# Indexing Boolean Expressions

Steven Euijong Whang, Hector Garcia-Molina
Stanford University, Stanford, CA
{swhang,hector}@cs.stanford.edu

Chad Brower, Jayavel Shanmugasundaram, Sergei Vassilvitskii, Erik Vee, Ramana Yerneni
Yahoo! Research, Santa Clara, CA
{cbrower,jaishan,sergei,erikvee,yerneni}@yahoo-inc.com

## ABSTRACT

We consider the problem of efficiently indexing Disjunctive Normal Form (DNF) and Conjunctive Normal Form (CNF) Boolean expressions over a high-dimensional multi-valued attribute space. The goal is to rapidly find the set of Boolean expressions that evaluate to true for a given assignment of values to attributes. A solution to this problem has applications in online advertising (where a Boolean expression represents an advertiser's user targeting requirements, and an assignment of values to attributes represents the characteristics of a user visiting an online page) and in general any publish/subscribe system (where a Boolean expression represents a subscription, and an assignment of values to attributes represents an event). All existing solutions that we are aware of can only index a specialized sub-set of conjunctive and/or disjunctive expressions, and cannot efficiently handle general DNF and CNF expressions (including NOTs) over multi-valued attributes.

In this paper, we present a novel solution based on the inverted list data structure that enables us to index arbitrarily complex DNF and CNF Boolean expressions over multi-valued attributes. An interesting aspect of our solution is that, by virtue of leveraging inverted lists traditionally used for ranked information retrieval, we can efficiently return the top-N matching Boolean expressions. This capability enables emerging applications such as *ranked* publish/subscribe systems [16], where only the top subscriptions that match an event are desired. For example, in online advertising there is a limit on the number of advertisements that can be shown on a given page and only the "best" advertisements can be displayed. We have evaluated our proposed technique based on data from an online advertising application, and the results show a dramatic performance improvement over prior techniques.

## 1. INTRODUCTION

We consider the problem of efficiently indexing Disjunctive Normal Form (DNF) and Conjunctive Normal Form (CNF) Boolean expressions (BEs) over a high-dimensional, discrete but multi-valued attribute space. An example DNF expression is of the form:

$(A \in \{a_1, a_2\} \wedge B \notin \{b_1, b_2\} \wedge C \in \{c_1\}) \vee (A \in \{a_1, a_3\} \wedge D \notin \{d_1\})$

where $A$, $B$, $C$ and $D$ are attributes, and $a_1$, $a_2$, $a_3$, $b_1$, $b_2$, $c_1$ and $d_1$ are attribute values. Similarly, an example CNF expression is of the form:

$(A \in \{a_1, a_2\} \vee B \notin \{b_1, b_2\} \vee C \in \{c_1\}) \wedge (A \in \{a_1, a_3\} \vee D \notin \{d_1\})$

Given a large collection of such DNF or CNF expressions, our goal is to rapidly find the subset of expressions that are satisfied (i.e., evaluate to true) for a particular assignment of attributes to values. An example assignment of attributes to values is:

$A$=$a_1$, $B$=$b_1$, $C$=$c_2$, $D$=$d_1$

which would satisfy the CNF expression above, but not the DNF expression.

An efficient solution to the above problem has a wide range of applications in diverse domains. As one example, consider online display advertising [17, 14, 23], where advertisers target various user and web page attributes and register them as campaigns[1]. Each campaign specification can be modeled as a Boolean expression, e.g., (Property $\in$ {Finance} $\wedge$ AgeCategory $\in$ {4,5,6}) $\vee$ (Property $\in$ {Sports} $\wedge$ Source $\notin$ {ESPN} $\wedge$ UserInterestNFL $\in$ {True}) because advertisers only wish to show an advertisement to users who satisfy all their constraints (and enforcing $\notin$ is frequently as important, or more important, as $\in$ to advertisers in this context). When a user visits a certain web page, this assignment can be modeled as an assignment of values to attributes, e.g., Property = Finance and AgeCategory = 5 and Source = MSN (while only a few user attributes are shown here for illustration purposes, typically, the number of possible user attributes is of the order of many hundreds because of various custom user targeting categories like UserInterestNFL above). Given a user visit assignment, the goal is to rapidly find all matching campaigns for the user so that the right set of advertisements can be shown to the user. Hence, online display advertising can be mapped to the Boolean expression indexing problem, whereby the campaigns are indexed and probed for each user visit.

In general, our solution can be used in content-based publish/subscribe systems [1], which are designed to rapidly match incoming assignments to pre-registered subscriptions. Again, each subscription is typically modeled as a Boolean expression, e.g., (Category $\in$ {Sports} $\wedge$ Rating $\in$ {4,5}) $\vee$ (Category $\in$ {Finance} $\wedge$ Industry $\notin$ {Tech}), and each assignment specifies the values of attributes used in the subscriptions, e.g., Category = Sports and

[1]Note that display advertising is different from search advertising, where advertisers target keywords. Display advertising is also a multi-billion dollar industry.

Rating = 5 (note that in reality, there will be many tens or even hundreds of possible assignment attributes). Clearly, having an efficient index over Boolean expressions will enable publish/subscribe systems to rapidly process assignments in order to determine the matching subscriptions.

Further applications of our work include expert systems and pattern matching in the AI field [20] and compliance checking [2, 3]. Expert systems match if/then rules against input parameters to quickly make deductions or choices. In computer security and trust management systems, compliance checkers are used to determine whether the credentials of a user match certain access control policies.

While the Boolean expression indexing problem has many applications, unfortunately, existing indexing techniques are not capable of handling such expressions. For example, Le Subscribe [11], one of the most efficient publish/subscribe implementations, can only index conjunctive predicates with single valued predicates (e.g., A $\in \{a_1\}$, but not A $\in \{a_1, a_2\}$); more complex predicates such as multi-valued predicates (e.g., to represent age ranges as in the above example) and other operations such as $\notin$ are not indexed, but have to be post-processed. Another state-of-the-art publish/subscribe system [8] extends the Le Subscribe indexing technique to support ORs in limited contexts, but again does not support multi-valued predicates, NOTs, or CNF expressions. Other publish/subscribe systems based on multi-dimensional tree indices [16] also only support conjunctive predicates that correspond to convex indexable regions (without NOTs), and also do not scale to a large number of attributes due to the inherent limitations of high-dimensional indexing. Finally, many of the indices used for online search and content match advertising [21, 26, 5] only support fuzzy matches, and do not ensure the strict Boolean semantics of expressions.

In order to address the above limitations, in this paper, we propose a new technique for indexing Boolean expressions. The key idea is to use the inverted list data structure [26], commonly used for ranked information retrieval, to index Boolean expressions. The inverted list data structure has many advantages, including impressive scalability properties (commonly used for large-scale search and information retrieval), and also the ability to handle a very large number of dimensions (attributes or keywords), all of which are well-suited to the Boolean expression indexing problem. However, there are some non-trivial problems that need to be solved in order to adapt inverted lists for the Boolean expression indexing problem. First, inverted lists are typically used to index documents that are viewed as a "bag of words", without much internal structure, and certainly not the level of precise structure required for Boolean expressions. Second, current inverted list processing algorithms are tailored for score-based pruning of indexed documents, and not designed to enable precise Boolean expression evaluation on the *indexed objects* (Note that there are many inverted list query processing algorithms that support complex *query expressions* [7, 26], but we are interested in the inverse problem of complex *indexed expressions*).

One of the main technical contributions of this paper is a set of techniques for indexing Boolean expressions using inverted lists, which address the above challenges. First, we develop a technique for mapping complex Boolean expressions into "flat" entries in an inverted list, which can be then reconstructed efficiently at matching time. Second, we develop a query processing technique that essentially maps Boolean expression evaluation into a scored evaluation on an inverted list, thereby reusing the decades of research in speeding up inverted list scoring algorithms for the Boolean expression evaluation problem (note that we still guarantee the strict Boolean semantics of expressions).

Another key technical contribution of this paper — and a key benefit of using inverted lists — is an extension of the above algorithms to only return the top-N Boolean expression matches (based on a fairly general notion of scores for Boolean expressions). Such top-N matches enables an even larger class of applications such as *ranked* publish/subscribe [16], whereby only the top few subscriptions that match an assignment are returned. This feature also better models applications such as online display advertising, where only a small number of advertisements (Boolean expressions) can be shown on a user page due to limited real-estate (in this case, the score of an advertisement could reflect the relevance of the advertisement to the user's profile). Note, however, that the focus of this paper is on *efficiently* retrieving the top-N BEs given a scoring method, and not on developing different scoring methods or evaluating their effectiveness.

We have implemented the proposed algorithms and compared them with other state-of-the-art techniques such as Le Subscribe [11] and SIFT [25], using data from an online advertising application. Our implementation has been designed for high-throughput and low-latency applications, and consequently, the index data structures are main-memory resident and optimized for reads; updates are maintained in a small tail data structure, which is periodically (offline) merged and then swapped with the main read-optimized data structure (please see Section 6 for more details). Our performance results indicate that the proposed approaches are competitive with existing approaches for simple Boolean expressions over a small number of attributes, but are dramatically better for the cases involving complex expressions, large dimensions, and top-N results.

In summary, the main contributions of this paper are:

- A technique for indexing and evaluating DNF expressions using inverted lists (Section 3)
- A technique for indexing and evaluating CNF expressions using inverted lists (Section 4)
- Incorporating scoring and top-N pruning during DNF and CNF expression evaluation (Section 5)
- A discussion of how standard techniques for inverted list scaling and updates can be applied to the BE indexing problem (Section 6)
- Experimental evaluation of the proposed techniques based on synthetic data from an online advertisement application (Section 7)

## 2. MODEL

### 2.1 Syntax

A Boolean expression (BE) uses two types of primitives: $\in$ and $\notin$ predicates. For example, the predicate $state \in \{CA, NY\}$ 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.

A BE is either a DNF (i.e., disjunctive normal form) or CNF (i.e., conjunctive normal form) expression of the basic $\in$ and $\notin$ predicates. For example, a subscription in DNF form could be $(age \in \{2, 3\} \wedge state \in \{CA\}) \vee (age \in \{2, 3, 4\} \wedge state \in \{NY\})$ while a BE in CNF form could be $age \in \{2, 3\} \wedge (state \in \{CA\} \vee gender \in \{F\})$. While the same attribute may occur in different disjuncts/conjuncts, an attribute does not appear in the same disjunct/conjunct more than once. Notice that we can easily express simple conjunctions using either of our two BE formats. We believe that using DNF/CNF expressions with $\in$ and $\notin$ predicates is a simple but rich way to specify conditions. For example, our DNF expressions with $\in$ predicates are more expressive than DNF expressions with $=$ predicates by being able to add an additional disjunction layer within a conjunction.

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 $\{gender = F, state = CA\}$. 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.

An attribute name and value pair $(A, v)$ is referred to as a *key*, and we will use both terms – pairs and keys – interchangeably throughout the paper.

## 2.2 BE Matching

A BE is satisfied by an assignment if its logical expression is satisfied. However, in the case where an assignment does not specify all possible attributes, there may be ambiguities in evaluating a BE. For example, suppose there is a BE $age \in \{2\} \wedge state \notin \{NY\}$ and an assignment $\{age$=2$\}$. Depending on the semantics for $\notin$, we can either consider the BE to be satisfied (assuming the assignment contains some default value for the unspecified state) or not satisfied (because state may in fact be NY). We thus define two types of $\notin$ predicates:

- A *WEAK-$\notin$* predicate is satisfied if an assignment does not specify a value for the attribute of the predicate. The attribute is assumed to have a default value that is different from the values in the *WEAK-$\notin$* list.
- A *STRONG-$\notin$* predicate is violated if an assignment does not specify a value for the attribute of the predicate. Hence, the predicate requires a non-NULL attribute value of the assignment to be satisfied.

Using the two types of $\notin$ predicates, we can exactly express the semantics of the BEs. Our algorithms support both types of predicates. The *WEAK-$\notin$* predicates are naturally supported while for *STRONG-$\notin$* predicates, we require default values to be specified in the assignments (this condition can be satisfied by assigning default values for the attributes involved in *STRONG-$\notin$* predicates for a given assignment). Strong $\notin$ is rewritten to a weak $\notin$ by using default (NULL) values. For example, X strong $\notin$ a,b is rewritten to X weak $\notin$ a,b,NULL. Also, if an attribute value X is unspecified in an assignment, we add X = NULL to the assignment.

## 2.3 Supporting Ranges

Many applications specify ranges of values in their BEs (e.g., $age < 30$). A simple way to use a range predicate in our model is to convert the range into $\in$ and $\notin$ predicates by enumerating all possible values or ranges of values. For example, if the granularity for ages is 10 years, the range $age < 30$ can exhaustively be enumerated into $age \in \{0, 1, 2\}$.

For longer ranges such as ranges of dates, we can combine different units of ranges based on a hierarchy. First, a basic $\in$ or $\notin$ BE predicate can divide a long range into non-overlapping smaller ranges of possibly different units. For example, the date range $2006/12 \leq date \leq 2009/01$ can be expressed as $date \in \{$2006/12, 2007(all), 2008(all), 2009/01$\}$ given that the hierarchy contains two units of ranges: year and month. Second, an attribute of an assignment still has a single value, but is internally expanded to include all the values in the path from the leaf (which represents the attribute value) to the root of the hierarchy tree. For example, an assignment $\{date = 2008/2\}$ is expanded to $\{date = 2008/2, date = 2008\}$ based on the hierarchy. (Notice that $date$ = 2008 was automatically generated from $date$ = 2008/2.) The assignment then matches the BE because $date$ = 2008 satisfies $date \in \{$2006/12, 2007(all), 2008(all), 2009/01$\}$. A full discussion of constructing the "best" hierarchy that minimizes the sizes of BEs and assignments is of separate interest, but not covered in this paper.

The advantage of converting range predicates into $\in$ and $\notin$ predicates is that we can efficiently process BEs as we shall see in our inverted indexing algorithms.

## 2.4 Scoring

The score of a BE $E$ reflects the "relevance" of $E$ to an assignment $S$. For example, a user interested in running might be more interested in an advertisement on shoes than an advertisement on flowers. If $E$ is a conjunction of $\in$ and $\notin$ predicates, the score of $E$ is defined as

$$Score_{conj}(E, S) = \sum_{(A,v) \in IN(E) \cap S} w_E(A, v) \times w_S(A, v)$$

where $IN(E)$ is the set of all attribute name and value pairs in the $\in$ predicates of $E$ (we ignore scoring $\notin$ predicates, which is a common approach in IR systems [7]) and $w_E(A, v)$ is the weight of the pair $(A, v)$ in $E$. Similarly, $w_S(A, v)$ is the weight for $(A, v)$ in $S$. For example, a BE $age \in \{1, 2\} \wedge state \in \{CA\}$ could be targeting young people, giving the pair $(age, 1)$ a high weight of 10 while giving $(age, 2)$ a lower weight of 5 and $(state, CA)$ a weight of 3. If there is an assignment $\{age$=1, $state$=$CA\}$, where the first pair has a weight of 1 while the second pair 2, the score of the BE to the assignment is $10\times1+3\times2 = 16$. Hence, our scoring for conjunctions is similar to vector space scoring in IR systems.

In order to do top-N pruning, we also generate an upper bound $UB(A, v)$ for each attribute name and value pair $(A, v)$ such that

$$UB(A, v) \geq \max(w_{E_1}(A, v), w_{E_2}(A, v), \ldots)$$

For instance, if $UB(age, 1) = 10$, then $(age, 1)$ may not contribute more than a weight of 10 regardless of the BE.

*DNF Scoring.* The score of a DNF BE $E$ is defined as the maximum of the scores of the conjunctions within $E$ where $E.i$ denotes the ith conjunction of $E$ and $|E|$ the number of conjunctions in $E$.

$$Score_{DNF}(E, S) = \max_{i=1..|E|} Score_{conj}(E.i, S)$$

While the score of $E$ could be defined in other ways (e.g., the sum of the scores of the conjunctions), we believe it is more intuitive to view a DNF BE as a collection of possible conjunctions where only one can represent the entire BE at a time.

*CNF Scoring.* The score of a CNF BE $E$ is similar to $Score_{conj}$ and is defined as the sum of the disjunction scores (using $Score_{DNF}$) within $E$ where $E.i$ denotes the ith disjunction of $E$ and $|E|$ the number of disjunctions in $E$.

$$Score_{CNF}(E, S) = \Sigma_{i=1..|E|} Score_{DNF}(E.i, S)$$

Intuitively, the CNF score combines all the contributions of each disjunction.

## 3. DNF ALGORITHM

Before presenting our DNF algorithm, it is important to understand why simple inverted list joining techniques [21] commonly used in IR systems do not return the matching BEs properly. Suppose that we have a set of two BEs: $E_1$: $A \in \{a\}$ and $E_2$: $A \in \{a\} \wedge B \in \{b\} \wedge C \in \{c\}$, and we create an inverted list with the posting list for $a$ containing both $E_1$ and $E_2$, and the posting list for $b$ (and $c$) containing only $E_2$. Now suppose that we are given an assignment $A = a \wedge B = b$. If we simply intersect the posting lists corresponding to $a$ and $b$, we would get the empty result set that would miss $E_1$ as a solution. If we simply union the posting lists corresponding to $a$ and $b$, we would get both $E_1$ and $E_2$ as results, which would be incorrect because $E_2$ is not satisfied by the assignment.

To address the above issues, the DNF algorithm first splits each DNF BE into conjunctions of $\in$ or $\notin$ predicates. For example, the BE $E = (age \in \{3\} \wedge state \notin \{CA\}) \vee (age \in \{3\} \wedge gender \in \{F\})$ is split into two conjunctions $c_1 = age \in \{3\} \wedge state \notin \{CA\}$ and $c_2 = age \in \{3\} \wedge gender \in \{F\}$. We define the size of a conjunction to be the number of $\in$ predicates (ignoring the $\notin$ predicates). For example, $c_1$ above has a size of 1 while $c_2$ has a size 2. We internally keep the DNF ID within each conjunction so that we can later on return the IDs of the satisfied DNFs based on the satisfied conjunctions.

*Inverted List Construction.* We now build an inverted list data structure on the conjunctions of the BEs. We first partition all the conjunctions by their sizes. We refer to the partition with conjunctions of size $K$ as the $K$-index. For each $K$-index, we create posting lists for all possible attribute name and value pairs (also called *keys*) among the conjunctions. A posting list head contains the key $(A, v)$. (The number next to each $(A, v)$ will be used in Section 5 for ranking and can be ignored for now). The keys of the posting lists are stored in a hash table, which will be used to search posting lists given keys of an assignment. Each entry of a posting list represents a conjunction $c$ and contains the ID of $c$ and a bit indicating whether the key $(A, v)$ is involved in an $\in$ or $\notin$ predicate in $c$ (ignore the third value for now). A posting list entry $e_1$ is "smaller" than another entry $e_2$ if the conjunction ID of $e_1$ is smaller than that of $e_2$. In the case where both conjunction IDs are the same (in which case $e_1$ and $e_2$ appear in different lists), $e_1$ is smaller than $e_2$ only if $e_1$ contains a $\notin$ while $e_2$ contains an $\in$. Otherwise, the two entries are considered the same. Using this ordering, the entries in a posting list are sorted in increasing entry order, while in each $K$-index, the posting lists themselves are sorted in increasing entry order of their first entry. (Notice that we never have two entries with the same conjunction ID within the same posting list because an attribute is only allowed to occur once in each conjunction.) Keeping the posting lists sorted in each $K$-index reduces the sorting time of posting lists in the Conjunction Algorithm (shown below).

As a special case, conjunctions of size 0 (e.g., $age \notin \{3\}$ is a conjunction of size 0 because it has no $\in$ predicates) are all included in a single posting list called $Z$. This special posting list is needed to ensure that zero-sized conjunctions appear in at least one posting list given an assignment. In addition, each entry in $Z$ contains an $\in$ predicate. This modification ensures that the Conjunction Algorithm also works for zero-sized conjunctions.

The total number of entries in the inverted list is proportional to $|C| \times P_{avg}$ where $|C|$ is the number of conjunctions and $P_{avg}$ is the average number of predicates per conjunction. Given that each entry is a few bytes large, the entire index can easily fit in memory for millions of BEs.

*Example.* Consider the six conjunctions in Figure 1. The third column (called $K$) shows the sizes of the conjunctions ($c_1, c_2, c_3, c_4$ have a size of 2, $c_5$ has a size 1, and $c_6$ has a size 0). The conjunctions are first partitioned according to their sizes. For each partition $K = 0, 1, 2$, we construct the $K$-indexes as shown in Figure 2. For instance, the key $(age, 4)$ has a posting list inside the partition $K = 1$ and contains an entry representing $c_5$. Notice that the weight for any entry that has a $\notin$ is 0 because we do not consider $\notin$ predicates for scoring.

**Figure 1: A set of conjunctions**

| ID | Expression | K |
|---|---|---|
| $c_1$ | $age \in \{3\} \wedge state \in \{NY\}$ | 2 |
| $c_2$ | $age \in \{3\} \wedge gender \in \{F\}$ | 2 |
| $c_3$ | $age \in \{3\} \wedge gender \in \{M\} \wedge state \notin \{CA\}$ | 2 |
| $c_4$ | $state \in \{CA\} \wedge gender \in \{M\}$ | 2 |
| $c_5$ | $age \in \{3, 4\}$ | 1 |
| $c_6$ | $state \notin \{CA, NY\}$ | 0 |

**Figure 2: Inverted list for Figure 1**

| K | Key & UB | Posting List |
|---|---|---|
| 0 | $(state, CA), 2.0$ | $(6, \notin, 0)$ |
| | $(state, NY), 5.0$ | $(6, \notin, 0)$ |
| | $Z, 0$ | $(6, \in, 0)$ |
| 1 | $(age, 3), 1.0$ | $(5, \in, 0.1)$ |
| | $(age, 4), 3.0$ | $(5, \in, 0.5)$ |
| 2 | $(state, NY), 5.0$ | $(1, \in, 4.0)$ |
| | $(age, 3), 1.0$ | $(1, \in, 0.1)$ $(2, \in, 0.1)$ $(3, \in, 0.2)$ |
| | $(gender, F), 2.0$ | $(2, \in, 0.3)$ |
| | $(state, CA), 2.0$ | $(3, \notin, 0)$ $(4, \in, 1.5)$ |
| | $(gender, M), 1.0$ | $(3, \in, 0.5)$ $(4, \in, 0.9)$ |

*Conjunction Algorithm.* We now describe how the Conjunction Algorithm (Algorithm 1) returns all the satisfying conjunctions given an assignment. Two observations help us efficiently find a conjunction $c$ that matches an assignment $S$ with $t$ keys:

1. For a $K$-index ($K \leq t$), a conjunction $c$ (with $K$ terms) matches $S$ only if there are exactly $K$ posting lists where each list is for a key $(A, v)$ in $S$ and the ID of $c$ is in the list with an $\in$ annotation.
2. For no $(A, v)$ keys in $S$ should there be a posting list where $c$ occurs with a $\notin$ annotation.

For example, the conjunction $c = age \in \{3\} \wedge state \in \{CA\}$ matches the assignment $S = \{age = 3, state = CA, gender = M\}$ because there are exactly two posting lists (for the keys $(age, 3)$ and $(state, CA)$) that contain the id of $c$ is their lists with an $\in$ annotation. On the other hand, if $c = age \in \{3\} \wedge state \in \{CA\} \wedge gender \notin \{M\}$, then although there are two posting lists that contain $c$ with an $\in$ annotation, $c$ does not match $S$ because there is also a posting list for $(gender, M)$ that contains the id of $c$ with an $\notin$ annotation.

Algorithm 1 iterates through the $K$-indexes in the inverted list (Step 4) and adds the satisfied conjunction IDs into $O$. Note that we do not need to look at $K$-indexes with $K > t$. Conjunctions in those indexes have more terms than what can be satisfied by $S$. For each conjunction size $K$, the GetPostingLists($S$,$K$) method is used to extract the posting lists that match $A$ (Step 6). PLists is thus a list of posting lists. In the case where $K$=0, GetPostingLists($S$,$K$) returns the $Z$ posting list in addition to the other posting lists matching $A$. Each posting list has a "current entry" (denoted as CurrEntry) that is initialized to the first entry in the list (Step 7). If $K$=0, we set $K$=1 (Step 9) once the posting lists are extracted because the processing of the posting lists for $K$=0 is identical to that of $K$=1. In Step 11, we include an optimization where we skip processing the conjunction size $K$ if the number of posting lists is smaller than $K$ because no conjunction can be satisfied.

From Step 13, Algorithm 1 starts skipping posting lists for conjunctions that are guaranteed not to match the assignment. This skipping is an extension and adaptation of the WAND algorithm [6] – which is an IR algorithm for top-N document ranking – for the purpose of evaluating and skipping complex expressions (see Section 8 for a detailed comparison). The SortByCurrentEntries(PLists) method first sorts the list of matching posting lists by their current entries. At this point, consider the first entry in the first list

**Algorithm 1** The Conjunction algorithm

```
1:  input: inverted list idx and assignment $S$
2:  output: set of IDs $O$ matching $S$
3:  $O \leftarrow \emptyset$
4:  for $K$=min(idx.MaxConjunctionSize, $|S|$)...0 do
5:      /* List of posting lists matching A for conjunction size $K$ */
6:      PLists ← idx.GetPostingLists($S$,$K$)
7:      InitializeCurrentEntries(PLists)
8:      /* Processing $K$=0 and $K$=1 are identical */
9:      if $K$=0 then $K \leftarrow 1$
10:     /* Too few posting lists for any conjunction to be satisfied */
11:     if PLists.size() $< K$ then
12:         continue to next for loop iteration
13:     while PLists[$K$-1].CurrEntry $\neq$ EOL do
14:         SortByCurrentEntries(PLists)
15:         /* Check if the first K posting lists have the same conjunction ID in their current entries */
16:         if PLists[0].CurrEntry.ID = PLists[$K$-1].CurrEntry.ID then
17:             /* Reject conjunction if a $\notin$ predicate is violated */
18:             if PLists[0].CurrEntry.AnnotatedBy($\notin$) then
19:                 RejectID ← PLists[0].CurrEntry.ID
20:                 for L = $K$ .. (PLists.size()-1) do
21:                     if PLists[L].CurrEntry.ID = RejectID then
22:                         /* Skip to smallest ID where ID > RejectID */
23:                         PLists[L].SkipTo(RejectID+1)
24:                     else
25:                         break out of for loop
26:                 continue to next while loop iteration
27:             else /*conjunction is fully satisfied */
28:                 O ← O ∪ {PLists[$K$-1].CurrEntry.ID}
29:             /* NextID is the smallest possible ID after current ID*/
30:             NextID ← PLists[$K$-1].CurrEntry.ID + 1
31:         else
32:             /* Skip first $K$-1 posting lists */
33:             NextID ← PLists[$K$-1].CurrEntry.ID
34:         for L = 0...$K$-1 do
35:             /* Skip to smallest ID such that ID $\geq$ NextID */
36:             PLists[L].SkipTo(NextID)
37: return O
```

(PLists[0].CurrEntry). Say this entry has an $\in$ annotation and is for conjunction $c$. The only way $c$ can match $S$ is if for lists PLists[0] through PLists[$K$-1], $c$ happens to be the first entry too. Because the way the lists are sorted, we can check this condition by only checking the last list (Step 16). Say the condition is not satisfied because PLists[$K$-1].CurrEntry.ID is $d(> c)$. Note that in this case, we do not need to consider conjunctions $c$,$c$+1,...$d$-1: they do not have the necessary $K$ lists. Thus, we can "skip" ahead to consider conjunction $d$, as done in lines 33-36. (The SkipTo(NextID) method advances the current entry of a posting list until the conjunction ID of the current entry is larger or equal to NextID.) Although the skipping seems minor in the simple example below, the effect becomes significant for a large number of conjunctions.

If PLists[0] and PLists[$K$-1] have the same conjunction ID in their current entries, we check whether any $\notin$ predicate of the conjunction was violated by looking at the current entry of the first posting list. (Notice that our sorting condition for entries guarantees that we can tell whether a $\notin$ predicate of a conjunction has been violated by only checking the first posting list.) If the conjunction is violated, we skip all the posting lists with the violated ID in their current entries to their next entries (Steps 23 and 36). If the conjunction is not violated, we conclude that the conjunction is satisfied and add the ID of the conjunction into $O$. The algorithm terminates when the $K$th posting list is empty (i.e., the current entry points to the end of the posting list).

*Example.* Given the assignment $S : \{age = 3, state = CA, gender = M\}$, the matching posting lists for $S$ from the inverted list of Figure 2 are shown in Figure 3. Notice that all the weights are omitted for convenience because we do not score BEs yet.

**Figure 3: Posting lists for assignment $S$**

| K | Key | Posting List |
|---|---|---|
| 0 | $(state, CA)$ | $(6, \notin)$ |
|  | $Z$ | $(6, \in)$ |
| 1 | $(age, 3)$ | $(5, \in)$ |
| 2 | $(age, 3)$ | $(1, \in)$ $(2, \in)$ $(3, \in)$ |
|  | $(state, CA)$ | $(3, \notin)$ $(4, \in)$ |
|  | $(gender, M)$ | $(3, \in)$ $(4, \in)$ |

We start by retrieving the posting lists for $K$=2 at Step 6 as shown in Figure 4. The current entry for each posting list is underlined. PLists now contains three posting lists.

**Figure 4: Posting lists for $K$=2**

| Key | Posting List |
|---|---|
| $(age, 3)$ | $\underline{(1, \in)}$ $(2, \in)$ $(3, \in)$ |
| $(state, CA)$ | $\underline{(3, \notin)}$ $(4, \in)$ |
| $(gender, M)$ | $\underline{(3, \in)}$ $(4, \in)$ |

At Step 16, we observe that the 1st and 2nd posting lists do not have the same conjunction ID in their current entries (i.e., PLists[0]. CurrEntry.ID = 1 while PLists[1].CurrEntry.ID = 3). Hence, conjunctions $c_1$ and $c_2$ do not match and we skip the 1st posting list to conjunction ID 3 (Steps 34∼36). After sorting the posting lists again at Step 14, the resulting posting lists are shown in Figure 5. The current entry of the posting list of $(age, 3)$ is now $(3, \in)$. Notice that after the sorting, the first and second posting lists have changed positions within PLists. Now the first two posting lists have the same conjunction ID in their current entries (Step 16) and we continue to check if any $\notin$ predicates of $c_3$ are violated. At Step 18, we observe that the current entry of the first posting list $(state, CA)$ indeed contains a $\notin$ so we know that $c_3$ does not match the assignment.

**Figure 5: Posting lists for $K$=2 after first skipping**

| Key | Posting List |
|---|---|
| $(state, CA)$ | $\underline{(3, \notin)}$ $(4, \in)$ |
| $(age, 3)$ | $(1, \in)$ $(2, \in)$ $\underline{(3, \in)}$ |
| $(gender, M)$ | $\underline{(3, \in)}$ $(4, \in)$ |

We then skip all the posting lists containing conjunction ID=3 in their current entries to their next entries. After sorting the posting lists again, we arrive at the posting lists of Figure 6. The current entry of posting list $(age, 3)$ now points to the end of the list (denoted as EOL) while the current entry for the first two posting lists is $(4, \in)$. This time, conjunction $c_4$ is satisfied because two $\in$ predicates are satisfied while no $\notin$ predicate is violated. As a result, we add conjunction ID=4 into $O$ at Step 28. After skipping the first two posting lists to their next entries, we arrive at the end of list for all posting lists and thus exit the first for loop of Step 4.

Next, we retrieve the posting list $(age, 3)$ for $K$=1. The conjunction $c_5$ is satisfied because one $\in$ predicate is satisfied (Step 16) while no $\notin$ predicate is violated (Step 18). Hence, at Step 28, we add 5 into $O$.

Finally, we retrieve the two posting lists for $K$=0. However, we know that $c_6$ is violated because the entry of the first posting list

**Figure 6: Posting lists for $K$=2 after second skipping**

| Key | Posting List |
|---|---|
| $(state, CA)$ | $(3, \notin)$ $(4, \in)$ |
| $(gender, M)$ | $(3, \in)$ $\overline{(4, \in)}$ |
| $(age, 3)$ | $(1, \in)$ $\overline{(2, \in)}$ $(3, \in)$ `EOL` |

contains a $\notin$. After exiting the for loop at Step 4, Algorithm 1 terminates by returning $O$={4,5}.

*Complexity.* The number of posting lists to process is bounded by the size of the assignment $S$ (i.e., the number of attribute name and value pairs). In the worst case, we need to sort PLists after advancing one posting list by one entry in Step 36. The sorting in Step 14 takes $O(log(|S|))$ because the inverted list is initially sorted, and we only need to bubble down one posting list in PLists using a heap for each posting list skipped. Since the total number of entries is bounded by the total number of $\in$ and $\notin$ predicates of all conjunctions, the complexity becomes $O(log(|S|) \times |C| \times P_{avg})$ where $|C|$ is the number of conjunctions and $P_{avg}$ is the average number of predicates per conjunction. Notice that this complexity is higher than that of a linear scan of BEs, $O(|C| \times P_{avg})$. In practice, however, the Conjunction algorithm is efficient because $|S|$ is small (i.e., only a few posting lists containing a fraction of all possible entries are processed), and skipping the posting lists saves a significant amount of time.

## 4. CNF ALGORITHM

The CNF algorithm returns the IDs of the CNF BEs that are satisfied by an assignment. Again, we do not consider scoring for now (which is done in Section 5). The intuition of the CNF algorithm is to extend the Conjunction algorithm to run on the outer conjunctions of the CNF BEs. Although the CNF algorithm is similar to the DNF algorithm in building an inverted index and processing posting lists, the key differences are as follows:

1. The CNF BEs are not split into disjunctions, but are processed as their full expressions.
2. The size of a CNF BE is defined as the number of disjunctions in the BE with no $\notin$ predicates. For example, the CNF $c = (A \in \{1\} \vee B \in \{2\}) \wedge (C \notin \{3\} \vee D \in \{4\})$ has a size 1 because the second disjunction contains a $\notin$ predicate. Notice that this definition is a natural extension from the size of a conjunction.
3. We now keep a "disjunction ID" in each posting list entry. For example, $c$ from above has two disjunction IDs 0 and 1, and an entry for $(A, 1)$ would contain disjunction ID 0 in addition to the CNF ID, $\in$ or $\notin$ annotation, and the weight. The purpose is to make sure all the disjunctions in the CNF are satisfied. As an exception, the posting list entries in the zero posting list $Z$ have a disjunction ID of -1.
4. We maintain the number of $\notin$ predicates in each disjunction of a CNF in a structure separate from the inverted list data structure. For example, we save the array [0,1] for $c$ above because the first disjunction contains 0 $\notin$ predicates while the second disjunction contains 1 $\notin$ predicate. The $\notin$ predicate counts are used to keep track of the $\notin$ predicates of a disjunction that are satisfied, and can be set while the CNFs are loaded into the inverted list

*Inverted List Construction.* In comparison to the inverted list for conjunctions, a posting list entry for key $(A, v)$ now contains the ID of the disjunction containing the predicate of $(A, v)$. As a result, there may be multiple entries for one CNF in the same posting list with different disjunction IDs. Since the CNF algorithm below requires each posting list to contain at most one entry per CNF (to prevent "false negatives" where a matching CNF is mistakenly rejected having too few posting lists), we store entries with the same CNF ID in different posting lists with the same key. (In the case where there are duplicate entries for more than one CNF, we create posting lists with the same key until any posting list has at most one entry per CNF, and assign entries to the first posting list available in a greedy fashion.)

*Example.* Consider the six CNF BEs in Figure 7. The CNFs are first partitioned according to their sizes ($c_1$ through $c_4$ have a size 2, $c_5$ has a size 1, and $c_6$ has a size 0). For each partition $K$=0,1,2, we construct the $K$-indexes as shown in Figure 8. Each posting list entry now contains its disjunction ID as its 3rd value (the 4th value contains the weight, which will not be used in this example). For example, the only entry in the $(A, 2)$ posting list indicates that the predicate for $(A, 2)$ is in the first disjunction of $c_4$. Also notice that for $c_4$, the key $(A, 1)$ appears in both of its disjunctions. Hence, the posting list $(A, 1)$ is duplicated where the first list contains entry $(4, \in, 0, 0.1)$ while the second list contains $(4, \in, 1, 0.1)$. For the other entries of $(A, 1)$ we simply add them to the first posting list of $(A, 1)$ in a greedy fashion.

**Figure 7: A set of CNF expressions**

| ID | Expression |
|---|---|
| $c_1$ | $(A \in \{1\} \vee B \in \{1\}) \wedge (C \in \{1\} \vee D \in \{1\})$ |
| $c_2$ | $(A \in \{1\} \vee C \in \{2\}) \wedge (B \in \{1\} \vee D \in \{1\})$ |
| $c_3$ | $(A \in \{1\} \vee B \in \{1\}) \wedge (C \in \{2\} \vee D \in \{1\})$ |
| $c_4$ | $(A \in \{1\} \vee B \in \{1\}) \wedge (A \in \{1, 2\} \vee D \in \{1\})$ |
| $c_5$ | $(A \in \{1\} \vee B \in \{1\}) \wedge (C \notin \{1, 2\} \vee D \notin \{1\} \vee E \in \{1\})$ |
| $c_6$ | $A \notin \{1\} \vee B \in \{1\}$ |

**Figure 8: Inverted list for Figure 7**

| K | Key & UB | Posting List |
|---|---|---|
| 0 | $(A, 1), 0.5$ | $(6, \notin, 0, 0)$ |
| | $(B, 1), 1.5$ | $(6, \in, 0, 0.1)$ |
| | $Z, 0$ | $(6, \in, -1, 0)$ |
| 1 | $(C, 1), 2.5$ | $(5, \notin, 1, 0)$ |
| | $(C, 2), 3.0$ | $(5, \notin, 1, 0)$ |
| | $(D, 1), 3.5$ | $(5, \notin, 1, 0)$ |
| | $(A, 1), 0.5$ | $(5, \in, 0, 0.1)$ |
| | $(B, 1), 1.5$ | $(5, \in, 0, 0.7)$ |
| | $(E, 1), 4.5$ | $(5, \in, 1, 3.9)$ |
| 2 | $(A, 1), 0.5$ | $(1, \in, 0, 0.1)(2, \in, 0, 0.3)(3, \in, 0, 0.3)(4, \in, 0, 0.1)$ |
| | $(B, 1), 1.5$ | $(1, \in, 0, 0.3)(2, \in, 1, 0.5)(3, \in, 0, 0.3)(4, \in, 0, 0.5)$ |
| | $(C, 1), 2.5$ | $(1, \in, 1, 0.2)$ |
| | $(D, 1), 3.5$ | $(1, \in, 1, 2.1)(2, \in, 1, 2.5)(3, \in, 1, 1.7)(4, \in, 1, 1.9)$ |
| | $(C, 2), 3.0$ | $(2, \in, 0, 2.5)(3, \in, 1, 2.7)$ |
| | $(A, 1), 0.5$ | $(4, \in, 1, 0.1)$ |
| | $(A, 2), 1.0$ | $(4, \in, 0, 0.1)$ |

*CNF Algorithm.* We describe how the CNF algorithm (Algorithm 2) returns all the satisfying CNF BEs given an assignment. The following observation helps us efficiently find a CNF $c$ that matches an assignment $S$: For a $K$-index, a necessary (but not sufficient) condition for CNF $c$ (with $K$ disjunctions without $\notin$ predicates) to match $S$ is that there are at least $K$ posting lists where each list is for a key $(A, v)$ in $S$ and the ID of $c$ is in the list. For conjunctions, the analogous property was necessary and sufficient and required *exactly* $K$ lists. In the CNF case, a key may now appear in several disjunctions of a CNF and satisfy the expression. The new condition requires two changes: First, we modify Step 4 of Algorithm 1 to consider all possible $K$-indexes regardless of $|S|$. Specifically, we change the line "*for $K$=min(idx.MaxConjunction Size, $|S|$)...0 do*" to "*for $K$=idx.MaxConjunctionSize...0 do*." Second, once we find a CNF with $K$ matching lists, we must perform additional checks, as detailed below.

**Algorithm 2** The CNF algorithm

```
1: [Steps 1∼15 of Algorithm 1 except for Step 4]
2: if PLists[0].CurrEntry.ID = PLists[$K$-1].CurrEntry.ID then
3:    /* For each disjunction in the current CNF, one counter is
      initialized to the negative number of $\notin$ predicates */
4:    Counters.Initialize(PLists[0].CurrEntry.ID)
5:    for L = 0...(PLists.size()-1) do
6:       if PLists[L].CurrEntry.ID = PLists[0].CurrEntry.ID then
7:          /* Ignore entries in the $Z$ posting list */
8:          if PLists[L].CurrEntry.DisjID = -1 then
9:             continue to next for loop
10:         if PLists[L].CurrEntry.AnnotatedBy($\notin$) then
11:            Counters[PLists[L].CurrEntry.DisjID]++
12:         else /*Disjunction is satisfied */
13:            Counters[PLists[L].CurrEntry.DisjID] $\leftarrow$ 1
14:      else
15:         break
16:   Satisfied $\leftarrow$ true
17:   for L = 0...Counters.size()-1 do
18:      /* No $\in$ or $\notin$ predicates were satisfied */
19:      if Counters[L] = 0 then
20:         Satisfied $\leftarrow$ false
21:   if Satisfied = true then
22:      O $\leftarrow$ O $\cup$ {PLists[$K$-1].CurrEntry.ID}
23:   [Steps 29∼37 of Algorithm 1]
```

Algorithm 2 is similar to Algorithm 1 where the first 16 steps (we have kept Step 16 of Algorithm 1 in Algorithm 2 for convenience) and the last 9 steps are identical code (with the exception of Step 4 in Algorithm 1). Hence, we only elaborate on the new code (Steps 3∼22 in Algorithm 2), which checks whether all the disjunctions of a CNF are satisfied. The new code is only invoked for a CNF $c$ where there are at least $K$ posting lists that have $c$'s ID in their current entries (Step 2). We then initialize an array of integer counters (Step 4) where each integer corresponds to a disjunction of $c$ and is initialized to the negative number of $\notin$ predicates in that disjunction. For instance, if $c = (A \in \{1\} \vee B \in \{2\}) \wedge (C \notin \{3\} \vee D \in \{4\}) \wedge (E \notin \{5\} \vee F \notin \{6\})$, Counters is initialized to [0,-1,-2]. (Recall that the number of $\notin$ predicates per disjunction is saved in a structure separate from the inverted list during load time.)

At Step 5 in Algorithm 2 (all the steps from now on in this section refer to Algorithm 2) we know there are $K$ posting lists containing $c$'s id, but there could actually be more than $K$. Thus, we now scan all lists in the $K$-index, looking for ID $c$. Suppose when we look at list $L$, its current entry contains disjunction ID $d$. We then either increase Counter[$d$] (Step 11) if the entry has a $\notin$ annotation or set Counter[$d$] to 1 (Step 13) if the entry has an $\in$ annotation. In Steps 17∼20, we check if all the disjunctions of $c$ have been satisfied by looking at the counters. A positive counter value means that at least one $\in$ predicate has been satisfied for disjunction $d$ while a negative counter value means that at least one $\notin$ predicate has been satisfied. Hence, the only case where a disjunction is not satisfied is when the counter value is 0 (i.e., no $\in$ predicates have been satisfied and all $\notin$ predicates, if they exist, have been violated).

*Example.* Given the assignment $S : \{A = 1, C = 2\}$, the matching posting lists for $S$ from the inverted list of Figure 8 are shown in Figure 9. The weights are omitted for convenience because we do not score BE's yet.

Since the posting list skipping is similar to the example in Section 3, we focus on the disjunction checking for CNFs. When $K$=2, the CNFs that are checked in Steps 3∼22 are $c_2, c_3, c_4$ (notice that

**Figure 9: Posting lists for assignment $S$**

| K | Key | Posting List |
|---|---|---|
| 0 | $(A,1)$ | $(6,\notin,0)$ |
|  | $Z$ | $(6,\in,-1)$ |
| 1 | $(C,2)$ | $(5,\notin,1)$ |
|  | $(A,1)$ | $(5,\in,0)$ |
| 2 | $(A,1)$ | $(1,\in,0)$ $(2,\in,0)$ $(3,\in,0)$ $(4,\in,0)$ |
|  | $(C,2)$ | $(2,\in,0)$ $(3,\in,1)$ |
|  | $(A,1)$ | $(4,\in,1)$ |

$c_1$ is skipped because there is only one posting list for $c_1$). Starting from $c_2$, we initialize the Counter array to [0,0] (both disjunctions of $c_2$ contain no $\notin$ predicates) and scan posting lists $(A, 1)$ and $(C, 2)$. Since the entries for $c_2$ in both posting lists refer to disjunction 0, the final state of the counters is [1,0]. Since the second counter is 0, $c_2$ is not satisfied. We then start processing $c_3$. This time, the two entries for $c_3$ in posting lists $(A, 1)$ and $(C, 2)$ refer to disjunction IDs 0 and 1, respectively. Hence, the final state of the counters is [1,1], and we accept $c_3$ into $O$. Finally, $c_4$ is a case where one key $(A, 1)$ satisfies both disjunctions of the CNF. The final state of the counters is also [1,1] and we accept $c_4$ into $O$.

We now illustrate entries with $\notin$ annotations when $K$=1. Since $c_5$ has two posting lists with entries for $c_5$, we start checking the disjunctions of $c_5$ from Step 4. Since $c_5$ has one disjunction with zero $\notin$ predicates and another with two $\notin$ predicates, the counters are initialized to [0,-2]. We then view the current entry of the posting list $(A, 1)$ from Step 6 and set Counter[0] to 1 at Step 13. For the next posting list $(C, 2)$, we increment Counter[1] to -1 at Step 11 because the current entry is annotated by a $\notin$. The final Counter is thus [1,-1]. The first disjunction is satisfied because one $\in$ predicate is satisfied while the second disjunction is also satisfied because one $\notin$ predicate is satisfied. We thus accept $c_5$ into $O$.

Finally, we illustrate the handling of $Z$ when $K$=0. Since $c_6$ has two posting lists with entries for $c_6$, we start checking its disjunctions from Step 4. Since $c_6$ only has one disjunction with one $\notin$ predicate, Counter is initialized as [-1]. When viewing the current entry of the posting list $(A, 1)$, we increment the counter to 0. However, we ignore the next posting list $Z$. Since the final counter is 0, we do not accept $c_6$. The final solution $O$ is thus $\{3, 4, 5\}$.

*Complexity.* Compared to the Conjunction algorithm, the CNF algorithm has the additional step of scanning posting lists (Steps 5∼15), which takes $O(|S|)$ each time PLists[0].CurrEntry.ID = PLists[K-1].CurrEntry.ID (Step 2). Hence, the total complexity is $O(|S| \times |C| \times P_{avg})$ where $|C|$ is the number of CNFs and $P_{avg}$ is the average number of predicates per disjunction.

## 5. RANKING BOOLEAN EXPRESSIONS

The ranking algorithms for DNF and CNF BEs return the top-N matching BEs based on the BE scoring defined in Section 2.4. We do not present the code for the scoring algorithms, but only explain the necessary changes to Algorithms 1 and 2. Notice that our focus is on efficiently retrieving the top-N BEs given a scoring method, and not on developing different scoring methods or evaluating their effectiveness.

A naive solution is to maintain the top-N matching BEs in a heap while running the algorithms in Sections 3 and 4. As each new matching BE is found, we would compute its score and place it to the heap if it beats a previous BE. In this section, we discuss pruning techniques that can improve the performance of the naive solution.

*DNF Ranking Algorithm.* Ranking DNF BEs can be done with the Conjunction Algorithm (Algorithm 1) by maintaining a top-N queue of conjunctions and restricting them to have unique

DNF IDs within the queue. Since the score of a DNF BE is the maximum score of its conjunction scores, we only need to keep the highest conjunction score for each DNF ID.

We now use the weights in the inverted list to rank BEs. In Figure 2, the number next to each posting list key $(A, v)$ denotes the upper bound weight $UB(A, v)$. In each posting list entry, the third value denotes the weight $w_c(A, v)$ for conjunction $c$. For example, the key $(age, 4)$ in Figure 2 has a posting list inside the partition $K = 1$ and contains an entry representing $c_5$ where $w_{c_5}(age, 4) = 0.5$ and $UB(age, 4) = 3.0$. The upper bound $UB(Z)$ is defined as 0. In addition, each entry in $Z$ has weight 0.

We extend Algorithm 1 by adding the following two pruning techniques.

1. After sorting the posting lists in Step 14, the sum of $UB(A, v) \times w_S(A, v)$ for every posting list PLists[L] such that PLists[L]. CurrentEntry.ID $\leq$ PLists[$K$-1].CurrentEntry.ID is an upperbound for the score of the conjunction PLists[$K$-1].CurrentEntry. ID. If the upperbound is less than the Nth highest conjunction score, we can skip all the posting lists with CurrentEntry.ID less than or equal to PLists[$K$-1].CurrentEntry.ID and continue to the next while loop at Step 13.
2. Before processing PLists from Step 7, the sum of the top-$K$ $UB(A, v) \times w_S(A, v)$ values for all the posting lists in PLists is an upperbound of the score for all the matching conjunctions with size $K$. If the upperbound is less than the Nth highest conjunction score, we can skip processing PLists for the current $K$-index and continue to the next for loop at Step 4.

*Example.* Given the assignment $S$ :{$age = 3, state = NY, gender = F$}, the matching posting lists for $K$=2 from the inverted lists of Figure 2 are shown in Figure 10. Notice that we also added the assignment weights in the first column, assuming we are given $w_S(age, 3)$ = 0.8, $w_S(state, NY)$ = 1.0, and $w_S(gender, F)$ = 0.9. Suppose that we set N=1 (i.e., we only maintain the conjunction with the highest score). The conjunction $c_1$ is first accepted in Step 28 of Algorithm 1 because two posting lists have current entries for $c_1$. The score of $c_1$ is $w_1(state, NY) \times w_S(state, NY) + w_1(age, 3) \times w_S(age, 3)$ = 4.0×1.0 + 0.1×0.8 = 4.08. The Nth highest score is thus set to 4.08.

**Figure 10: Posting lists for $S$ where $K$=2**

| $w_S$ | Key & UB | Posting List |
|---|---|---|
| 1.0 | $(state, NY), 5.0$ | $\underline{(1, \in, 4.0)}$ |
| 0.8 | $(age, 3), 1.0$ | $\underline{(1, \in, 0.1)}$ $(2, \in, 0.1)$ $(3, \in, 0.2)$ |
| 0.9 | $(gender, F), 2.0$ | $\underline{(2, \in, 0.3)}$ |

The first pruning technique is illustrated in Figure 11 where the posting lists are sorted (Step 14, Algorithm 1) after accepting $c_1$. Before we check whether the first and second posting lists have the same conjunction in their current entries (Step 16), we compute the upperbound score of $c_2$ by computing $UB(age, 3) \times w_S(age, 3) + UB(gender, F) \times w_S(gender, F)$ = 1.0×0.8 + 2.0×0.9 = 2.6. Since 2.6 is smaller than the Nth score 4.08, we can immediately skip the first two posting lists to conjunction ID 2+1 = 3 without invoking Step 16 and continue to the next while loop at Step 13. We eventually finish processing the posting lists for $K$=2 and continue to $K$=1 from Step 4.

The second pruning technique is illustrated in Figure 12, which shows the posting lists for $K$=1. Before we process the posting lists from Step 6, we first derive the upperbound score for all the conjunctions in the $K$-index by computing $UB(age, 3) \times w_S(age, 3)$ = 1.0×0.8 = 0.8. Since 0.8 is less than the current Nth score 4.08,

**Figure 11: Sorted posting lists after accepting $c_1$**

| $w_S$ | Key & UB | Posting List |
|---|---|---|
| 0.8 | $(age, 3), 1.0$ | $(1, \in, 0.1)$ $\underline{(2, \in, 0.1)}$ $(3, \in, 0.2)$ |
| 0.9 | $(gender, F), 2.0$ | $\underline{(2, \in, 0.3)}$ |
| 1.0 | $(state, NY), 5.0$ | $(1, \in, 4.0)$ $\underline{\texttt{EOL}}$ |

we can immediately skip processing the posting lists for $K$=1. Similarly, we can also skip $K$=0 to return the final solution $c_1$, which has the highest score 4.08.

**Figure 12: Posting lists for $S$ where $K$=1**

| $w_S$ | Key & UB | Posting List |
|---|---|---|
| 0.8 | $(age, 3), 1.0$ | $\underline{(5, \in, 0.1)}$ |

*CNF Ranking Algorithm.* Ranking CNF BEs can be done with the CNF algorithm (Algorithm 2) by maintaining a top-N queue of CNF BEs. We can use the first pruning technique of the DNF ranking algorithm (exact same code) and use it in the CNF algorithm. Since the score of a CNF BE is the sum of the disjunction scores while the score of a disjunction is the maximum score of its predicates, the sum $UB(A, v) \times w_S(A, v)$ for every posting list PLists[L] where PLists[L].CurrentEntry.ID $\leq$ PLists[$K$-1].CurrentEntry.ID is still an upperbound for the score of the CNF of PLists[$K$-1].CurrentEntry.ID.

However, we cannot use a pruning technique that corresponds to the second pruning technique of the DNF ranking algorithm because more than $K$ disjunctions may contribute to the score of a CNF with size $K$ (i.e., disjunctions that contain both $\in$ and $\notin$ predicates do not count in the size of the CNF, but may have scores that add to the CNF score). Hence, the sum of the top-$K$ $UB(A, v) \times w_S(A, v)$ values in PLists is no longer an upperbound for the CNF score.

*Example.* Given the assignment $S : \{A = 1, C = 2\}$, the matching posting lists for $K$=2 from the inverted list of Figure 8 are shown in Figure 13 along with the given assignment weights $w_S(A, 1)$ = 0.1 and $w_S(C, 2)$=0.9. As we discussed in Section 4, the only matching CNFs in Figure 13 are $c_3$ and $c_4$. However, after we accept $c_3$ and derive the score $w_3(A, 1) \times w_S(A, 1)$ + $w_3(C, 2) \times w_S(C, 2)$ = 0.3×0.1 + 2.7×0.9 = 2.46, we can skip processing CNF ID 4 from Step 2 in Algorithm 2 because the upperbound of $c_4$ is $UB(A, 1) \times w_S(A, 1) + UB(A, 1) \times w_S(A, 1)$ = 0.5×0.1 + 0.5×0.1 = 0.1, which is smaller than 2.46.

**Figure 13: Posting lists for $S$ where $K$=2**

| $w_S$ | Key & UB | Posting List |
|---|---|---|
| 0.1 | $(A, 1), 0.5$ | $\underline{(1, \in, 0, 0.1)}$$(2, \in, 0, 0.3)$$(3, \in, 0, 0.3)$$(4, \in, 0, 0.1)$ |
| 0.9 | $(C, 2), 3.0$ | $\underline{(2, \in, 0, 2.5)}$$(3, \in, 1, 2.7)$ |
| 0.1 | $(A, 1), 0.5$ | $\underline{(4, \in, 1, 0.1)}$ |

## 6. LATENCY, SCALABILITY, UPDATES

We now turn to three main systems challenges in designing an efficient BE index — latency, scalability and updates. With regards to latency, applications such as online advertising and high-performance publish/subscribe systems have response time requirements of less than a few hundred milliseconds (for the matching component). With regards to scalability, display advertising systems have to process many billions of impressions every day. Finally, with regards to updates, the index has to be capable of handling a small number of updates in real-time (e.g., new subscriptions or advertisements); however, the update volume is typically many orders of magnitude less than the read volume. How do we adapt inverted lists for use in such environments?

Fortunately, it turns out that inverted lists have already been adapted for similar use in a different (familiar!) application — Web search (e.g., see [21]). The latency problem is solved by storing a read-optimized inverted list data structure in main-memory, without incurring the cost of disk reads. The scalability problem is solved by replicating and/or partitioning the index (based on documents, or in our case, BEs) across multiple machines. The update problem is solved by maintaining a small "tail" data structure for updates, in addition to the main read-only data structure; periodically, the main index is rebuilt (offline) to absorb the changes in the tail, and the newly built index is used to replace the old index without a perceptible impact on performance. We use very similar techniques in our implementation, and as shown in the next section, the resulting implementation meets our goals, without consuming excessive main-memory resources (e.g., an index for a million BEs consumes less than 100MB).

## 7. EXPERIMENTS

In this section, we evaluate our DNF and CNF algorithms on synthetic datasets from an online advertising application and compare their performance to other efficient algorithms for BE matching. We address the questions of how our algorithms compare against others and how our algorithms perform in different scenarios such as different DNF sizes. All algorithms were implemented in C++, and our experiments were run on a 2.5GHz Intel(R) Xeon(R) processor with 16GB of RAM.

*Data Set.* We used a 1,000 random subset of 1 million real display advertising adlogs for our assignments. An adlog is the information of a user visiting a web page and consists of attribute name and value pairs. For example, a man living in California who is interested in sports may produce the adlog $Gender = M \wedge State = CA \wedge InterestSports = true$. In addition for each assignment, we added a month attribute (randomly selected) indicating the month of a certain year (say 2009).

We used synthetic BE workloads generated from statistics of real display advertising contracts (the maximum workload size we generated was 1 million BEs). A contract is a conjunction of $\in$ predicates and specifies the demographic of users an advertiser is interested in. In order to scale the contracts and to vary the structure of the Boolean expressions, we generated BEs based on the statistics of the contracts. We first extracted the attribute names and the possible sets of values for each name (each set containing all the values in a single $\in$ predicate) along with their frequencies within the contracts. We then generated DNF and CNF BEs by choosing attribute names and possible sets of values based on the frequencies (e.g., the more frequent an attribute name appears in contracts, the more likely it was used in the generated BEs).

We explain in detail how we generated DNF BEs. (The CNF BEs were generated in a similar fashion.) First, we generated the number of conjunctions within a DNF based on a Zipfian number generator. Given an exponent number $e$, we chose an integer from a range 1$\sim$20 that had a Zipfian distribution with exponent $e$. For each conjunction in the DNF, we generated the number of predicates based on the real distribution of the contract sizes. For each predicate in a conjunction, we selected an attribute name and a set of values based on the frequencies above. We also made sure the same attribute name did not appear multiple times within the same conjunction. The predicate was either an $\in$ or $\notin$ predicate based on a given probability $n$. Starting from the second conjunction generated for a DNF, we simulated correlations between conjunctions by copying a portion $p$ of predicates from the previously generated conjunction with a probability of $q$. Finally, we added in each conjunction a predicate containing a random month of 2009 (For CNF BEs, we added a separate disjunction of the month instead of inserting month predicates in each disjunction). Conjunctions of the same DNF were given the same month. The month predicate was used to adjust the "selectivity" of the BEs (i.e., the inverse probability of a BE satisfying an assignment). Table 1 shows the parameters and average values of our BE workloads.

**Table 1: Parameters and average values**

| Parameter | Description | Value(s) |
|---|---|---|
| $e$ | Zipfian exponent number | 2.0$\sim$3.0 |
| $p$ | Portion of preds to copy | 0.5 |
| $q$ | Probability of copying | 0.5 |
| $n$ | Portion of $\notin$ predicates | 0.1 |
| Avg val | Description | Value(s) |
| $\lvert S\rvert_{avg}$ | Avg # keys in assignment | 91 |
| $\lvert \text{DNF}\rvert_{avg}$ | Avg DNF size | 1.6$\sim$3.5 |
| $\lvert \text{Conj}\rvert_{avg}$ | Avg # preds in conjunction | 3.65 |
| $\lvert \text{CNF}\rvert_{avg}$ | Avg CNF size | 2.6$\sim$4.5 |
| $\lvert \text{Disj}\rvert_{avg}$ | Avg # preds in disjunction | 2.65 |

*Weight Generation.* We also generated the upperbounds and weights for all the keys in order to run the DNF and CNF ranking algorithms. We first set the upperbound of a key $(A, v)$ as the inverse of the frequency of $(A, v)$ appearing in the 1,000 assignments. For example, if $(age, 1)$ appeared in 100 assignments, then $UB(age, 1)$=0.1. The upperbound thus reflects the "rarity" of a key. (Notice that our model does not restrict the way of generating upperbounds and any other generation scheme can be used.) We then generated for each $w_c(A, v)$ a random gaussian with an average of $0.8 \times UB(A, v)$ and a variance of $0.05 \times UB(A, v)$ (If the weight exceeded $UB(A, v)$, we set the weight to $UB(A, v)$). Again, the weights can be generated in any other way. For assignments, each key was given a random weight from 0 to 1.

*Algorithms.* In Section 7.1, we compare the DNF algorithm with the algorithms below. Since the Le Subscribe and SIFT algorithms only run on conjunctions, we simply split all the DNF BEs into conjunctions and ran the two algorithms on the conjunctions. We also extended Le Subscribe and SIFT to support $\notin$ predicates and multi-valued predicates in a straightforward manner. For example, while the original Le Subscribe stores for each BE an array of pointers to boolean values for its predicates, we added a bit on each pointer to distinguish $\in$ or $\notin$ predicates.

- **Le Subscribe:** A clustering-based algorithm [11] that groups BEs by common subexpressions. Only the clusters whose subexpressions are satisfied by the assignment are fully evaluated. While there were several versions of Le Subscribe, we implemented the Propagation algorithm, which clusters each conjunction on its most selective predicate and is considered the fastest algorithm for high-dimensional data among the different versions.
- **SIFT**: A counting algorithm [25] that uses an inverted list for searching BEs. For all matching posting lists, SIFT scans each entry and increases/decreases the count of satisfied predicates for each conjunction. Only the conjunctions that have a count of at least the number of $\in$ predicates are accepted.
- **SCAN:** An exhaustive algorithm that scans and evalutes all BEs one by one for each assignment. While there are optimized techniques for evaluating one BE against assignments [19], 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.

We evaluate our CNF algorithm in Section 7.2. Since there were no related works to compare the CNF algorithm with, we implemented an improved version of SCAN (called PSCAN) where only the BEs in the posting lists of the assignment are fully evaluated. The PSCAN algorithm is thus similar to the SIFT algorithm except that there are no counters involved.

Finally, we compare the DNF and CNF algorithms with their ranking algorithms in Section 7.3.

*Inverted List Construction.* Constructing the inverted list offline includes parsing the BEs from the input file, deduplicating identical BEs, and loading the BEs into the inverted list. An inverted list of 1M DNF BEs without weights takes 7 minutes to load and has a size of 35MB (derived by summing the sizes of the key lookup hash table and the posting list entries). An inverted list of 1M CNF BEs without weights takes 1.5 hours to load and has a size of 91MB. In our experiments, we only measure the online runtimes of our algorithms after the inverted list is constructed offline.

## 7.1 DNF Algorithm

*Selectivity Impact.* We define the selectivity of a BE workload as the inverse of the probability that a BE matches an assignment. We can create various scenarios with different selectivity values by adjusting the range of random months assigned to BEs. If the range is large and the months are spread out (say January to December), then few BEs match with an assignment. On the other hand, if we only use one month (say January) for all BEs, we get the most matches (assuming the assignment also has the month January) and thus the lowest selectivity.

Figure 14 compares the average matching times (i.e., the average times to search all matching BEs given an assignment) for the DNF, Le Subscribe, and SIFT algorithms on 1 million DNF BEs while varying the probability of a BE matching an assignment. For small probabilities, the DNF algorithm is slightly slower than Le Subscribe. The reason is that there is a larger initial overhead for the DNF algorithm to run compared to Le Subscribe (e.g., scanning all the possible $K$-indexes and sorting posting lists). Le Subscribe on the other hand does little work when few BEs match. However, as the match probability increases, the DNF algorithm shows a sublinear increase in runtime while Le Subscribe has a more linear increase in runtime. A significant factor of the good performance of the DNF algorithm is the internal optimization of deduplicating conjunctions before loading them into the inverted list. Compared to the SIFT algorithm, the DNF algorithm is 11.2 to 20.9 times faster.

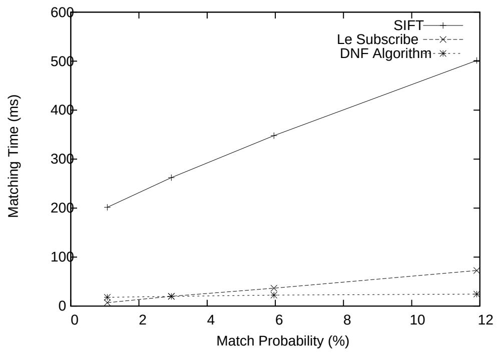

**Figure 14: Selectivity impact, 1M DNF BEs**

*High Dimension Impact.* We define the dimension of a BE workload as the number of different attribute names that can appear in the BEs. Our default DNF BE workload has a dimension of 1461, reflecting the high dimension of the contract data. In order to generate workloads of lower dimensions, we removed attribute names that had low frequencies and re-generated the BE workload.

Figure 15 compares the matching times of the DNF, Le Subscribe, and SIFT algorithms on 1 million DNF BEs while varying the dimension of the workload. (The match probability was 11.91%.) A noticeable trend is that the DNF algorithm improves in runtime for low-dimension BEs. The main reason is that there were fewer unique conjunctions to load into the inverted list because of the fewer attribute names and values to choose from when generating the BEs. On the other hand, Le Subscribe runs slightly slower for low-dimension BEs. Although the DNF algorithm seems to be much faster than Le Subscribe for low-dimension BEs, it is important to understand that we have implemented the Propagation algorithm [11], which only clusters BEs on their single most selective predicates, and not the other versions of Le Subscribe that cluster on multiple predicates. While the Propagation algorithm has a relatively bad performance on low-dimension BEs, it is the best algorithm for high-dimension BEs, and we are more focused on comparing the DNF algorithm with the Propagation algorithm on high-dimension BEs. We suspect that the multiple predicate clustering algorithms of Le Subscribe will perform much better for low-dimension BEs. In summary, the DNF algorithm performs the best for high-dimension BEs.

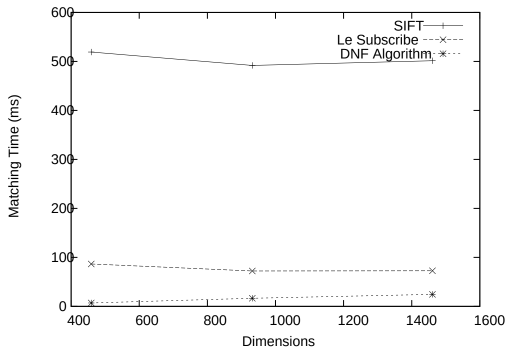

**Figure 15: Dimension impact, 1M DNF BEs**

*BE Size Impact.* Figure 16 compares the matching times of the DNF, Le Subscribe, and SIFT algorithms on 1 million DNF BEs while varying the DNF size (i.e., the average number of conjunctions per DNF) with the Zipfian exponent $e$. (The match probability was 11.91%.) For a high exponent, the distribution of DNF sizes becomes more skewed, lowering the average DNF size. For a low exponent, the size distribution becomes flat, and a DNF BE has a higher chance of having many conjunctions. In our experiments, we used three exponents for $e$ – 3, 2.5, and 2 – to generate the average DNF sizes of 1.6, 2.3, and 3.5, respectively. As the average DNF size grows by 2.19 times, the runtimes for the DNF, Le Subscribe, and SIFT algorithms grow by 1.9, 4.3, and 2.1 times, respectively. Hence, the DNF algorithm shows the best scalability for larger BE sizes.

*Scalability.* Figure 17 compares the scalability of the DNF, Le Subscribe, SIFT, and SCAN algorithms for a range of 0.2 to 1 million DNF BEs where the match probability was 11.91%. The DNF algorithm is up to two orders of magnitude faster than SCAN. Also, the DNF algorithm is 1.58∼2.97 times faster than Le Subscribe and 11.2∼20.5 times faster than SIFT.

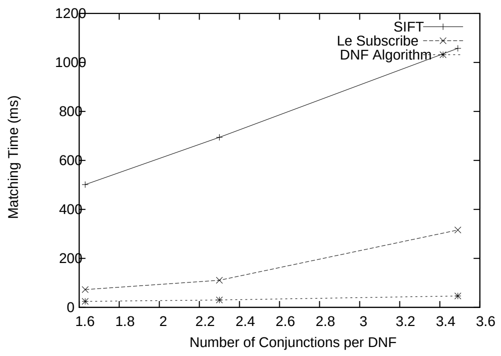

**Figure 16: DNF size impact, 1M DNF BEs**

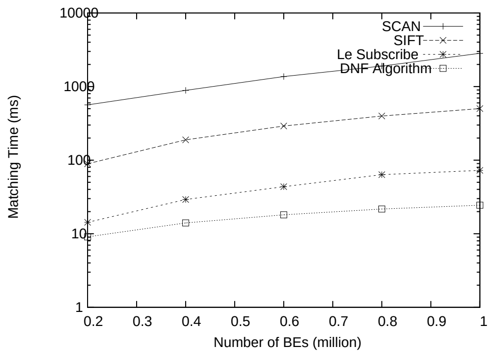

**Figure 17: DNF scalability**

## 7.2 CNF Algorithm

We now show the performance of the CNF algorithm. Figure 18 compares the matching time performance of the CNF algorithm against the PSCAN and SCAN algorithms for 0.2 to 1 million CNF BEs where the match probability was 5.16%. As a result, the CNF algorithm is 1.9∼2.6 times faster than PSCAN and 10.5∼12.7 times faster than SCAN.

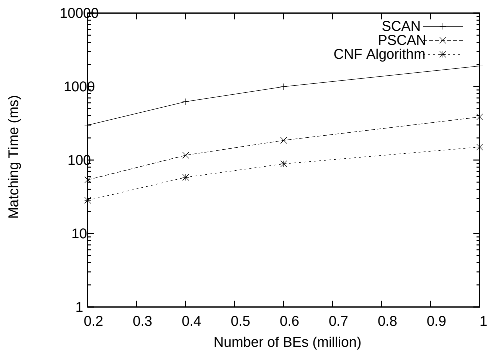

**Figure 18: CNF scalability**

## 7.3 Ranking Algorithms

We show the runtime performances of the ranking algorithms. Figure 19 compares the matching times of the DNF and CNF algorithms with their ranking algorithms for 1 million DNF BEs and 0.4 million CNF BEs. While the DNF and CNF algorithms return all the matching BEs given an assignment, the DNF and CNF ranking algorithms only return the top-N matching BEs with their scores. The match probabilities of the CNF results were relatively lower than those of the DNF results because the CNF expressions, being conjunctions of disjunctions, were in general harder to satisfy. Both ranking algorithms returns the top 5 matching BEs. The DNF ranking algorithm is 19% slower than the DNF algorithm for the lowest match probability, but 11% faster for the highest match probability. The slower speed for a low match probability illustrates the overhead of the computation needed for the additional skipping. However, as more BEs match an assignment (i.e., the match probability increases), we are more likely to get high scores of matching BEs that can prune further BEs. Thus, the DNF ranking algorithm is most effective when there are many matching BEs. The CNF ranking algorithm significantly outperforms the CNF algorithm by 35∼45% where the runtime gap increases as the match probability increases. Compared to the DNF ranking algorithm, the CNF ranking algorithm has more opportunities for pruning because the CNF BEs tend to have more varying scores. In summary, the additional pruning for the ranking algorithms can improve the performances of the non-ranking algorithms for high match probabilities.

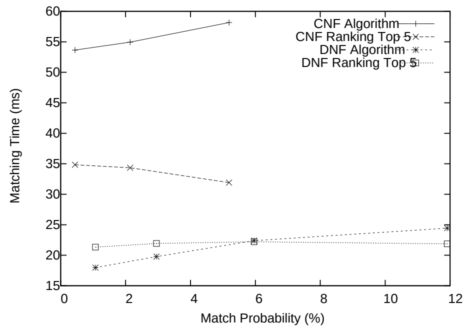

**Figure 19: Ranking algorithm performance**

## 8. RELATED WORK

Various publish/subscribe systems (see the surveys [10] for an overview) propose indexing techniques for complex expressions. Both SIFT [25] and Le Subscribe [11] are efficient publish/subscribe algorithms that index conjunctions. Carzaniga et al. [8] extends SIFT to index DNF expressions. In comparison, our proposed approach is more efficient for DNF expressions (Section 7), and additionally supports multi-valued attributes with NOTs, CNF BEs, and ranking. Cayuga [4] and SASE [22] are examples of "stateful" publish/subscribe systems, which index complex subscription expressions that can monitor a *sequence* of events (e.g., notify the subscriber when the stock price of IBM drops by more than 10% within a 5 day window). This work is complementary to our work in the sense it considers complex expressions of a different kind - i.e., expressions over multiple events - while our focus is on evaluating complex Boolean operations over a single event. A recent line of work focuses on ranked publish/subscribe [16] where only the "best" BEs are chosen instead of all the satisfied BEs. Our algorithms also support ranking, but differ from the above work in several ways. First, we support general BEs instead of intervals in a multi-dimensional space. Also, our algorithms perform very well for high-dimensional data while the above work only performs well on very small dimensions [16].

IR systems [21, 26], which efficiently search documents given a query, have been heavily studied. Our application is different in that we are searching for queries (BEs) given the data (instead of the other way around), and that we exploit the syntax of the complex queries in order to exactly find the satisfied BEs. A related piece of work is the WAND algorithm [6], which efficiently searches documents whose weighted scores against a given keyword query are larger than a certain threshold. Our method of skipping over inverted lists is inspired by the WAND algorithm, but extends it to deal with DNF and CNF BEs (including $\notin$) instead of documents of keywords, and also extends it to enable top-N ranking while enforcing strict Boolean semantics.

Scalable trigger systems [15] can also be used to support BE matching where each trigger supports one BE. While triggers provide the expressiveness of SQL, they are generally not as scalable as specialized BE matching systems. The Oracle RDBMS supports the EVALUATE operator and can store entire BEs under a column of a special data type [24]. Indexes called Expression Filters are used to efficiently process the EVALUATE operator on a large set of BEs. While users can take advantage of the expressive power of SQL to support complex BEs, Expression Filters are only designed for a small number of dimensions (each indexed dimension results in a SQL sub-query, and most sparse dimensions are not indexed), and only for DNF expressions with single-valued predicates. Further, RDBMSs do not support ranking for triggers or BEs.

Expert systems [13] use efficient matching algorithms [18, 12] to match patterns against objects. In comparison, our algorithms solve the more general problem of matching DNF/CNF BEs against objects (notice that a pattern can be expressed as a conjunction) and also support BE ranking. Policy compliance checking [2, 3] involves matching credentials against policies that are possibly BEs. However, there is more emphasis on the expressiveness of the policies than on the matching performance. At best, the number of policies scale to the order of thousands while our indexes can store millions of BEs.

Finally, complementary to our work is a technique for indexing regular expressions [9], which have a different structure than BEs.

## 9. CONCLUSION

We have proposed a general solution for efficiently matching complex boolean expressions (BEs) using the inverted list data structure. While there are many inverted list query processing algorithms that support complex *query expressions*, our methods solve the inverse problem of complex *indexed expressions*. Our model supports DNF and CNF BEs with $\in$ and $\notin$ primitives. Our techniques can be used in online advertising and in general any content-based publish/subscribe system. We have experimentally shown that our algorithms are scalable especially for high-dimensional data, and are several times faster than prior approaches.

An important feature of our algorithms is their ability to rank BEs based on their relevance score to the given assignment. We have experimentally shown that the DNF ranking algorithm improves the non-ranking version by up to 11% while the CNF ranking algorithm improves by up to 45%. The ranking algorithms can be used in emerging settings that require ranked publish/subscribe.

## 10. 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 *PODC*, pages 53–61, 1999.

[2] M. Blaze, J. Feigenbaum, and M. Strauss. Compliance checking in the policymaker trust management system. In *Financial Cryptography*, pages 254–274, 1998.

[3] K. Borders, X. Zhao, and A. Prakash. Cpol: high-performance policy evaluation. In *ACM Conference on Computer and Communications Security*, pages 147–157, 2005.

[4] 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 *SIGMOD '07*, pages 1100–1102, New York, NY, USA, 2007. ACM.

[5] A. Broder, M. Fontoura, V. Josifovski, and L. Riedel. A semantic approach to contextual advertising. In *SIGIR '07*, pages 559–566, New York, NY, USA, 2007. ACM.

[6] 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.

[7] E. W. Brown. Fast evaluation of structured queries for information retrieval. In E. A. Fox, P. Ingwersen, and R. Fidel, editors, *SIGIR'95, Seattle, Washington, USA, July 9-13, 1995*, pages 30–38. ACM Press, 1995.

[8] A. Carzaniga and A. L. Wolf. Forwarding in a content-based network. In *SIGCOMM*, pages 163–174, 2003.

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

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

[11] 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.

[12] C. Forgy. Rete: A fast algorithm for the many pattern/many object pattern match problem. *Artificial Intelligences*, 19(1):17–37, 1982.

[13] J. C. Giarratano and G. D. Riley. *Expert Systems: Principles and Programming, Third Edition: Principles and Programming*. Course Technology, 3 edition, February 1998.

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

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

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

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

[18] D. P. Miranker. TREAT: A better match algorithm for AI. In K. S. H. Forbus, editor, *Proceedings of the 6th National Conference on Artificial Intelligence*, pages 42–47, Seattle, WA, July 1987. Morgan Kaufmann.

[19] K. A. Ross. Selection conditions in main memory. *ACM Trans. Database Syst.*, 29:132–161, 2004.

[20] S. J. Russell and Norvig. *Artificial Intelligence: A Modern Approach (Second Edition)*. Prentice Hall, 2003.

[21] I. H. Witten, A. Moffat, and T. C. Bell. *Managing Gigabytes: Compressing and Indexing Documents and Images*. Morgan Kaufmann Publishers, San Francisco, CA, 1999.

[22] E. Wu, Y. Diao, and S. Rizvi. High-performance complex event processing over streams. In *SIGMOD Conference*, pages 407–418, 2006.

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

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

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

[26] J. Zobel and A. Moffat. Inverted files for text search engines. *ACM Comput. Surv.*, 38(2), 2006.