Figure 6(b) and the optimized version shown in Figure 6(c). In both Figure 6(b) and

Figure 6(c), the leaf nodes 1, 2, 3, 4, 5, 6, 7, and 8 correspond to the predicates  $P_1, P_2, P_3, P_4, P_5, P_6, P_7$  and  $P_8$ , respectively. Because the *not* operator is removed for the *zero suppression filter* optimization, there are two more leaf nodes in Figure 6(c). These two leaf nodes correspond to the predicates  $P_9$  and  $P_{10}$ .  $P_9$  is equal to  $\neg P_7$ , and  $P_{10}$  is equal to  $\neg P_8$ . Another difference is that there are not only *child-to-parent* links but also some *parent-to-child* links in Figure 6(c).

To save space, here, we focus mainly on illustrating optimized event matching. Given an event  $E$ , we assume that only predicates  $P_1, P_9$  and  $P_{10}$  are identified as matching in the predicate matching phase. Thus, only nodes 1, 9, and 10 are queued into  $Q_1$ . These queues are then processed from Level 1 to Level 4 as follows:

**Level 1:** Nodes residing in  $Q_1$  are separately processed. First, Node 1 is dequeued from  $Q_1$ , and its matching result *true* is propagated only to Node 11. Node 11 is queued into  $Q_2$ . Second, Node 9 is dequeued from  $Q_1$ , and its result is propagated to Node 14. Node 14 is also queued into  $Q_2$ . Last, Node 10 is dequeued from  $Q_1$ , and its matching result *true* is stored in the node with the current event's signature.

**Level 2:**  $Q_1$  is empty, and the nodes in  $Q_2$  start to be processed. First, Node 11 is dequeued from  $Q_2$ . Because its operator is *or* and one of its operands is *true*, its evaluation result is *true*. This evaluation result is propagated only to Node 15, and Node 15 is queued into  $Q_3$ . Second, Node 14 is dequeued from  $Q_2$ . Because its operator is *and*, the stored matching result at Node 10 is propagated on demand to Node 14 with the associated event signature. Node 14 is identified as a satisfied node. Thus, the associated expression  $S_6$  is identified as a matching expression.

**Level 3:** First, Node 15 is dequeued from  $Q_3$ . Its operator is *or*, and one operand is *true*; thus, its evaluation result is *true*. The expression  $S_3$  associated with this node is identified as a matching expression.  $Q_4$  is empty. Thus, proceeding to the next level is not necessary. After completing these steps, we retrieved the matching expressions  $S_3$  and  $S_6$ .

### 5.4 Matching Complexity Analysis

A-Tree-based event matching identifies matching predicates, subexpressions, and expressions by traversing the index from bottom to top in a layer-by-layer manner. For an event with  $|E|$  attribute-value pairs, the number of matching predicates is at most  $|E| * N_v$ , where  $N_v$  is the average number of unique predicates that match a given attribute-value pair. For most applications, the number of matching unique subexpressions and expressions is usually smaller than the number of matching unique predicates for an event. Therefore, the overall event matching time complexity is roughly  $|E| * N_v * N_l$ , where  $N_l$  is the number of levels of the A-Tree index.



## 6 EXPERIMENTS

In this section, first, we discuss the synthesized and real-world workloads used in our experiments. For the real-world workload, a detailed analysis is given to assess the importance of considering subexpression sharing in the algorithm. Second, we use microexperiments to analyze the effects of different A-Tree design elements. Last, we compare the performance of A-Tree with existing solutions using comprehensive experiment configurations.

We compare A-Tree-based ABE matching against the following algorithms: Dewey ID [19], Interval ID [19], BoP [6], BDD [9], Translation, and Scan. We implemented all algorithms in C. We compiled our code with gcc 7.4 using optimization level O3 on an Ubuntu 18.04 system. All experiments were run on a machine with an Intel 2.20 GHz CPU and 512 GB of memory.

In the experiments, we consider a variety of controlled experimental conditions: workload size, operator distribution, arbitrary expression tree depth, number of child nodes, expression distribution, number of dimensions, dimension cardinality, dimension distribution, and event size. In this paper, a dimension refers to the value domain underlying an attribute. We define dimension cardinality as the number of possible values for the dimension. For example, if a dimension is *country*, its cardinality is the number of countries in the world (in the associated domain). Dimension distribution refers to the distribution of the predicates' attributes.

### 6.1 Synthesized Workloads

To synthesize ABE workloads, we designed a workload generator that is referred to as ABE-Gen. In ABE-Gen, the generation of an ABE, i.e., an  $n$ -ary tree, starts from the root node and recursively moves to the child nodes. For each node, we first decide the logical operator, which is selected from *and*, *or*, *not*, *xor* and *xnor*. If the logical operator is selected as *and* or *or*, we then decide on the number of child nodes. In addition to the logical operator and child node number, we use another parameter that is referred to as *tree depth* to control the expression generation. When a node's depth is equal to the maximum tree depth, this node is identified as a leaf node. For each leaf node, we generate a predicate.

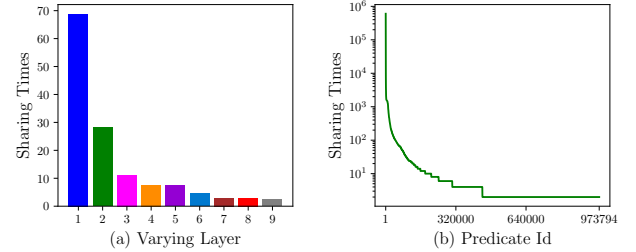
To generate a variety of ABEs, we use a wide range of parameters and settings, as shown in Table 3, with the default values highlighted in bold. To evaluate scalability, we vary the number of expressions from 100 K to 30 M. The default logical operator distribution is 40% *and*, 40% *or*, 10% *not*, 5% *xor* and 5% *xnor*. The expression tree depth varies from 1 to 6. The child node number varies from 2 to 12. In real-world workloads, the same subexpression often appears many times. In addition, as we show next, the sharing ratio of subexpressions often follows the Zipf distribution in practice. Thus, we also support the generation of subexpressions in our synthesized workloads following the Zipf distribution  $P(r) = \frac{C}{r^\alpha}$ ,  $r \neq 0$ . When  $\alpha$  is 0, the distribution is uniform and all generated subexpressions are different. Otherwise, the distribution is Zipfian and subexpressions are shared. The number of dimensions varies from 10 to 30 K. The default number of dimensions is set to 1000. We vary the dimension cardinality from 1 to 3 K. The predicates' attributes are also drawn following the Zipf distribution. We vary the average event size from 5 to 30. For the default synthesized workload, on average, a predicate is shared approximately 18.35 times, while a subexpression is shared approximately 4.33 times.

**Table 3: ABE-Gen Parameters on Synthetic Datasets**

Number of Expressions	100K, 300K, <b>1M</b> , 3M, 10M, 30M
Operator Distribution	40%, 40%, 10%, 5% 5%
Tree Depth	1, 2, <b>3</b> , 4, 5, 6
Child Node Number	2, <b>4</b> , 6, 8, 10, 12
$\alpha$ in Zipf Distribution	0, 0.2, 0.4, <b>0.6</b> , 0.8, 1
Number of Dimensions	30, 100, 300, <b>1K</b> , 3K, 10K
Dimension Cardinality	3, 10, 30, <b>100</b> , 300, 1K
Event Size	5, 10, 15, <b>20</b> , 25, 30

### 6.2 Real-World Workload

In addition to these synthetic datasets, we also employ a real-world workload by using a Display Ads dataset from a major online seller. When a user surfs on the online shop, product advertisements are shown to the user. An advertisement specifies conditions to promote products to users. The conditions include channel (e.g., mobile, PC, or tablet), region (e.g., US, DE, or CN), ads position, etc. By mapping conditions into predicates, advertisements are naturally modeled as ABEs. 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, login region, and user's profile. By mapping the profile and attributes into attribute-value pairs, each session is modeled as an event. For example, if a user is male, then the resulting event contains an attribute-value pair  $\langle \text{gender}, \text{male} \rangle$ .



**Figure 7: Subexpression Sharing Analysis**

Our workload contains 1,392,196 expressions. The number of predicates in an expression ranges from 1 to 56. The depth of an expression ranges from 1 to 9. The number of dimensions is 122. On average, each event contains approximately 20 attribute-value pairs.

In addition to these basic metrics, we also carefully analyzed subexpression sharing in this workload (i.e., number of subexpressions over unique subexpressions at a given level.) Figure 7(a) shows the average number of shared subexpressions at different expression levels: we say that predicates are at Level 1, while the subexpressions that are directly constructed from predicates are at Level 2, etc. As can be seen, on average, a Level 1 subexpression is shared a maximum of 68.76 times. Subexpressions at higher levels have lower average sharing ratios. At Level 9, the average sharing ratio is approximately 2.88. Figure 7(b) shows the sharing ratio distribution of the 973,794 distinct Level 1 subexpressions, a.k.a., predicates. As can be seen, the sharing ratios range from 1 to hundreds of thousands. Overall, the sharing ratios of these predicates roughly follow a Zipf distribution, which is the same as the subexpressions at other levels. These analysis results indicate the importance of considering and utilizing common subexpression sharing.

### 6.3 Microexperiments

In this section, we perform three groups of microexperiments to provide an independent evaluation of improvements generated by different A-Tree design elements. Specifically, we answer three questions: (1) How large are the benefits of a multitree representation of shared subexpressions combined with a bottom-to-top traversal for matching? (2) What are the benefits and costs of using a unique ID to identify common subexpressions and support index dynamicity? (3) What is the impact of different matching optimizations?

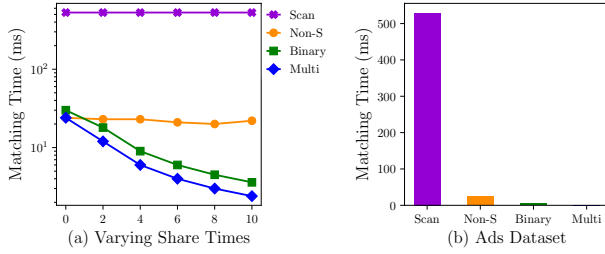


Figure 8: Sharing and Multitree

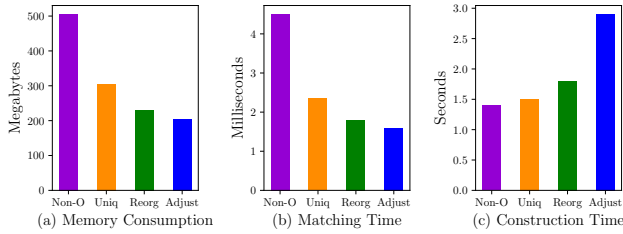


Figure 9: Uniqueness and Dynamicity

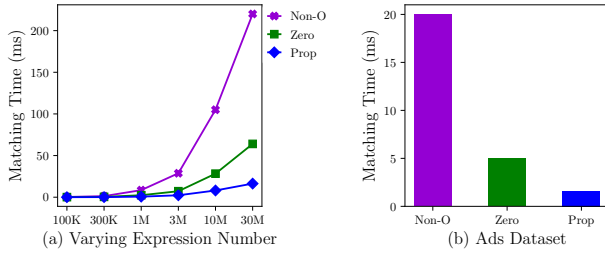


Figure 10: Matching Optimizations

**Sharing and Multitree:** Figure 8 compares three candidate design options against Scan: *a multitree index not supporting sharing*, *a binary tree index supporting sharing*, and *a multitree index supporting sharing*. Figure 8(a) shows the matching time when the subexpression sharing ratio varies in the synthesized workload, while Figure 8(b) shows the matching time of these design options on the Ads workload. As can be seen, *multitree index supporting sharing* performs the best. Moreover, *multitree index not supporting sharing* still performs much better than Scan. The reason is that we use access predicates and subexpressions to identify many fewer potentially matching candidate expressions and match in a bottom-to-top manner.

**Uniqueness and Dynamicity:** Figure 9 compares the four design options for A-Tree index construction: (1) *not adopting any*

*optimization*, (2) *utilizing only the unique ID mechanism*, (3) *further supporting expression reorganization*, and (4) *further supporting index self-adjustment*. Since the effects of these design options can be highly influenced by the workload distribution, we directly employed the real-world Ads workload rather than synthesized workloads to derive our results. By adopting more optimizations, both the memory consumption and matching time are reduced. The only cost is the increased index construction time. In this group of experiments, by changing the order in which the expressions arrive, we observe that memory consumption varies for the first three design options, while the memory consumption is stable when the *index self-adjustment* optimization is used.

**Matching Optimizations:** Figure 10 shows matching time reductions when the *zero suppression filter* and *propagation on demand* optimizations are used. It can be seen that these two optimizations are very effective at reducing the matching times for both synthesized workloads and real-world workload. Note that similar mechanisms can also be applied to related work, such as to BoP and Scan. However, they do not present obvious matching time reductions because they run event matching in a top-to-bottom manner and identify many more candidate expressions. An important contribution of our work is to combine several optimizations to construct an overall more efficient solution.

### 6.4 Experiments on Synthesized Workloads

The Translation-based solution can be based on different conjunctive expression matching algorithms, such as PSTBloom, BE-Tree and OpIndex. Because experiments showed that PSTBloom performs better than other algorithms, in this paper, Translation is built on top of PSTBloom. The basic ideas behind Dewey ID and Interval ID are similar [19]. In addition, our experiments show that Interval ID always performs slightly better than Dewey ID, which is consistent with findings by Fontoura et al. [19]. Therefore, we only use Interval ID for a comparison in this section.

**6.4.1 Memory Consumption.** For ABE workloads, the memory consumption of different algorithms is affected by the workload size, expression tree depth, average number of child nodes, and expression distribution.

Figure 11(a) shows the memory usage as the number of expressions increases. Unsurprisingly, all algorithms require more memory when the number of expressions increases. However, the memory usage of A-Tree and BDD increases more slowly than that of Interval ID, BoP and Translation because the common subexpressions can be shared in A-Tree and BDD. In particular, A-Tree requires the least memory when there are more than 300 K expressions, because A-Tree adopts several other index construction optimizations. BoP also presents a good memory footprint, because BoP compresses the expressions and requires only a count for each expression in its index. Translation needs the most memory because a single arbitrary expression can be translated into a maximum of  $4^4 = 256$  conjunctive expressions in our workload. The green line, which is marked as *TR1*, represents the memory consumed by the underlying PSTBloom index, while the gray line, which is marked as *TR2*, represents the memory consumed for the translated conjunctive Boolean expressions. When there are 30 M expressions, compared with the next-best algorithm BoP, A-Tree reduces the memory cost by 71%, while Translation runs out of memory.

Figure 11(b) shows the memory usage as the average ABE tree depth increases. BDD and Translation run out of memory when the average tree depth increases to 4 and 5, respectively. For Translation, this situation occurs because a single arbitrary expression can be translated into a large number of conjunctive expressions. For BDD, this situation occurs because the complexity of the BDD index increases exponentially with the number of predicates in an expression. A-Tree needs the least amount of memory.

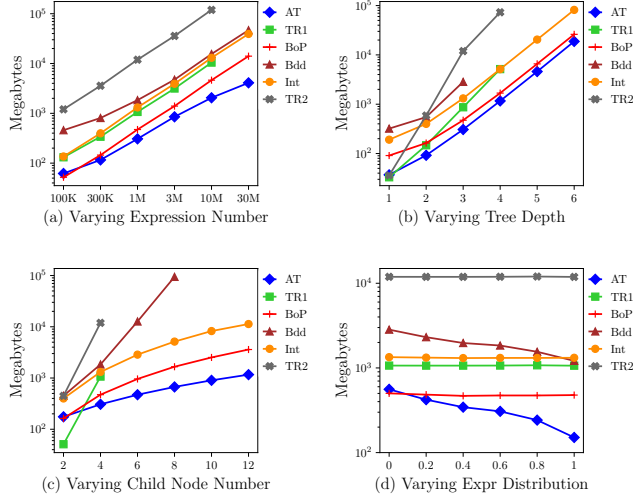


Figure 11: Memory Consumption

Figure 11(c) shows the memory usage as the average number of child nodes increase. Similar to the tree depth experiments, Translation runs out of memory when the average child node number increases to 6, while BDD runs out of memory when the child node number increases to 10. The ranking of the algorithms is A-Tree, BoP and Interval ID.

Figure 11(d) shows the memory usage as the expression distribution changes. The memory usage of A-Tree and BDD decreases as  $\alpha$  of the Zipf distribution increases. When  $\alpha$  increases to 0.2, A-Tree consumes the least amount of memory. When  $\alpha$  is 1, A-Tree reduces the memory usage by 68%.

**6.4.2 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.

**Workload Size:** We consider the matching time as we increase the number of expressions processed. As illustrated by Figure 12(a), all algorithms scale linearly with respect to the number of expressions. Among them, A-Tree always performs the best because any common subexpressions are evaluated at most once. Moreover, the unnecessary propagation of evaluation results is terminated by the event matching optimizations. When there are 30 M expressions, A-Tree reduces the matching time by 86% compared to the next-best algorithm BDD. The matching times of Interval ID and BoP are very similar. Translation ranks second before it runs out of memory. This finding reveals that Translation is efficient at event matching. However, the massive memory cost limits its scalability.

**Expression Tree Depth:** As shown in Figure 12(b), when the expression tree depth increases, the matching times of all algorithms increase. However, the increasing speeds are not equivalent.

BDD and Translation run out of memory when the tree depth increases to 4 and 5, respectively. Interval ID and BoP perform even worse than Scan when the tree depth increases to 6. Only A-Tree continues to present a short matching time when the tree depth increases. When the tree depth is 6, compared with the next-best algorithm Scan, A-Tree reduces the matching time by a maximum of 99%.

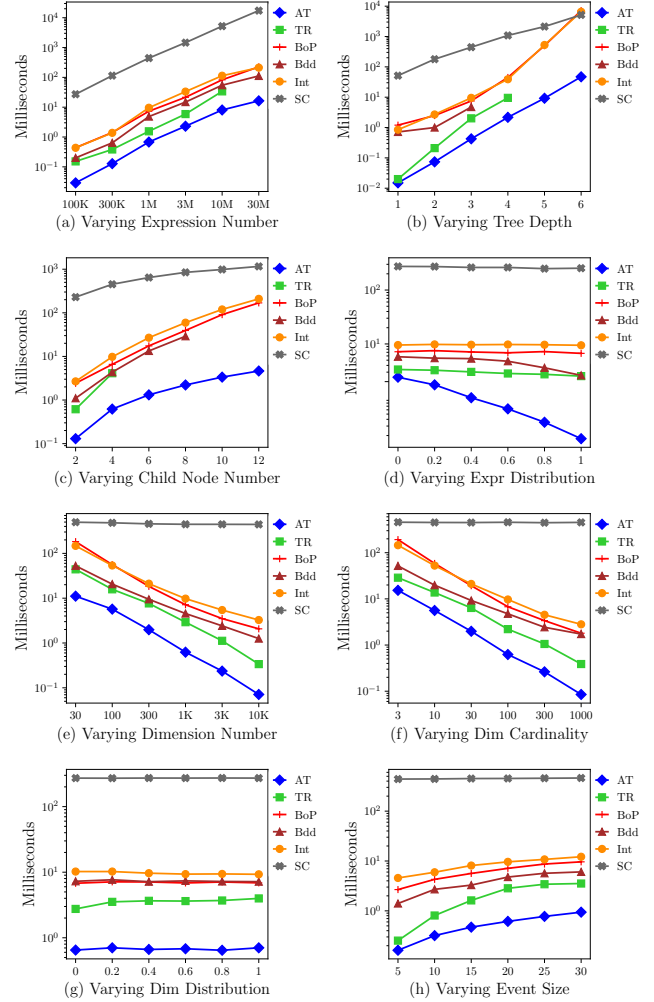


Figure 12: Event Matching Time

**Child Node Number:** The effect of the child node number is similar to the effect of the expression tree depth. As shown in Figure 12(c), only A-Tree scales well for expressions with a large number of child nodes. Interval ID and BoP do not scale well because their mechanisms to retrieve candidate matching expressions are not efficient for complex expressions.

**Expression Distribution:** Figure 12(d) shows that only the matching time of A-Tree decreases as the value of  $\alpha$  increases. The matching time of BDD also decreases but not as pronounced as that of A-Tree because the evaluation of events with BDD involves the traversal of the entire BDD index. BDD-based matching occurs from top-to-bottom, which is not as efficient as the bottom-to-top approach of A-Tree-based matching.

**Number of Dimensions:** As shown in Figure 12(e), when the number of expressions is fixed and the number of dimensions increases, the matching times of all algorithms decrease, except for Scan, because a higher number of dimensions results in a lower number of matching predicates and expressions. The ranking of these algorithms is A-Tree, Translation, BDD, BoP, Interval ID and Scan.

**Dimension Cardinality:** Figure 12(f) shows how the dimension cardinality affects the matching time. Similar to the number of dimensions, when the dimension cardinality increases, the matching times of A-Tree, Translation, BDD, BoP and Interval ID decrease, because a higher dimension cardinality also results in a lower number of matching predicates and expressions.

**Dimension Distribution:** Figure 12(g) shows the matching times of all algorithms when the attributes of predicates, a.k.a. dimensions, follow the Zipf distribution. It can be seen that the matching times of all algorithms are not obviously affected by the dimension distribution. This experimental result is consistent with the notion that none of these algorithms adopt dimension-related optimizations.

**Event Size:** When the average number of attribute-value pairs of an event increases, there are more matching predicates and expressions. As shown in Figure 12(h), all algorithms except for Scan present higher matching times as the event size increases. The ranking of these algorithms is A-Tree, Translation, BDD, BoP, Interval ID and Scan. The ranking order does not change as the event size increases.

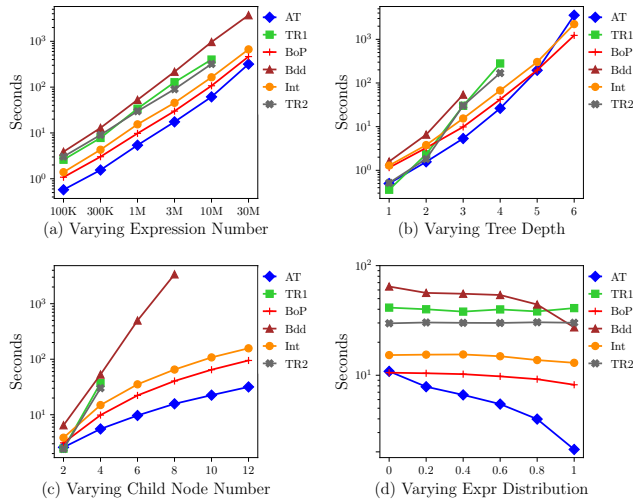


Figure 13: Index Construction Time

**6.4.3 Index Construction Time.** The index construction times of different algorithms are affected by workload size, tree depth, average number of child nodes and expression distribution. As shown in Figure 13(a), the index construction times of all algorithms increase with an increase in the expression number. A-Tree, BoP and Interval ID require lower index construction times than Translation and BDD. Among them, A-Tree performs the best. A similar phenomenon is observed in Figure 13(b) and Figure 13(c). These experimental results reveal that Translation and BDD are not suitable for workloads with a large number of complex arbitrary expressions. In Figure 13(d), when we increase  $\alpha$  of the expression distribution,

the index construction speed of A-Tree becomes faster than that of the other algorithms. When  $\alpha$  increases to 1, compared to the next-best algorithm BoP, A-Tree reduces the index construction time by 75%.

## 6.5 Experiments on the Ads dataset

We compare A-Tree with related approaches by using our workload generated based on a configuration derived from our real-world Ads dataset. As shown in Table 4, under this workload, A-Tree achieves the shortest matching time, followed by BDD and Translation. A-Tree also achieves the best index construction performance and memory footprint. The matching time of BoP is similar to that of Dewey ID and Interval ID. However, the index construction performance and memory usage of BoP is better than those of Dewey ID and Interval ID. Interval ID performs slightly better than Dewey ID. Compared with BDD, A-Tree reduces the matching time, index construction time and memory usage by 86%, 89%, and 75%, respectively. Under this workload, Translation presents a good index construction time and memory footprint for its underlying PSTBloom index. However, Translation is not a good solution for this workload because it takes an extra 14.6 s to translate arbitrary expressions to conjunctive expressions, and a maximum of 4,592 MB of memory is used to store the resulting conjunctive expressions. Overall, these experimental results are approximately consistent with the experimental results on synthetic workloads.

Table 4: Experiments on the Ads Dataset

Index	Matching	Construct	Memory
ATree	1.6 ms	2.9 s	205 MB
BDD	11.5 ms	25.6 s	823 MB
Translation	13.7 ms	11.1 + 14.6 s	320 + 4,592 MB
BoP	44.1 ms	8.6 s	684 MB
Interval ID	53.6 ms	12.2 s	1775 MB
Dewey ID	62.7 ms	14.8 s	1887 MB
SCAN	529.1 ms	-	-

## 7 CONCLUSIONS

In this paper, we proposed the dynamic A-Tree data structure to efficiently index tens of millions of ABEs. Based on the A-Tree index, we propose algorithms to quickly match events against ABEs. A-Tree-based event matching shows advantages over existing solutions in terms of *subexpression sharing*, its *multirooted tree-based indexing structure*, *dynamic index adjustment*, *access predicate and subexpression-based efficient filtering* and *non-exponential increase in size for expression transformation*. Detailed microexperiments show that all these design elements contribute to the performance footprint of A-Tree. We conducted extensive experiments using both synthetic datasets and real-world datasets. The results show that A-Tree outperforms existing solutions by a large margin.

## 8 ACKNOWLEDGMENTS

The authors are indebted to Mohammad Rubaiyat Ferdous Jewel who worked on a prior version of A-Tree. This work was in part supported by the Alexander von Humboldt Foundation, by NSERC, by the National Key R&D Program of China (#2017YFB1001804), and by NSF of China (#61732019).



## REFERENCES

- [1] M. K. Aguilera, R. E. Strom, D. C. Sturman, M. Astley, and T. D. Chandra. Matching events in a content-based subscription system. In *Proceedings of the eighteenth annual ACM symposium on Principles of distributed computing*, pages 53–61. ACM, 1999.
- [2] A. V. Aho, A. V. Aho, M. Ganapathi, and S. W. Tjiang. Code generation using tree matching and dynamic programming. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 11(4):491–516, 1989.
- [3] G. Ashayer, H. Leung, and H.-A. Jacobsen. Predicate Matching and Subscription Matching in Publish/Subscribe Systems. In *Distributed Event-based Systems (DEBS) Workshop at ICDCS*, pages 539–548, Vienna, Austria, May 2002.
- [4] A. AWS. <https://docs.aws.amazon.com/sns/latest/dg/sns-message-filtering.html>.
- [5] J. L. Bentley and D. Wood. An optimal worst case algorithm for reporting intersections of rectangles. *IEEE Trans. Computers*, C-29(7):571–576, July 1980.
- [6] S. Bittner and A. Hinze. The arbitrary boolean publish/subscribe model: making the case. In *Proceedings of the 2007 inaugural international conference on Distributed event-based systems*, pages 226–237. ACM, 2007.
- [7] L. Brenna, A. Demers, J. Gehrke, M. Hong, J. Ossher, B. Panda, M. Riedewald, M. Thatte, and W. White. Cayuga: a high-performance event processing engine. In *ACM SIGMOD*, pages 1100–1102, 2007.
- [8] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *Computers, IEEE Transactions on*, 100(8):677–691, 1986.
- [9] A. Campailla, S. Chaki, E. Clarke, S. Jha, and H. Veith. Efficient filtering in publish-subscribe systems using binary decision diagrams. In *Proceedings of the 23rd International Conference on Software Engineering*, pages 443–452. IEEE Computer Society, 2001.
- [10] A. F. CEP. <https://ci.apache.org/projects/flink/flink-docs-stable/dev/libs/cep.html>.
- [11] S. Ceri, R. Cochrane, and J. Widom. Practical applications of triggers and constraints: Success and lingering issues. In *PVLDB*, pages 254–262, 2000.
- [12] J. Claussen, A. Kemper, G. Moerkotte, K. Peithner, and M. Steinbrunn. Optimization and evaluation of disjunctive queries. *IEEE Transactions on Knowledge and Data Engineering*, 12(2):238–260, 2000.
- [13] A. de Castro Alves and J. Taylor. Event processing query language using pattern matching, Dec. 4 2008. US Patent App. 12/043,552.
- [14] Y. Diao, M. Altimel, M. J. Franklin, H. Zhang, and P. Fischer. Path sharing and predicate evaluation for high-performance xml filtering. *ACM TODS*, 28(4):467–516, 2003.
- [15] M. Elhemali, C. A. Galindo-Legaria, T. Grabs, and M. M. Joshi. Execution strategies for sql subqueries. In *ACM SIGMOD*, pages 993–1004. ACM, 2007.
- [16] F. Fabret, H. A. Jacobsen, F. Llibat, J. Pereira, K. A. Ross, and D. Shasha. Filtering algorithms and implementation for very fast publish/subscribe systems. In *ACM SIGMOD*, volume 30, pages 115–126. ACM, 2001.
- [17] 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 *INFOCOM 2016-The 35th Annual IEEE International Conference on Computer Communications, IEEE*, pages 1–9. IEEE, 2016.
- [18] E. Fidler, H.-A. Jacobsen, G. Li, and S. Mankovski. The padres distributed publish/subscribe system. In *FIW*, pages 12–30, 2005.
- [19] M. Fontoura, S. Sadanandan, J. Shanmugasundaram, S. Vassilvitskii, E. Vee, S. Venkatesan, and J. Zien. Efficiently evaluating complex boolean expressions. In *ACM SIGMOD*, pages 3–14. ACM, 2010.
- [20] M. Fontoura, J. Shanmugasundaram, S. Vassilvitskii, E. Vee, and J. Zien. System and method for efficiently evaluating complex boolean expressions, Sept. 15 2011. US Patent App. 12/724,415.
- [21] A. Gal and E. Hadar. Generic architecture of complex event processing systems. In *Principles and Applications of Distributed Event-Based Systems*, pages 1–18. IGI Global, 2010.
- [22] A. Ghosh, P. McAfee, K. Papineni, and S. Vassilvitskii. Bidding for representative allocations for display advertising. In *International workshop on internet and network economics*, pages 208–219. Springer, 2009.
- [23] D. Gyllstrom, E. Wu, H. Chae, Y. Diao, P. Stahlberg, and G. Anderson. SASE: complex event processing over streams (demo). In *CIDR*, pages 407–411, 2007.
- [24] E. N. Hanson. The interval skip list: A data structure for finding all intervals that overlap a point. In *Workshop on Algorithms and Data Structures*, pages 153–164. Springer, 1991.
- [25] E. N. Hanson, C. Carnes, L. Huang, M. Konyala, L. Noronha, S. Parthasarathy, J. Park, and A. Vernon. Scalable trigger processing. In *Proceedings 15th International Conference on Data Engineering (Cat. No. 99CB36337)*, pages 266–275. IEEE, 1999.
- [26] E. N. Hanson, M. Chaabouni, C.-H. Kim, and Y.-W. Wang. A predicate matching algorithm for database rule systems. *ACM SIGMOD*, 19(2):271–280, 1990.
- [27] E. N. Hanson, M. Chaabouni, C.-H. Kim, and Y.-W. Wang. A predicate matching algorithm for database rule systems. In *ACM SIGMOD*. ACM, 1990.
- [28] S. Huo and H.-A. Jacobsen. Predicate-based Filtering of XPath Expressions. In *IEEE International Conference on Data Engineering (ICDE)*, page 53, Atlanta, GA, April 2006.
- [29] M. Jarke. Common subexpression isolation in multiple query optimization. In *Query Processing in Database Systems*, pages 191–205. Springer, 1985.
- [30] S. Ji and H.-A. Jacobsen. Ps-tree-based efficient boolean expression matching for high-dimensional and dense workloads. *ACM SIGMOD*, 12(3):251–264, 2018.
- [31] G. Li, S. Huo, and H.-A. Jacobsen. A Unified Approach to Routing, Covering and Merging in Publish/Subscribe Systems Based on Modified Binary Decision Diagrams. In *25th International Conference on Distributed Computing Systems (ICDCS)*, pages 447–457. Columbus, Ohio, June 2005.
- [32] G. Li, S. Huo, and H.-A. Jacobsen. XML Routing in Data Dissemination Networks. In *IEEE 23rd International Conference on Data Engineering (ICDE)*, pages 1400–1404, April 2007.
- [33] G. Li, S. Huo, and H.-A. Jacobsen. Routing of XML and XPath Queries in Data Dissemination Networks. In *28th International Conference on Distributed Computing Systems (ICDCS)*, pages 627–638, June 2008.
- [34] G. Li and H.-A. Jacobsen. Composite subscriptions in content-based publish/subscribe systems. In *ACM/IFIP/USENIX International Conference on Distributed Systems Platforms and Open Distributed Processing*, pages 249–269. Springer, 2005.
- [35] 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.
- [36] A. Machanavajjhala, E. Vee, M. Garofalakis, and J. Shanmugasundaram. Scalable ranked publish/subscribe. *PVLDB*, 1(1):451–462, 2008.
- [37] E. M. McCreight. Efficient algorithms for enumerating intersecting intervals and rectangles. Technical Report Report CSL-80-9, Xerox PARC, Palo Alto, CA, 1980.
- [38] Y. Mei and S. Madden. Zstream: A cost-based query processor for adaptively detecting composite events. In *ACM SIGMOD*. ACM, 2009.
- [39] P. O’Neil, E. O’Neil, S. Pal, I. Cseri, G. Schaller, and N. Westbury. Ordpaths: insert-friendly xml node labels. In *ACM SIGMOD*, pages 903–908. ACM, 2004.
- [40] N. W. Paton and O. Díaz. Active database systems. *ACM Computing Surveys (CSUR)*, 31(1):63–103, 1999.
- [41] S. Qian, J. Cao, Y. Zhu, and M. Li. Rein: A fast event matching approach for content-based publish/subscribe systems. In *INFOCOM, 2014 Proceedings IEEE*, pages 2058–2066. IEEE, 2014.
- [42] M. Ray, C. Lei, and E. A. Rundensteiner. Scalable pattern sharing on event streams. In F. Özcan, G. Koutrika, and S. Madden, editors, *ACM SIGMOD*, pages 495–510. ACM, 2016.
- [43] M. Sadoghi, I. Burcea, and H.-A. Jacobsen. Gpx-matcher: a generic boolean predicate-based xpath expression matcher. In *Proceedings of the 14th International Conference on Extending Database Technology*, pages 45–56, 2011.
- [44] 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. ACM, 2011.
- [45] M. Sadoghi and H.-A. Jacobsen. Analysis and optimization for boolean expression indexing. *ACM TODS*, 38(2):8, 2013.
- [46] M. Sadoghi and H.-A. Jacobsen. Adaptive parallel compressed event matching. In *Data Engineering (ICDE), 2014 IEEE 30th International Conference on*, pages 364–375. IEEE, 2014.
- [47] B. Segall and D. Arnold. Elvin has left the building: A publish/subscribe notification service with quenching. In *AUUG*, 1997.
- [48] A. Surana, C. Trowbridge, I. Y.-h. Cheng, S. Poduval, and T. P. Scott. Ecommerce system with evaluation of boolean expression sets, Sept. 8 2020. US Patent 10,769,696.
- [49] I. Tatarinov, S. D. Viglas, K. Beyer, J. Shanmugasundaram, E. Shekita, and C. Zhang. Storing and querying ordered xml using a relational database system. In *ACM SIGMOD*, pages 204–215. ACM, 2002.
- [50] D. Wang, E. A. Rundensteiner, and R. T. Ellison III. Active complex event processing over event streams. *PVLDB*, 4(10):634–645, 2011.
- [51] I. Wegener. *The Complexity of Boolean Functions*. Wiley, 1987.
- [52] 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.
- [53] E. Wu, Y. Diao, and S. Rizvi. High-performance complex event processing over streams. In *ACM SIGMOD*, pages 407–418, 2006.
- [54] R. M. Wyman, T. Strohman, P. Haahr, L. Leavitt, and J. Sarapata. Predictive searching and associated cache management, Dec. 16 2010. US Patent App. 12/484,171.
- [55] T. W. Yan and H. Garcia-Molina. Index structures for selective dissemination of information under the boolean model. *ACM TODS*, 19(2):332–364, 1994.
- [56] D. Zhang, C.-Y. Chan, and K.-L. Tan. An efficient publish/subscribe index for e-commerce databases. *PVLDB*, 7(8):613–624, 2014.
- [57] W. Zhang, S. Yuan, and J. Wang. Optimal real-time bidding for display advertising. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 1077–1086. ACM, 2014.
- [58] Y. Zhao and J. Wu. Towards approximate event processing in a large-scale content-based network. In *Distributed Computing Systems (ICDCS), 2011 31st International Conference on*, pages 790–799. IEEE, 2011.