Efficiently Evaluating Complex Boolean Expressions

Marcus Fontoura Suhas Sadanandan Jayavel Shanmugasundaram
Sergei Vassilvitski Erik Vee Srihari Venkatesan Jason Zien
Yahoo! Research, 701 First Ave., Sunnyvale, CA 94089
{marcusf, suhas, jaishan, sergei, erikvee, venkates, jasonyz}@yahoo-inc.com

ABSTRACT

The problem of efficiently evaluating a large collection of complex Boolean expressions – beyond simple conjunctions and Disjunctive/Conjunctive Normal Forms (DNF/CNF) – occurs in many emerging online advertising applications such as advertising exchanges and automatic targeting. The simple solution of normalizing complex Boolean expressions to DNF or CNF form, and then using existing methods for evaluating such expressions is not always effective because of the exponential blow-up in the size of expressions due to normalization. We thus propose a novel method for evaluating complex expressions, which leverages existing techniques for evaluating leaf-level conjunctions, and then uses a bottom-up evaluation technique to only process the relevant parts of the complex expressions that contain the matching conjunctions. We develop two such bottom-up evaluation techniques, one based on Dewey IDs and another based on mapping Boolean expressions to one-dimensional intervals. Our experimental evaluation based on data obtained from an online advertising exchange shows that the proposed techniques are efficient and scalable, both with respect to space usage as well as evaluation time.

Categories and Subject Descriptors

H.2.4 [**Systems**]: Query processing

General Terms

Algorithms,Performance

Keywords

Boolean expressions, Pub/sub, Dewey, Interval

1. INTRODUCTION

We consider the problem of efficiently evaluating a large collection of arbitrarily complex Boolean expressions, given an assignment of attributes to values. The problem of efficiently evaluating Boolean expressions has many applications, including (1) publish/subscribe systems [8], where a subscription can be modeled as a Boolean expressions and an event can be modeled as a collection of attribute-value pairs, and the goal is to rapidly return the subscriptions that match an event, and (2) online display advertising systems [10, 13, 18], where an advertiser campaign can be modeled as a Boolean expression targeting user visit features, and a user visit can be modeled as a collection of attribute-value pairs, and the goal is to rapidly return the set of advertiser campaigns that are eligible for the user visit. Current solutions for evaluating Boolean expressions, however, are primarily limited to simple Boolean expressions such as conjunctions [1, 9, 14, 20, 21] and conjunctive/disjunctive normal form expressions [5, 17]. While these restrictions are reasonable for many of the above applications, some emerging applications require support for arbitrarily complex Boolean expressions, as discussed below.

1.1 Motivating Applications

Online Display Advertising Exchanges. One of the emerging trends in online display advertising is the notion of an advertising exchange, or simply, an ad exchange. An ad exchange is essentially an electronic hub that connects online publishers to advertisers, either directly or through intermediaries. An ad exchange can thus be represented as a directed graph, where the nodes are publishers, advertisers and intermediaries, and an edge exists between a node X and a node Y if X agrees to sell advertisement slots associated with user visits to Y (note that X may either be a publisher who generates advertisement slots through user visits, or an intermediary who obtains user visits from publishers or other intermediaries). In addition, each edge is typically annotated with a Boolean expression that restricts the set of user visits that can be sold through that edge, for various business reasons such as protecting certain user visits for sales through specific channels (edges). For instance, a node X may only want to sell to Y male user visits to Sports pages, and female user visits that are not to Finance pages; this could be represented as a Boolean expression on the X-Y edge: $(Gender \in \{Male\} \wedge Category \in \{Sports\}) \vee (Gender \in \{Female\} \wedge Category \notin \{Finance\}$.

Given the ad exchange graph, an advertiser A can decide to book an advertising campaign with a publisher P, even though A may only be indirectly connected to P. For instance, if an advertiser A wants to book a campaign with P that targets users in age category 4 who are interested in NFL, and age category 5 who are interested in NBA,

SIGMOD'10, June 6–11, 2010, Indianapolis, Indiana, USA.

this campaign can also be represented as a Boolean expression: $(Age \in \{4\} \wedge Interest \in \{NFL\}) \wedge (Age \in \{5\} \wedge Interest \in \{NBA\})$. Note, however, that any user visit from P that satisfies the above Boolean expression cannot simply be served the ad from A; the user visit should also satisfy all the Boolean expressions on the *path* from P to A[1]. Consequently, the campaign booked with P is the conjunction of the advertiser campaign Boolean expression, and the Boolean expressions along the path from P to A. Since advertiser and targeting constraints can themselves be complicated DNF or other expressions, conjunctions of such expressions quickly leads to fairly complex Boolean expressions which are booked to publishers.

When a user visits a publisher Web page, the user visit can be viewed as an attribute-value assignment such as the one below:

$\{Gender = Male, Interest = NFL, Category = Sports\}$

and the goal is to rapidly find all the ad campaigns booked to P that can be satisfied by the user visit, so that the best ads can then be selected to be shown to the user. In other words, the exchange has to rapidly evaluate complex Boolean expressions to determine which one satisfy the given assignment of attributes to values.

Automatic user targeting. A related application that generates complex Boolean expressions is automatic user targeting for display advertising. Unlike manual user targeting where advertisers specify the Boolean expression of interest (as in the examples above), in automatic user targeting, the system automatically generates targeting expressions that try to maximize advertiser objectives such as clicks or conversions. For instance, an advertiser who is interested in obtaining clicks on ads may specify a fairly broad targeting constraint, and allow the system to explore the high dimensional targeting space to generate boolean targeting expressions that optimize desired measurement objectives. Clearly these can get quite complex very quickly because they are automatically generated. Given many such advertiser campaigns, the advertising system again needs to rapidly evaluate these Boolean expressions given an attribute assignment (user visit).

1.2 Contributions

Given the above motivating applications, we now turn to the issue of efficiently evaluating arbitrarily complex Boolean expressions. A simple evaluation method, of course, is to sequentially loop over all the Boolean expressions, and do a recursive top-down evaluation of the expression tree given the attribute value assignment. This method has the obvious downside of having to evaluate every single expression, even though the assignment may only match a small fraction of them. Another simple method is convert the Boolean expressions to DNF or CNF form, and leverage state-of-the-art techniques (e.g., [5, 17]) to efficiently evaluate these expressions. Again, this method has the downside of an exponential blow-up in the size the expressions due to normalization; this issue is exacerbated by the fact that most online ad systems are entirely memory-resident (for latency reasons), which leads to excessive memory requirements. Given the limitations of the obvious approaches, the question that arises is the following: *is there a way to evaluate Boolean expressions that does not require evaluating every expression, and that does not result in an exponential space blow-up?*

The main technical contribution of this paper is a novel evaluation method that addresses the above issues. The method consists of two key steps. In the first step, existing conjunction matching techniques [9, 17, 21] are used to find the leaf-level conjunctions of the (un-normalized) Boolean expressions that match the given assignment. In addition, each conjunction is annotated with a compact description of where it occurs in the Boolean expressions. In the second step, the matching conjunctions along with the information on where they occur is used to perform a *bottom-up* evaluation of the Boolean expressions. The bottom-up evaluation is performed in a *single pass* over the matching conjunctions, and only selectively evaluates the expressions — and parts of these expressions — that have at least one matching conjunction. The above two-step approach thus leverages existing conjunction matching techniques without blowing up the size of the expressions, and also avoids explicitly evaluating all expressions by using selective bottom-up evaluation of only those (parts of) expressions that can possibly be satisfied.

As mentioned above, the key idea that enables the bottom-up evaluation of Boolean expressions is the annotation that identifies the position of each conjunction within the Boolean expression. There are two annotation variants that we consider, both of which work on the Boolean tree representation of expressions. In the first variant, each conjunction is identified based on a Dewey labeling of the Boolean expression tree (similar to the Dewey labeling of an XML tree [16]). Given this labeling, the bottom-up evaluator uses a stack-based algorithm to efficiently find the ids of the contracts that evaluate to true. In the second variant, each Boolean expression tree is mapped to a one-dimensional space, and the bottom-up evaluator uses a simple interval-matching technique to find the ids of the matching contracts. While both approaches are efficient, one of the advantages of the one-dimensional mapping is that the conjunction annotations are fixed-length, as compared to variable-length Dewey labels.

We have implemented the proposed methods and evaluated them using data obtained from an online display advertising exchange. Our performance results show that the proposed methods significantly outperform existing methods, both in terms of latency and memory requirements.

1.3 Roadmap

The rest of the paper is organized as follows. In Section 2, we describe the problem and the system architecture. In Section 3, we present the evaluation method based on Dewey labeling, and in Section 4, we present the evaluation method based on the one-dimensional interval mapping. In Section 5, we present our experimental results and, in Section 6, we discuss related work. Finally, in Section 7, we present our conclusions.

2. PROBLEM DESCRIPTION

Our problem is to efficiently find which Boolean expressions from a large set are satisfied by an input assignment. An assignment is a set of attribute name and value pairs $\{A_1 = v_1, A_2 = v_2, \ldots\}$. For example, a woman in California may have the assignment $\{Gender = F, State = CA\}$.

[1]There might be multiple paths from P to A and the best path is usually chosen based on revenue and other constraints that are not germane to the current discussion.

An assignment does not necessarily specify all the possible attributes. Allowing unspecified attributes is important to support high-dimensional data where the number of attributes may be in the order of hundreds. Consequently, our model does not restrict assignments to use a fixed set of possible attributes known in advance.

A Boolean expression (BE) is a tree in which intermediate nodes are of two types: AND nodes and OR nodes. Leaf nodes in the tree are simple conjunctions of basic $\in$ and $\notin$ predicates. The predicate $State \in \{CA, NY\}$, for example, means that the state can either be California or New York while the predicate $State \notin \{CA, NY\}$ means the state cannot be either of the two states. Notice that the $\in$ and $\notin$ primitives subsume simple $=$ and $\neq$ predicates. Without loss of generality, we restrict our BE trees to have alternating AND-OR nodes in every path from the root to the leafs. Any arbitrarily complex BE can be represented by these alternating AND-OR trees with conjunction leafs, including DNFs (i.e., disjunctive normal form), CNFs (i.e., conjunctive normal form), ANDs of DNFs, ORs of CNFs, and so on.

2.1 System Architecture

The overall system architecture is presented in Figure 1. In an offline process, before query evaluation starts, BEs are annotated and indexed. The Conjunction Annotator module is responsible for annotating each conjunction with a compact description of where it occurs in the BE. These annotations are stored in a Conjunction Annotations database. The conjunctions are then indexed by the Conjunction Index Builder. Our approach works with any existing scheme for indexing and evaluating conjunctions, e.g. [1, 9, 14, 17]. During runtime, given an assignment, the index is used to retrieve the matching conjunctions. Given these set of matching conjunctions, the Expression Evaluator uses the Conjunction Annotations database to retrieve the annotations for the conjunctions that need to be evaluated. The job of the Expression Evaluator is to efficiently verify if the entire BE tree can be satisfied from the conjunctions retrieved by the index. We highlighted components Expression Evaluator and Expression Annotator since these are the main contributions of the paper. Sections 3 and 4 describe two different annotation schemes and evaluation strategies for these components.

Scalability, latency and updates. The main focus points of this paper are the Expression Evaluator and Conjunction Annotator components and we can reuse any conjunction indexing scheme. Each of these different schemes will handle scalability, latency and updates differently. Our driving applications are online advertising systems, which have to process billions of requests a day. However, the update volume is typically many orders of magnitude less than the read volume. Fortunately, there are several conjunction indexing schemes optimized for this scenario, e.g., [17]. For instance, scalability can be solved by index replication and partitioning, latency can be solved by keeping the indexes in main memory, while updates can handled by keeping small "delta" indexes in addition to the main indexes [17].

3. DEWEY ID MATCHING

Our first algorithm uses the notion of Dewey IDs to perform boolean expression matching. We first describe how Dewey IDs are generated, and then how they are used in evaluating BEs. The main challenge comes from the fact that we *do not* store the Boolean Expression Tree for evaluation. Rather, we reconstruct the relevant parts of the tree only from the information encoded in the *matching* Dewey IDs and decide whether the overall tree evaluates to true.

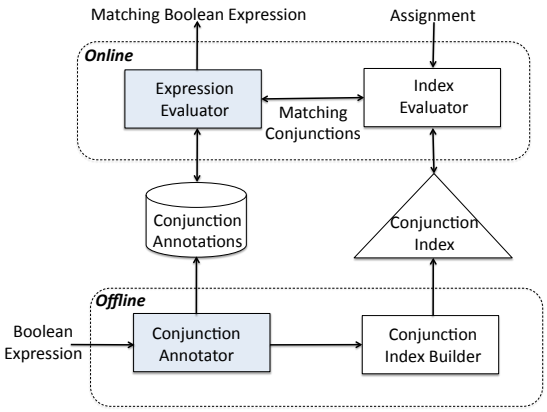

Figure 1: Online and offline architectural view of the system. We focus on the highlighted components, Expression Evaluator and Conjunction Annotator.

3.1 Conjunction Annotator

We first describe the information stored with each conjunction. As we mentioned earlier, any BE can be expressed as an alternating AND-OR Boolean tree. We label each leaf node of the tree with its corresponding Dewey ID as follows:

1. Without loss of generality let the root of the tree be an AND node. We can always add an artificial AND at the top if needed.
2. Let edges to the children of a node be sequentially numbered starting from 1, with the last child marked with a special symbol $*$. The root-to-node numbering (based on those edge numbers) is referred to as the Dewey ID of a node.
3. The length of a Dewey ID is the number of edges from the root to the node. Observe that a node with an odd length Dewey ID is beneath an AND, and a node with an even length Dewey ID is beneath an OR.

For example, consider the BE tree in Figure 2. The label of D is $1*.3.1$ – to reach D from the root, one takes the first branch (which happens to be the last branch as well, as denoted by $*$), then the third branch, and then the first branch again. The Dewey IDs of other leaves are given in the Figure.

3.2 Expression Evaluator

We first give intuition on the functioning of the algorithm. We then describe it in more detail.

The algorithm acts recursively on the Boolean expression tree. Note that this tree is not actually available during online processing. We only have the list of matching dewey IDs. However, these dewey IDs implicitly encode this tree (or more precisely, the portion of the tree where the dewey IDs lie). As we process the IDs, we create this "virtual" tree on the fly.

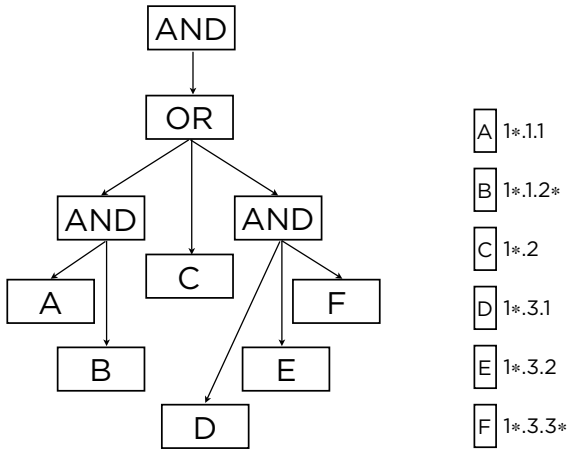

Figure 2: An example of a BE tree with Dewey ID labels. The special symbol ∗ indicates the last child of an AND node.

Algorithm 1 The DEWEY EVALUATION ALGORITHM

Require: `deweyList` {*a list of dewey IDs in sorted order*}
1: Initialize `curr` ← `deweyList`.getFirst() {*`curr` and `deweyList` are global variables*}
2: **return** EVALUATEAND(Empty DeweyId)

Throughout the running of the algorithm, we alternate calls to EVALUATEAND and EVALUATEOR. Each call takes as input a dewey label, which we call `dewLabel`. We think of each call as corresponding to exploring a node in the Boolean expression tree, and this label corresponds precisely to the dewey ID of that node. A call to EVALUATEAND is like exploring an AND node of the tree, while EVALUATEOR corresponds to exploring an OR node.

We iterate through the list of dewey IDs, in sorted order. The value of `curr` is the dewey ID we are currently considering. Note the `curr` corresponds to a leaf node in the Boolean expression tree.

At this point, it is helpful to imagine a depth-first traversal of the virtual nodes of the Boolean expression tree. If we are exploring a node in the virtual tree that is an ancestor of the leaf node corresponding to `curr`(which is equivalent to the dewey label in the current call being a prefix of `curr`), then we move down toward that leaf.

When we reach a leaf, we evaluate it `true`, since the index only returns those conjunctions which evaluate to true. We then pop up a level, partially evaluating that node. The `curr` dewey ID is updated to the next dewey ID in the list. We continuing popping up levels (evaluating nodes as we go), until we reach an ancestor of the newly updated `curr`.

Continuing in this manner allows us to evaluate the entire expression.

We now walk through the algorithm. Pseudo-code is shown in Algorithms 1,2, and, 3. We use several helper functions. First, CHILD, which takes as input a dewey prefix and a dewey ID. (The input prefix must be a prefix of the input dewey ID.) It returns the dewey ID of entry where they first differ. For example, CHILD(0.1.2, 0.1.2.3.4) returns 0.1.2.3. Note that CHILD returns the Dewey label for one of the children of the dewey prefix. The other two functions, work on the dewey IDs, LAST returns the value of the last

Algorithm 2 The EVALUATEOR Algorithm

Require: `dewLabel` {current position in the tree}
1: **if** `dewLabel` = `curr` **then** {*We are at a leaf*}
2: **return true**
3: **end if**
4: Initialize `result` ← **false**
5: **while** `dewLabel` is a prefix for `curr` **do** {*`curr` is a descendant of this node*}
6: Let `child` ← CHILD(`dewLabel`, `curr`)
7: `result` ← `result` ∨ EVALUATEAND(`child`)
8: `curr` ← `deweyList`.next()
9: **end while**
10: `curr` ← `deweyList`.prev()
11: **return** `result`

Algorithm 3 The EVALUATEAND Algorithm

Require: `dewLabel` {current position in the tree}
1: **if** `dewLabel` = `curr` **then** {*We are at a leaf*}
2: **return true**
3: **end if**
4: Initialize `result` ← **true**, `lastExplored` ← 0, and `lastChild`←**false**
5: **while** `dewLabel` is a prefix for `curr` **do** {`curr` *is a descendant of this node*}
6: Let `child` ← CHILD(`dewLabel`, `curr`)
7: `lastExplored` ← `lastExplored` + 1
8: **if** LAST(`child`) ≠ `lastExplored` **then**
9: `result` ← **false**
10: **end if**
11: `lastChild`←MARKED(`child`)
12: `result` ← `result` ∧ EVALUATEOR(`child`)
13: `curr` ← `deweyList`.next()
14: **end while**
15: `curr` ← `deweyList`.prev()
16: **return** `result` & `lastChild`

id. For example LAST(0.1.2) returns 2. Finally, MARKED returns `true` if the last node of the Dewey id is marked with a ∗ and `false` otherwise.

The algorithm is initialized by setting `curr` to the first element in the sorted deweyID list. It then calls EVALUATEAND, with input dewey label of "Empty DeweyID." We think of this first call as moving to the root node of the Boolean expression tree.

Now, within the EVALUATEAND call, our base case (Steps 1 to 3) corresponds to being at a virtual leaf node, in which case we return `true`.

The while loop (Step 5) iterates through all ancestors in the dewey list of the node currently being explored. It does this by first exploring the child under which the `curr` dewey ID lies. Thus, we recursively call EVALUATEOR with its label corresponding to the child of the currently explored node. After this evaluation takes place, we AND its result with our result so far. Note that `curr` may have been updated during the EVALUATEOR call. We continue to iterate through each of the children.

In the call to EVALUATEAND, we need every child to evaluate to true. We ensure that every child is explored my maintaining `lastExplored`, and checking that we never jump over a child (Steps 7 to 10). We also check that we encounter a starred dewey ID along the way.

3.3 Example

We walk through an example of the Evaluation algorithm. Let the set of Dewy IDs presented to the expression evaluator be:

1∗.1.1, 1∗.3.1, 1∗.3.2, 1∗.3.3∗

Is the BE satisfied?

The Dewey IDs represent nodes A, D, E and F in Figure 2, so it is easy to see that the expression is satisfied. However, remember that the evaluation algorithm does *not know* what the tree for the BE was, it only sees the matching Dewey IDs.

The Dewey evaluation algorithm first looks at id 1∗.1.1, and recursively calls EvaluateAND and EvaluateOR until it reaches the leaf (A). It then pops up a level to the AND at position 1∗.1 and increments `curr`. Since the next id, 1∗.3.1 does not have 1∗.1 as a prefix, the evaluation stops, and the AND is evaluated to false since the `lastChild` was never set to `true`.

The algorithm then proceeds to evaluate the AND at position 1∗.3. It successfully evaluates all of the leaves, at which point the `result` is set to `true` and so is `lastChild`, the latter being set to `true` during the evaluation of 1∗.3.3∗, since the id ends in the special symbol ∗. Therefore the OR at position 1∗ is set to true as well. Finally, the original call to EvaluateAND returns with `true`.

3.4 Correctness

Theorem 1. *The* Dewey Evaluation *algorithm is correct.*

Proof. The proof of correctness follows quickly from the recursive nature of the algorithm. We sketch the proof that EvaluateAND and EvaluateOR both evaluate correctly, and further, the value of `curr` after the call is set to the last dewey ID for which the dewey label of the call is a prefix.

Clearly, the algorithm produces the correct result when the Boolean expression tree is a single node. By induction, assume that the algorithm works on a tree of depth $d-1$, and consider a tree of depth d. There are two cases, whether the top level node is an AND or an OR. Suppose it is an AND (the OR case is even simpler). In the call to EvaluateAND, we call EvaluateOR iteratively for each child of the explored node. By induction, each of these calls returns the correct result. The method returns `false` if one of these subroutines returned `false` (since all of the results are ANDed together), one of the children is skipped (the check of `lastExplored`), or if the final child was not seen (the check of `lastChild`). Otherwise we can conclude that all of the children of the AND returned `true`, and thus this node evaluates to `true` as well. □

Finally, we note the running time:

Theorem 2. *Let ℓ be the set of leaves returned, and for a leaf $n \in \ell$ denote by $len(n)$ the length of the Dewey ID of n. Then the* Dewey Evaluation *algorithm runs in time $O(\sum_{n\in\ell} len(n))$.*

Proof. The running time follows from the fact that we evaluate each of the returned Dewey IDs one level at a time, so the time to process an id n is proportional to $len(n)$. □

We note that while the pseudocode presented is not optimized (one can, for example exit the EvaluateAND loop as soon as the result is set to `false`), this does not change the worst case running time of the algorithm.

4. INTERVAL IDS

We describe an alternative algorithm for evaluating BE trees. At a high level the algorithm works by mapping each leaf node of the BE tree onto a 1-dimensional interval on the real line. A contract is satisfied if there is a subset of intervals that cover the real line without overlap. At first glance it sounds like we have made our problem more difficult, however, the matching algorithm is simpler and more intuitive than the Dewey ID evaluation algorithm. Moreover, for each conjunction we need to store only two fixed length values, namely the beginning and the end of the corresponding interval. Hence, the amount of the stored information is constant.

4.1 Intuition

Consider an arbitrary BE tree, we will map the leaf nodes of the tree to intervals on the real line. We denote an interval $[s,t)$ as $\langle s,t\rangle$. (In what follows s and t will always be integers.) Fix M to be the maximum number of leaves in a BE tree. We will represent each contract by a line segment $\langle 1, M\rangle$. Each leaf of the tree will be mapped to a subinterval of $\langle 1, M\rangle$. The key to the algorithm lies in the mapping of conjunctions to intervals. We aim to find a mapping so that a contract is satisfied if and only if there exists a subset of satisfied leaves covering the entire segment $\langle 1, M\rangle$ without overlap.

To develop the intuition, we first describe two simple cases. Consider a hypothetical Boolean expression, $A \vee B$ shown in Figure 3. The contract is satisfied if either of the two leaves is satisfied. Therefore, the interval mapping scheme assigns the same interval $\langle 1, M\rangle$ to both A and B.

Invariant 1. *Consider a BE tree, and a node* `n` *corresponding to an OR. Then every child of* `n` *has the same interval as* `n`.

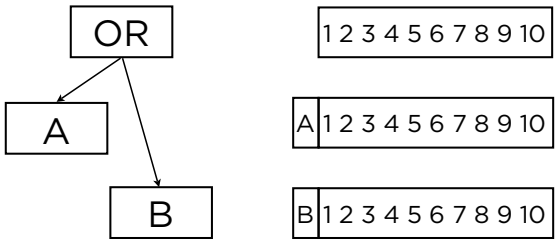

Figure 3: The children of an OR node inherit the same interval as the parent.

The situation is the opposite for an AND node. Consider a hypothetical Boolean expression, $A \wedge B$ shown in Figure 4. The contract is satisfied only if both of the leaves are satisfied. The interval mapping scheme splits the interval $\langle 1, M\rangle$ of the parent node into two non-overlapping intervals, $\langle 1, x\rangle$ and $\langle x, M\rangle$ for the children. We describe the exact choice for x later. More generally, this leads to a second invariant:

Invariant 2. *Consider a BE tree, and a node* `n` *corresponding to an AND. Then the interval corresponding to* `n` *is partitioned among its children.*

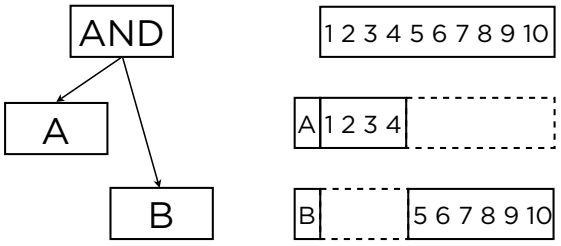

Figure 4: The children of an AND node partition the interval.

It is straightforward to apply these two invariants recursively. A more complicated example, corresponding to $(A \wedge B) \vee C \vee (D \wedge E \wedge F)$ is shown in Figure 5. Notice that there are three ways the tree can be satisfied. Either both A and B are satisfied, C is satisfied, or all three of D, E and F are satisfied. This example also demonstrates why we must find a set of *non-overlapping* intervals that fully cover $\langle 1, 10\rangle$. If B and D are returned, then $\langle 1, 10\rangle \subseteq \langle 1, 4\rangle \cup \langle 2, 10\rangle$, that is, together B and D cover the whole interval. However, since B and D overlap, this is not an eligible assignment.

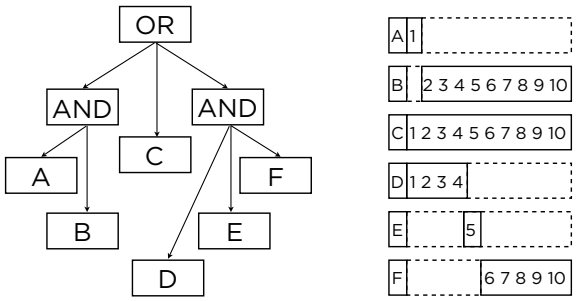

Figure 5: A more complex example of intervals.

Another potential pitfall in assigning intervals is shown in Figure 6. Here because B and E share the same starting point, an assignment of B and D would evaluate to true: together B and D cover the whole interval $\langle 1, 10\rangle$ without overlap. Thus we have the third invariant :

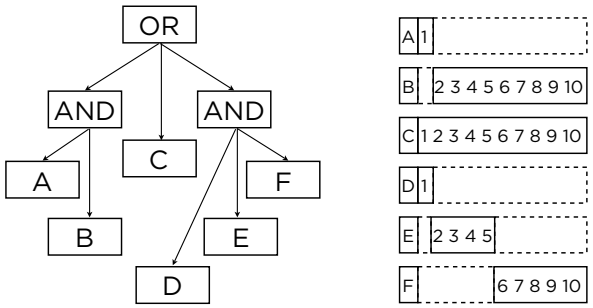

Figure 6: An invalid labeling. If B and D are true, the full interval is covered without overlap.

Invariant 3. *Given a boolean expression tree, and let C be the set of children of AND nodes. Let F be the set of first (left-most) children of AND nodes. Then no two nodes in $C \setminus F$ have the same starting point for their intervals.*

The last invariant precisely precludes the case in Figure 6. Since both B and E are second children of an AND node, their corresponding intervals must start at different positions. We will refer to the assignment of intervals to the leaves of a tree as the labeling of the tree.

Definition 1. *We call a labeling* valid *if Invariants 1, 2 and 3 are maintained.*

We describe an algorithm for generating a valid labeling in the next section, and in Section 4.3 show how we can quickly evaluate whether the full BE is true, based only on the intervals corresponding to the satisfied conjunctions.

4.2 Conjunction Annotation

In this section we describe an algorithm for providing a valid labeling of the nodes of the tree. Let the size of a node be the total number of children in its subtree (with size of a leaf node set to 1). Further, for each node `n`, let `n`.leftLeaves denote the total number of leaves appearing before `n` in a pre-order traversal of the tree.

The algorithm is recursive, it starts by labeling the root node with interval $\langle 1, M\rangle$ where M is the maximum number of leaves supported, and then calling the subroutine presented in Algorithm 4.

Algorithm 4 The Label algorithm

```
Require: Node n.
 1: if n is a leaf then
 2:   return
 3: else if n is an OR node then
 4:   for all children c of n do
 5:     c.begin ← n.begin
 6:     c.end ← n.end
 7:     Label(c)
 8:   end for
 9: else if n is an AND node then
10:   for first child c do
11:     c.begin ← n.begin
12:     c.end ← n.leftLeaves + c.size
13:     Label(c)
14:     curr ← c.end+1
15:   end for
16:   for all intermediate children c of n do
17:     c.begin ← curr;
18:     c.end ← curr + c.size -1;
19:     Label(c)
20:     curr ← c.end+1
21:   end for
22:   for last child c do
23:     c.begin ← curr;
24:     c.end ← n.end;
25:     Label(c)
26:   end for
27: end if
```

For example, consider the Tree in Figure 5 with $M = 10$. When labeling node A, we have `n`.leftLeaves $= 0$, since there are no prior leaf nodes in an in-order traversal of the tree. Therefore the interval for A is $\langle 1, 1\rangle$. B is relegated the rest

Interval	matched										
	0	1	2	3	4	5	6	7	8	9	10
Initial	1	0	0	0	0	0	0	0	0	0	0
A = $\langle 1, 1\rangle$	1	1	0	0	0	0	0	0	0	0	0
D = $\langle 1, 4\rangle$	1	1	0	0	1	0	0	0	0	0	0
E = $\langle 5, 5\rangle$	1	1	0	0	1	1	0	0	0	0	0
F = $\langle 6, 10\rangle$	1	1	0	0	1	1	0	0	0	0	1

Table 1: The `matched` array during the evaluation of the algorithm on A, D, E, F. For instance, row D shows the array `matched` after adding the interval for D.

of the interval, $\langle 2, 10\rangle$. The interval for C is easy, since it is a child of an OR node, it must be $\langle 1, 10\rangle$. Now consider the label for D. For D's parent node, `n`.leftLeaves is set to three, therefore the endpoint of the interval for D is 1+3 = 4. The intervals for E and F follow. Note that the labeling of the tree can be constructed in a single in-order traversal of the BE tree.

4.3 Expression Evaluation

The input to the evaluation algorithm is a set of intervals, one for each matching conjunction. The algorithm attempts to find a non-overlapping set of intervals that cover the range $\langle 1, M\rangle$. To do so, the algorithm will maintain a Boolean array `matched`, where `matched`[i] is true if there is a non-overlapping set of intervals that ends in i. We give the full matching algorithm below:

Algorithm 5 MATCH Algorithm.

Require: I: set of intervals $\langle begin, end\rangle$ sorted by *begin*
1: `matched` ← Boolean Array of length $M + 1$
2: Initialize `matched`[i] to **false** for all i
3: `matched`[0] = **true**
4: **for all** intervals $\langle begin, end\rangle$ in I **do**
5: **if** `matched`[*begin*-1] **then**
6: `matched`[*end*] ← **true**
7: **end if**
8: **end for**
9: **if** `matched`[M] **then**
10: **return true**
11: **else**
12: **return false**
13: **end if**

Consider again the example in Figure 5, and suppose that A, D, E, F were returned as matching conjunctions. The algorithm maintains the state of the `matched` array, with the individual steps presented in Table 1. Note that processing each interval requires only two probes into the boolean array.

4.4 Correctness

In this section we prove that both the labeling algorithm and the label evaluation algorithms are correct.

THEOREM 3. *The* LABEL *Algorithm produces a* valid *labeling.*

PROOF. It is easy to see that Invariants 1 and 2 are trivially satisfied, that is every child of an OR node has the same interval as the parent, and the children of an AND node partition the interval among themselves. It remains to show invariant 3. Let C be the set of children of AND nodes, and F be the set of first, or leftmost children. We show that the interval corresponding to every node $n \in C \setminus F$ starts at `n`.leftLeaves + 1. Recall that `n`.leftLeaves is the number of leaves occurring before `n` in a pre-order traversal of the tree. If this holds than no two nodes in $C \setminus F$ can have the same starting points for intervals.

We prove the claim by induction on the depth of the node. For the base case, consider the root node `n`, and let $\mathtt{c}_1, \mathtt{c}_2, \ldots, \mathtt{c}_k$ be its k children. Observe that $\mathtt{c}_i$.leftLeaves = $\sum_{j=1}^{i-1} \mathtt{c}_j$.size, and the base case follows. Suppose that the claim holds for nodes at depth d. Let `n` be a node at depth $d+1$ and $\mathtt{c}_1, \mathtt{c}_2, \ldots, \mathtt{c}_k$ be its k children. Since $\mathtt{c}_i$.leftLeaves = `n`.leftLeaves + $\sum_{j=1}^{i-1} \mathtt{c}_j$.size, the theorem follows. □

We now show that the matching algorithm is correct. We begin with a theorem about valid labelings.

THEOREM 4. *Consider a BE tree with a valid labeling. Let I be the set of intervals corresponding to leaf nodes in N that evaluate to true. Then the BE is satisfied iff there exists a subset $I' \subset I$ such that $\cup_{i \in I'} \langle begin_i, end_i\rangle = \langle 1, M\rangle$ and any two intervals $i, i' \in I'$ are disjoint: $\langle begin_i, end_i\rangle \cap \langle begin_{i'}, end_{i'}\rangle = \emptyset$.*

To prove the theorem, we first show that given a satisfying assignment to the BE we can find a set of intervals I' as described above. We then prove the converse, that is, given a set of intervals satisfying the condition above, we show that the BE must be satisfied.

PROOF. To prove the forward direction, consider the minimal set of leaves of the tree that lead to a satisfied assignment (a set is minimal if removing any element would lead to the BE evaluating as false). Then the set of corresponding intervals covers the whole segment $\langle 1, M\rangle$ and is non-overlapping. The formal proof is by induction on the height of the tree: since an AND partitions the interval, to be satisfied all of its children must be satisfied, therefore its interval would be fully covered by its children. On the other hand, since all of the children of an OR inherit the same interval, only one of the children needs to be satisfied (otherwise the initial set of trees is not minimal). Therefore the intervals corresponding to the minimal set of leaves satisfy the conditions of the theorem.

To prove the reverse direction, the main obstacle is to show that an interval corresponding to an AND node can only be fully covered without overlap by the nodes corresponding to its children. This is guaranteed by Invariant 3. Since the starting points of the intervals of all intermediate AND children are disjoint, and AND node can only be satisfied by its children because no subset of other children would result in a continuous and non-overlapping interval. The formal proof again proceeds by induction on the height of the tree rotted at the AND node. □

Finally, we can prove the correctness of the MATCH algorithm. Let ℓ be the number of leaves of the tree returned by the indexing system.

THEOREM 5. *The* MATCH *algorithm is correct and runs in time $O(\ell)$.*

PROOF. To prove correctness, we show that the MATCHING algorithm finds a non-overlapping subset of intervals covering the whole interval if one exists. The invariant maintained by the algorithm is that `matched`[i] is set to true only if there exists a non-overlapping set of intervals that cover

the interval $\langle 1, i\rangle$. When processing an individual interval $i = \langle begin, end\rangle$, we check to see if *begin* continues a previous interval, in which case `matched`[end] is set to true. Since the intervals are sorted by the *begin* position when processing, all intervals that end before *begin* will have been processed before (since they must begin even earlier).

To show the running time, observe that each evaluation of an interval results in at most two lookups into a boolean array. □

4.5 Discussion

The Interval evaluation algorithm has a number of appealing properties that improve upon the Dewey ID algorithm.

- It can handle trees of very large depth with a fixed size Interval ID. In contrast, the Dewey ID grows linearly with the depth of the tree. If the total number of leaves is n, the space taken by storing the interval is $O(\log n)$, whereas the space required for some Dewey ids may be as large is $\Omega(n)$.
- Although we described the algorithm in the context of AND-OR trees, it can naturally handle arbitrary BE trees without any change to the code.
- Finally, the Interval algorithm is faster, requiring only two memory look up calls for each interval, instead of being linear in the size of the Dewey IDs.

5. EXPERIMENTS

In this section, we evaluate our Dewey and Interval matching algorithms for BE evaluation on synthetic datasets from an online advertising application. We compare their performance to other efficient algorithms for BE evaluation and study how the algorithms behave for different scenarios, such as BE tree depth and selectivity. All algorithms were implemented in C++, and our experiments were run on a 2.5GHz Intel(R) Xeon(R) processor with 16GB of RAM.

5.1 Data set

In order to test the efficiency of the Dewey evaluation, Interval evaluation and other algorithms, we used a synthetic dataset generated from statistics of real advertising contracts. To gather the statistics we looked at individual conjunctions appearing in each contract. For example, a contract looking for Males from California would specify $\texttt{Gender} \in \{\texttt{Male}\} \wedge \texttt{State} \in \{\texttt{CA}\}$. We first collected statistics over the size of these conjunctions. We denote by c_i the probability of seeing a conjunction on i elements. We then recorded how often each attribute i (e.g. `gender, state,` etc) is present in the contracts, which we denote by a_i. For example, if half the contracts targeted on `gender` then $a_{gender} = 1/2$. For each attribute i, we collected statistics on the targeting values for this attribute. We denote the distribution by $p_{attribute}(target)$. For example, if, from the set of contracts targeting on gender, 2/3 targeted males, and 1/3 targeted females, then we say $p_{gender}(male) = 2/3$ and $p_{gender}(female) = 1/3$.

These statistics served as input to the BE generator. We first generated the logical BE tree, namely an alternating AND-OR tree. Without loss of generality the root node was selected to be an AND. The tree was then generated recursively. For each node, we first decide how many children the node will have. If it has 0, then we stop and mark the node as a conjunction; otherwise we generate the children, mark them as OR nodes (or in the case of processing an OR node, we mark the children as AND nodes), and recurse.

The input to the tree generator specified the minimum and maximum depths of the desired tree, and the probability distribution on the number of children. If the node being generated is below the minimum depth, then the number of children is set to be non-zero. If the node is at the maximum depth, then the number of children is set to 0. Otherwise, we select from the distribution. In our experiments, the number of children was 2 with probability 0.7, 3 with probability 0.2, and 1 or 4 with probability 0.05. This resulted in a wide distribution on the trees.

Given the tree structure, we then generated a conjunction for each leaf node of the tree. To generate a conjunction we randomly select its size, selecting size s with probability c_s. Next for each of the s slots, we randomly select an attribute with probability a_i. The attribute selection is done without replacement. Finally, for each attribute i we select a target with probability $p_i(\cdot)$.

For the evaluation test, the queries came from a random subset of 2,000 real display advertising adlogs. Each query represents the information of a user visiting a webpage, and so specifies the feature values that are known about the user. For example, a query may look like $\texttt{gender} = \texttt{M} \wedge \texttt{State} = \texttt{NY}$.

5.2 Algorithms

We have implemented the Dewey and Interval evaluation algorithms using the Conjunction Evaluation algorithm described [17] as the Index Evaluator component (Figure 1). We have chosen [17] since it was shown to outperform other state-of-the-art algorithms, such as Le Subscribe [9]. We compare the Dewey and Interval algorithms with the algorithms below.

- **DNF expansion:** We used the DNF evaluation algorithm described in [17] to evaluate BE trees by converting each BE tree into a single DNF. As shown in the experimental results, this method has the downside of an exponential blow-up in the size the expressions due to normalization.
- **Scan:** An exhaustive algorithm that scans and evaluates all BEs one by one for each assignment. While there are optimized techniques for evaluating one BE against assignments [15], since we have to evaluate multiple BEs, we use a straightforward implementation that iterates through (pre-parsed) operator trees corresponding to BEs, and evaluates each tree according to the semantics of Boolean operators.
- **Scan with conjunction match:** This algorithm improves over the basic Scan by just evaluating the BEs that have at least one conjunction returned by the index.

5.3 BE depth

In this section we study the impact of the BE tree depth in memory size and index evaluation. First, in Table 2, we show the memory consumption for the different algorithms when varying the depth. Both the Dewey and Interval algorithms use the same amount of memory for the index and the annotations database. This is due to the fact that we

Depth	Dewey and Interval	DNF	Scan with conj	Scan
1	9.5	55	373	364
2	18	501	854	836
3	35	NA	NA	NA
4	70	NA	NA	NA

Table 2: Memory usage by all algorithms for different BE depths (in MB).

used a 32 bit field to hold a Dewey ID, and a 16 bit field for each of the *begin*, *end* markers of the interval. As shown in Table 2 the size of these data structures grow linear with depth for Dewey and Interval algorithms. We could not evaluate the Scan and DNF algorithms for depths greater than 2 since their memory consumption was too extreme for these cases. The reason for the large memory consumption in the case of the Scan algorithms is that they need to maintain the Boolean tree in memory for evaluation. Scan with conjunction matching has to maintain the index in addition to the Boolean tree for each expression being evaluated. Table 2 indicates that the memory requirement for the DNF expansion algorithm grows exponentially with depth (due to BE normalization). To further investigate this point, in Figure 7 we plot how the average size of a DNF expansion of a BE grows with depth. (The results are averages of 20 randomly generated BEs of each size.)

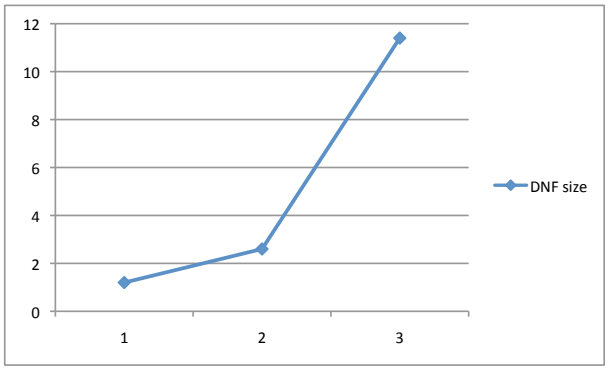

Figure 7: Average size of a single DNF expanded BE for depths 1, 2 and 3 (in KB).

In Figure 8 we show the running time for all algorithms for depths 1 and 2. These numbers are average latency numbers over 2,000 queries for each depth. For each depth d, queries were executed against an index with BE depth at most d. The selectivity for the different depths we tested is shown in Table 3. The index and the BE trees were loaded in memory prior to the evaluation (this is true for all the experiments that report query latency).

It is clear from Figure 8 that both Scan and Scan with conjunction matching behave orders of magnitude worse than the other algorithms. Figure 9 shows the same numbers omitting the results for the Scan algorithms. This figure shows that the running time for evaluating DNF-expanded trees also does not scale even for depth 2. Therefore, for the remainder of the experiments we focus on the Dewey and Interval algorithms. In the case the indexed expressions are small DNFs, as shown for depth 1 in the figure, the performance for DNF algorithm can be as good as the Dewey and Interval algorithms.

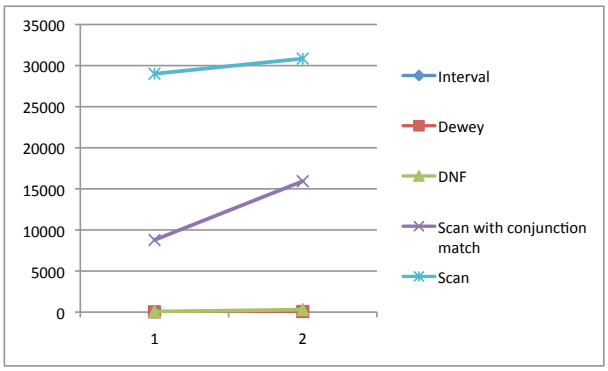

Figure 8: Running time for all algorithms for BE depths 1 and 2 (in ms).

Depth	Selectivity
1	22.85%
2	57.97%
3	10.97%
4	49.23%

Table 3: Average selectivity (over 2,000 queries) for indexes of different BE depths.

Figure 10 shows the total running time for Dewey and Interval for depths between 1 and 4. It also shows the time spent in index evaluation and sorting. The sorting time is the time spent to sort the conjunctions returned by the index either by Dewey or by the begin value of the Interval ID (both algorithms require this sorting step). In Figure 11 we present results only for the Expression Evaluation component, i.e., only the overhead incurred by the two algorithms over the index evaluation and sorting steps. In this figure it is more clear that Interval scales better with depth.

5.4 Number of BEs

In this section we study the impact of the size of the index. Table 4 shows the the memory consumption incurred by both Dewey and Interval for different number of indexed BEs. As in the case of depth, the index and annotations database data structures grow linearly with index size for Dewey and Interval.

Figure 12 shows the total running time for both Dewey and Interval for different index sizes. It also shows the amount of time spent in index evaluation and sorting. For each index size we kept the selectivity as 50% and we limited the depth of the indexed BEs to a maximum of 2. The reported numbers are average latency numbers over 2,000 queries for each index. The index sizes we used for these experiments are smaller than the ones we used for the depth in-

Number of BEs	Dewey and Interval
10K	1.1
20K	2.0
50K	4.7
100K	9.1
150K	14

Table 4: Memory usage by Dewey and Interval for different number of indexed BEs (in MB).

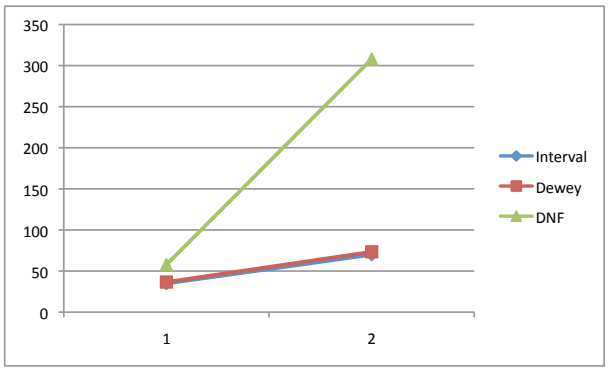

Figure 9: Running time for DNF expansion, Dewey, and Interval for BE depths 1 and 2 (in ms).

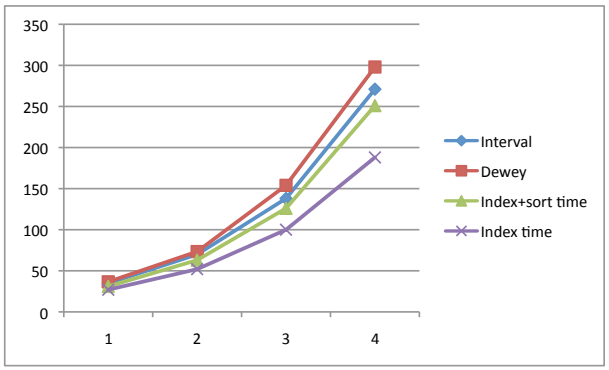

Figure 10: Total running time for Dewey and Interval evaluations for depths between 1 and 4 (in ms). The figure also shows the time spent in index evaluation and sorting.

vestigation, therefore the latency numbers are smaller than those reported in Figure 10. Figure 13 isolates the time spent in Expression Evaluation. Again the Interval algorithm scales better than Dewey w.r.t. the number of BEs indexed.

5.5 Selectivity

In this section we study the impact of BE selectivity. To adjust the selectivity we generated a large number of candidate BEs that did not match any of the queries in the test set. By adding those BEs to the full set, we can precisely adjust the selectivity in the dataset. Figure 14 shows the evaluation time, i.e., the time spent in Expression Evaluation, for different selectivity values. Both Dewey and Interval scale linearly with respect to selectivity. For these experiments we set the maximum BE depth to 3. The numbers shown are average latency numbers over 2,000 queries for each selectivity value.

6. RELATED WORK

One way to view our system is as a publish/subscribe system where a subscription is represented by a boolean expression and a stream of published events are modeled by a set of attribute-value pairs. The simplest, and slowest evaluation method is scanning all of the BEs and performing an evaluation of those expressions. There are ways to optimize those evaluations [15] but it is not scalable in practice for a large number of complex BEs. An improvement over scanning all expressions is the counting algorithm of [20, 21], where information retrieval (IR)-style inverted lists are used to find candidate conjunctions for DNF expressions. They use posting lists and count the number of words (conjuncts) in the incoming document that are present in the BEs using posting lists to determine matches efficiently. The counting algorithm was also enhanced by Carzaniga et al. [5] to do more efficient short-circuit evaluation. Extensions to the counting algorithm to handle arbitrary AND/OR trees (without explicit NOTs) have been proposed in [3].

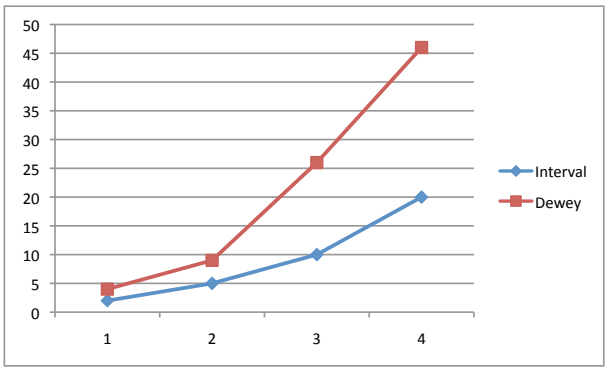

Figure 11: Running time for Dewey and Interval evaluations for depths between 1 and 4 (in ms), excluding the time spent in index evaluation.

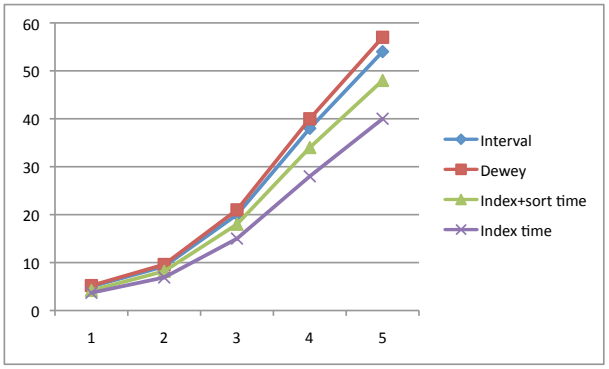

Figure 12: Total running time for Dewey and Interval evaluations for different number of indexed BEs (in ms). The figure also shows the time spent in index evaluation and sorting.

Le Subscribe [9, 14] is an algorithm for evaluating conjunctions in a publish/subscribe system. One variant of Le Subscribe uses a hash table to store clusters of conjunctions using their most selective conjunct as the hash table key. The most selective conjunct is thus used as a filter criteria for qualifying that conjunction for full evaluation. In [17] a method of using inverted lists to efficiently evaluate CNF and DNF expressions. The algorithm works by finding matching conjunctions efficiently using enhancements to the WAND algorithm [4]. These algorithms (except for simple scanning) are limited by their inability to handle general, complex BEs. Although that limitation may be partially mitigated by doing DNF expansion, for very complicated BEs, the explosion of conjunctions that results from DNF expansion may be prohibitively large.

Inverted lists have also been extended to handle the indexing of regular expressions expressions [6, 7], though the structure of that problem is different from ours.

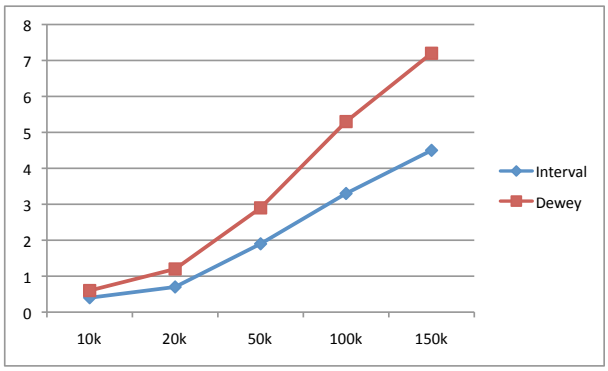

Figure 13: Running time for Dewey and Interval evaluations for different number of indexed BEs (in ms), excluding the time spent in index evaluation.

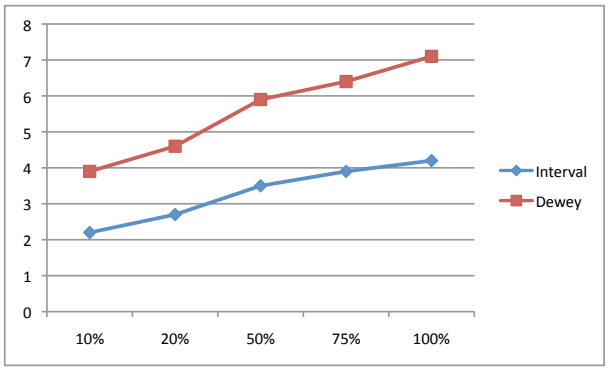

Figure 14: Running time for Dewey and Interval evaluations for different selectivity values (in ms), excluding the time spent in index evaluation.

Database systems have a myriad of techniques for evaluating boolean expressions. These techniques include scalable trigger systems [11], the EVALUATE operator [2, 19] (in Oracle), and table lookup methods [1]. All of these schemes can not match the performance scalability of a specialized BE matching algorithm and efficiently handle very complex boolean, high-dimensional expressions at the high throughput.

7. CONCLUSIONS

We have presented two solutions for the problem of evaluating complex boolean expressions. We show that both of these solutions substantially outperform the previously best known approaches which relied on first transforming the complex boolean expression into a CNF or DNF. The two algorithms are similar in spirit, both working bottom up and evaluating the tree given only the satisfied leaves. They differ, however, on the type and amount of information stored at each conjunction leaf. Of the two, the Interval matching approach is more advantageous – it is faster, uses less memory and is simpler to describe and implement.

As future work we want to examine the impact of common subexpression elimination across indexed BEs – i.e., to identify common AND/OR trees and to index and evaluate these trees only once. Common-subexpression identification is an interesting problems by itself, where techniques such as predicate ordering [12] may be helpful.

8. REFERENCES

[1] G. Ashayer, H. K. Y. Leung, and H.-A. Jacobsen. Predicate matching and subscription matching in publish/subscribe systems. In *ICDCSW '02*, pages 539–548, Washington, DC, USA, 2002. IEEE Computer Society.

[2] M. Ault, D. Liu, and M. Tumma. *Oracle Database 10g New Features: Oracle 10g Reference for Advanced Tuning and Administration*. Rampant TechPress, 2003.

[3] S. Bittner and A. Hinze. The arbitrary boolean publish/subscribe model: making the case. In *DEBS'07*, pages 226–237, Toronto, Ontario, Canada, 2007. ACM.

[4] A. Z. Broder, D. Carmel, M. Herscovici, A. Soffer, and J. Zien. Efficient query evaluation using a two-level retrieval process. In *CIKM '03*, pages 426–434, New York, NY, USA, 2003. ACM.

[5] A. Carzaniga and A. L. Wolf. Forwarding in a content-based network. In *SIGCOMM'03*, pages 163–174, Karlsruhe, Germany, 2003. ACM.

[6] C. Y. Chan, M. N. Garofalakis, and R. Rastogi. Re-tree: an efficient index structure for regular expressions. *VLDB Journal*, 12(2):102–119, 2003.

[7] J. Cho and S. Rajagopalan. A fast regular expression indexing engine. In *ICDE'02*, pages 419–430, San Jose, CA, USA, 2002. IEEE Computer Society.

[8] P. T. Eugster, P. A. Felber, R. Guerraoui, and A.-M. Kermarrec. The many faces of publish/subscribe. *ACM Computing Surveys*, 35(2):114–131, 2003.

[9] 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 *SIGMOD '01*, pages 115–126, New York, NY, USA, 2001. ACM.

[10] Google Advertising. http://www.google.com/ads/.

[11] E. N. Hanson, C. Carnes, L. Huang, M. Konyala, L. Noronha, S. Parthasarathy, J. B. Park, and A. Vernon. Scalable trigger processing. In *ICDE'99*, pages 266–275, Sydney, Austrialia, 1999. IEEE Computer Society.

[12] A. Kemper, G. Moerkotte, and M. Steinbrunn. Optimizing boolean expressions in object-bases. In *VLDB'92*, pages 79–90, San Francisco, CA, USA, 1992. Morgan Kaufmann Publishers Inc.

[13] Microsoft Advertising. http://advertising.microsoft.com/.

[14] J. Pereira, F. Fabret, F. Llirbat, and D. Shasha. Efficient matching for web-based publish/subscribe systems. In *CooplS '02*, pages 162–173, London, UK, 2000. Springer-Verlag.

[15] K. A. Ross. Selection conditions in main memory. *ACM TODS*, 29:132–161, 2004.

[16] 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 *SIGMOD'02*, pages 204–215, Madison, Wisconsin, 2002. ACM.

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

[18] Yahoo! Advertising. http://advertising.yahoo.com/.

[19] A. Yalamanchi, J. Srinivasan, and D. Gawlick. Managing expressions as data in relational database systems. In *CIDR*, Asilomar, CA, USA, 2003.

[20] T. Yan and H. Garcia-Molina. Index structures for selective dissemination of information under the boolean model. *ACM TODS*, 19(2):332–364, 1994.

[21] T. Yan and H. Garcia-Molina. The sift information dissemination system. *ACM TODS*, 24(4):529–565, 1999.