Efficiently Evaluating Graph Constraints in Content-Based Publish/Subscribe

Andrei Broder[1] Shirshanka Das[2*] Marcus Fontoura[3*] Bhaskar Ghosh[2*]
Vanja Josifovski[1] Jayavel Shanmugasundaram[3*] Sergei Vassilvitski[1]
1. Yahoo! Research, 701 First Ave., Sunnyvale, CA 94089
2. LinkedIn, 2029 Stierlin Court, Mountain View, CA 94043
3. Google Inc., 1600 Amphitheatre Pkwy, Mountain View, CA 94043
broder@yahoo-inc.com, sdas@linkedin.com, marcusf@google.com, bghosh@linkedin.com,
vanjaj@yahoo-inc.com, jaishan@google.com, sergei@yahoo-inc.com

ABSTRACT

We introduce the problem of evaluating graph constraints in content-based publish/subscribe (pub/sub) systems. This problem formulation extends traditional content-based pub/sub systems in the following manner: publishers and subscribers are connected via a (logical) directed graph G with node and edge constraints, which limits the set of valid paths between them. Such graph constraints can be used to model a Web advertising exchange (where there may be restrictions on how advertising networks can connect advertisers and publishers) and content delivery problems in social networks (where there may be restrictions on how information can be shared via the social graph). In this context, we develop efficient algorithms for evaluating graph constraints over arbitrary directed graphs G. We also present experimental results that demonstrate the effectiveness and scalability of the proposed algorithms using a realistic dataset from Yahoo!'s Web advertising exchange.

Categories and Subject Descriptors

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

General Terms

Algorithms, Performance

Keywords

Graph Constraints, Pub/sub, Indexing

1. INTRODUCTION

Content-based publish/subscribe (pub/sub) systems are designed to match a large number of content-based subscriptions (e.g., weather = "rain" ∧ temperature < 0) to a rapid stream of events (e.g., weather ← "rain", temperature ← -1), and to do so in a scalable and efficient manner. In this context, a subscription is said to match an event if the subscription predicate evaluates to true over the event specification. There has been a large and rich body of work on designing efficient content-based indices to enable efficient matching (e.g., [1, 8, 9, 11, 18, 22], and on developing efficient dissemination strategies for delivering the matched events to the subscribers (e.g., [2, 3, 5, 6, 10, 20, 21]).

One of the implicit assumptions in current content-based pub/sub systems is that all subscribers are (logically) directly connected to the event sources, i.e., a subscription matches an event so long as its predicate evaluates to true over the event. However, this assumption is too strong for many emerging applications such as Web advertising and social networks, where (logical) *intermediaries* exist between subscribers and publishers and have a say as to when a subscription matches an event. For instance, in Web advertising, advertisers (subscribers) are typically connected to publishers (event sources) through intermediate advertising networks, which enforce their own rules (predicates) on which events can be matched against which subscriptions based on various business rules and ad quality concerns. Similarly, in social networks, status updates (events) flow through the social graph and intermediate nodes in the graph (i.e., persons connected to the event producers and consumers) may have privacy settings that disallow certain updates from some connections to flow to other connections.

The key point in these applications is the fact that the constraints on the intermediate graph define the semantics of a match, and may restrict the set of subscribers eligible for the event. This is semantically different from the problem of disseminating the results of matches to subscribers in a physically distributed network (e.g., [2, 3, 5, 6, 10, 20, 21]). In that case, the semantics of the matches remains the same, based on subscribers being logically directly connected to publishers. In contrast, the new semantics of matching specified by intermediaries changes the set of matches itself, and is orthogonal to the dissemination problem once the matches are performed. We now describe the Web advertising and social networking applications in more detail.

Web advertising exchange. A Web advertising exchange connects content publishers to advertisers through advertising networks. Advertising networks enable publishers to reach a wider set of advertisers, and also enforce content and ad quality rules that ensure that publishers see high-quality ads and that advertisers reach high-quality content sites. One example is shown in Figure 1. In this simple example, the fact that the three advertising networks are connected allows the three publishers to have access to the four available advertisers in the system. However, to con-

*Work done while the author was at Yahoo!

WWW 2011, March 28–April 1, 2011, Hyderabad, India.
ACM 978-1-4503-0632-4/11/03.

trol for quality and other business reasons, each ad network may specify targeting attributes, constraining the types of opportunities that they are willing to forward. Note that these constraints encode complex business rules and relationships and are expressed as arbitrary Boolean formulae. For instance, an ad network may be interested only in traffic from sports and finance pages with users older than 30, as is the case for Net_1 in Figure 1. In a different scenario, an ad network may target only mobile device users in California who are also interested in sports or those from New York with an interest in travel.

Advertisers themselves specify campaigns (subscriptions) based on targeting attributes, which describe the set of user visits that they wish to show ads against. The entire Web advertising exchange graph is typically hosted on a (logically centralized) Web exchange, such as Yahoo!'s RightMedia exchange [1]. The individual advertisers, and advertising networks are allowed to changes their subscriptions and predicates, but all of these changes are made on the hosted Web advertising exchange.

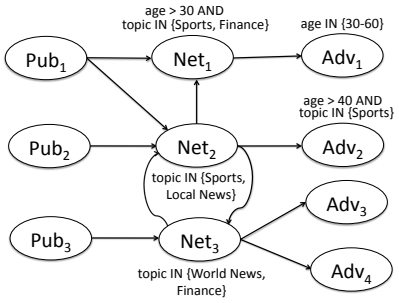


Figure 1: Sample Web advertising exchange.

Every time a publisher's Web page is visited, an advertising opportunity arises. At that time, an event from the publisher is generated indicating the targeting attributes for the opportunity. Such targeting attributes can include information about the page (such as the page content and its main topics), information about the available advertising slots (such as number of ads in the page and their maximum dimensions in pixels), and information about the user (such as user demographics and geographic location). Given an ad opportunity (event), the advertising exchange is responsible for notifying all advertisers (subscribers) with least one valid path from the publisher that originated the opportunity. That is, each node in the path must satisfy its targeting constraints. In Figure 1's example, if a user with age 35 visits a sports page from publisher Pub_1, networks Net_1 and Net_2 and advertiser Adv_1 would satisfy the targeting and graph constraints for the event. Therefore Adv_1 would be the only valid advertiser for the event.

Social networks. In social networks, users are connected to each other forming a connection graph. Consider an application where every user subscribes and produces a stream of "interesting tidbits." Such tidbits could include music concerts, books of interest and status updates. A user can choose to incorporate in his collection the tidbits produced by other users in the network, but with some restrictions. For instance, he or she may be only interested in publishing and aggregating tidbits related to music. In this case, each user is acting as an intermediary between other indirectly connected users, selectively aggregating user feeds and publishing them to other users. Given a user tidbit, the system needs to find all valid paths to users through connections in the social graph, while satisfying restrictions on intermediate nodes (e.g., only publish music related content).

The "News Feed" feature in Facebook[2] can be viewed as a simplified version of the tidbit idea, with the restriction that in Facebook the status updates are delivered only to the immediate friends of a user (i.e., only to users that are one-hop away from the publishing user) and users have limited control on which updates are of their interest and who should receive their updates. In Twitter[3] intermediate services can act as content dissemination nodes accumulating and redistributing tweets to interested subscribers.

Outline of the solution. Given the above applications, we can now formalize the problem as follows. Publishers and subscribers are connected through a graph of intermediary nodes. The overall system can be represented by a directed graph with three types of nodes (i) publishers, (ii) intermediaries and (iii) subscribers. For a given event, each node in the graph can act as a publisher, an intermediary or a subscriber, with the restriction that nodes with no incoming edges can only act as publishers and nodes with no outgoing edges can only act as subscribers. In this setting, events from a publisher p can only be delivered to subscribers that have at least one path from p in the graph. Moreover, the path from p must satisfy the constraints on its nodes and edges.

One naive solution for this problem is to use existing solutions for content-based indexing [1, 8, 9, 11, 18, 22] and to post-filter the results, discarding subscribers that do not have valid paths leading to them. We show that this naive solution can be greatly improved by keeping track of node reachability while using an index to evaluate the graph constraints. When the constraints on the nodes are simple subset constraints [14] showed how to preprocess the graph into a set of overlapping trees that would allow for faster online evaluation. However, their solution does not extend to the scenario when the constraints can be arbitrary Boolean expressions. One could flatten any Boolean expression into a DNF and then use the subset algorithm presented in [14] but that would result in an exponential blow up in the solution cost [12].

Our solution works as follows: we create one entity in the content-based index for each node in the graph. While existing solutions would simply use this index to evaluate the targeting attributes and return the index results as the set of valid subscribers, in our setting this is a required but not sufficient condition. We also much check that there is at least one valid path from the publisher to each node returned by the index. To verify reachability, our algorithms use an efficient representation of the graph structure and use an "online" breath-first search (BFS) from the publisher node to compute the reachable set, using the nodes returned by the index as input. This ensures that every matching subscription satisfies both the path predicates as well as the targeting constraints.

We also exploit the structure of the graph to speedup evaluation by skipping over nodes that are unreachable. For DAGs we use the topological sort order of the graph to de-

[1] rightmedia.com

[2] facebook.com

[3] twitter.com

cide which nodes are unreachable without having to retrieve them from the index. In the case of general directed graphs with cycles, we compute a condensation of the graph by mapping each strongly connected component (SCC) into a single node. We then use the resulting condensed DAG to avoid retrieving from the index nodes that belong to unreachable SCCs.

We have implemented the proposed techniques and evaluated them using data from Yahoo!'s Web advertising exchange. In order to meet the latency needs of Web advertising, the indices and stored and evaluated in main-memory — this is quite feasible because the size of the intermediary graph is by itself relatively compact given current main-memory sizes (this is also true for the social graph), while the graph nodes corresponding to the event producers and consumers can be partitioned on multiple machines and the matching results can be simply aggregated to get the full set of matches. Our performance results show that the proposed techniques are scalable and efficient, and offer significant performance benefits over approaches that do not explicitly consider graph constraints during query evaluation.

Contributions and roadmap. In summary, the main contributions of this paper are:

- A formal definition of the problem of evaluating graph constraints in content-based pub/sub systems (Section 2).
- Algorithms to solve the problem of evaluating graph constraints in content-based pub/sub systems. We propose algorithms that work for DAGs (Section 3.1) and *any* directed graph G (Section 3.2). These algorithms are currently deployed in production at the core of Yahoo!'s RightMedia advertising exchange.
- Correctness and efficiency proofs, showing that the proposed algorithm is correct and optimal (Sections 3.1, 3.2, 3.4).
- Experimental evaluation of the proposed algorithms that demonstrate the benefits of our solution in a realistic Web advertising exchange application scenario (Section 4).

2. PROBLEM DEFINITION

In this paper we focus on the problem of evaluating graph constraints in content-based pub/sub. This is modeled as a graph G in which publishers and subscribers are connected through nodes an edges. Our queries (events) have two components (a) a start node s, representing the publisher, and (b) a set Q of labels representing the event. We model the network by a directed graph $G = (N, E)$, with each node $n \in N$ having an associated set of labels L_n corresponding to its targeting attributes. We define a directed path P to be valid for Q if P is a path in G and the set of labels L_n associated to every node n in P is valid for Q, with respect to a matching function $match(Q, L_n)$. The output of the system is defined as the set of nodes in G reachable from s via valid paths for Q. For simplicity we restrict our presentation to the case when targeting attributes can only be placed on nodes. The problem of edge based targeting is equivalent, as any edge can be split into two and the targeting constraints applied to the new node.

The function $match(Q, L_n)$ is application specific, and is given ahead of time. For example, it can be defined as "superset," meaning that the set of labels L_n must be a superset of the labels in Q. This is the usual semantics in information retrieval systems, where every query label must be present in the qualifying documents. If $match(Q, L_n)$ is "subset," the targeting attributes specified for each node must be a subset of the event attributes (e.g. a subscriber is interested in sports pages only and the event identifies a page as belonging to both the sports and news categories).

We abstract away the details of the $match(Q, L_n)$ function, and instead assume that each node has a unique node id and that there is an underlying index which returns matching nodes in order of their ids. The index implements a $getNextEntity(Q, n)$ function call which returns the next matching node with node id at least n.

To describe the algorithms, we will use the following notation throughout:

- *Graph* $G = (N, E)$. A representation of the graph that efficiently returns the children of a given node. In other words, an efficient implementation of $C_n = \{v \in N, (n, v) \in E\}$, which denote the set of children of node n.
- *Valid nodes* $N_V \subseteq N$. The set of nodes that are valid with respect to its targeting attributes. This means that for every node $n \in N_V$, $match(Q, L_n)$ is true.
- *Reachable nodes* $N_R \subseteq N$. The set of of nodes that are reachable from s using only nodes in N_V. Note that every node $n \in N_R$ is guaranteed to have a path from s consisting only of valid nodes. However, n itself may not be valid.
- *Result nodes.* The set the nodes that should be returned as query results. This is exactly $N_R \cap N_V$: the set of valid nodes reachable through valid paths.

Figure 2 shows the general architecture of our solution, where the *evaluator* component uses both the *index* and the *graph* structures simultaneously to compute the set of valid subscribers for each event. Our solution can use any index structure (e.g. [9, 11, 12, 22]) that provides an interface for retrieving the *valid* nodes for a given event. The graph component is responsible for returning the children of a given node and it is used during our BFS evaluation of reachability. The evaluator is responsible for computing the intersection of the reachable and valid nodes.

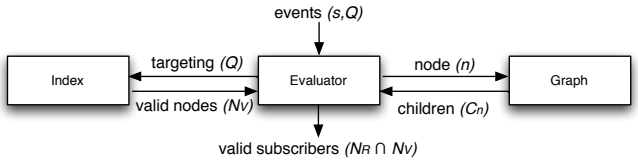


Figure 2: Overall systems architecture – the evaluator uses the index and the graph structures simultaneously to evaluate the set of valid subscribers for a given event.

3. EVALUATION ALGORITHMS

We first present our evaluation algorithms for DAGs, before generalizing them to arbitrary graphs.

3.1 DAG evaluation

We begin by describing an algorithm for the special case when the graph G is a DAG. We assign node ids in the order

of a topological sort of G. This maintains the invariant that for any node n, its children $v \in C_n$ come later in the node id order. The algorithm is shown in Figure 3.

The algorithm begins by adding the children of the start node s to the reachable set N_R (line 1). It then retrieves the first valid node with node id greater than s from the index (line 3). If the retrieved node is already in the reachable set, we know it is both reachable and valid and we add it to the results set (line 5). Moreover, we also know that its children are reachable and we add them to the reachable set (line 6). We then resume the search using the index to retrieve the next valid node after node id $n+1$. At the end of processing we return the nodes that are in the result set (line 10).

```
evaluate(s, Q)
  // Returns the valid and reachable nodes.
  1. reachable.add(graph.children(s));
  2. skipIndex = s + 1;
  3. while (n = index.getNextEntity(Q, skipIndex)) {
  4.    if (reachable.contains(n)) {
  5.       result.add(n);
  6.       reachable.add(children(n));
  7.    }
  8.    skipIndex = n + 1;
  9. }
 10.return result.nodes();
```

Figure 3: Query evaluation algorithm for DAGs.

Figure 4 shows a simple DAG where each node is annotated with its node id. Node ids are assigned in topological sort order in an offline process before query evaluation starts. The figure also show the labels associated with each node. Let us consider that for this example the start node is $s = 0$ and the query labels are $Q = \{A, B, C\}$. Function $match(Q, L_n)$ is "subset," meaning that node n is valid w.r.t. its targeting attributes if and only if $L_n \subseteq Q$. Given this $match(Q, L_n)$ semantics, the set of valid nodes N_V is {2,3,5,6,8}.

Figure 4's table shows the valid, reachable and result sets after each valid node is returned by the index. When nodes 2 and 3 are returned by the index they are simply discarded since they are not reachable. When node 5 is returned we know it is reachable, and therefore, we add it to the result set and we add its children to the reachable set. The same happens for nodes 6 and 8.

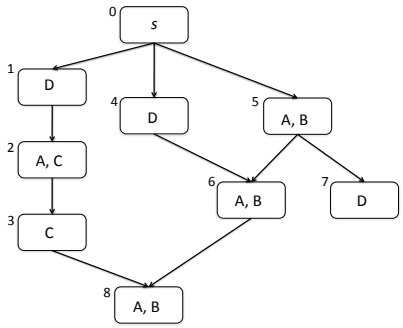
To prove the algorithm's correctness, we observe the following important invariant.

INVARIANT 1. *For any node n, let $P_n = \{v \in N, (v, n) \in E\}$ denote the set of* parents *of n. Then for any $n \in N_R \cap N_V$ there exists one node $v \in P_n$ such that $v \in N_R \cap N_V$.*

PROOF. Assume the contrary, let n be a node so that none of the nodes $v \in P_n$ are present in the result set. Then n cannot be reached from s using only valid nodes, since none of its parents are valid. □

THEOREM 1. *The algorithm in Figure 3 is correct.*

PROOF. By sorting the nodes in order of the topological sort, we can conclude that at the point node n is examined, all of its parents already have been examined by the algorithm. Node n can be added to the reachable set if and only if one of the nodes $v \in P_n$ was added to the result set. Therefore, n is added to the result set only if one of its parents is valid and reachable. □



n	N_V	N_R	$N_R \cap N_V$
$s = 0$	$\emptyset$	{1,4,5}	$\emptyset$
2	{2}	{1,4,5}	$\emptyset$
3	{2,3}	{1,4,5}	$\emptyset$
5	{2,3,5}	{1,4,5,6,7}	{5}
6	{2,3,5,6}	{1,4,5,6,7,8}	{5,6}
8	{2,3,5,6,8}	{1,4,5,6,7,8}	{5,6,8}

Figure 4: DAG example. The table shows the state of N_V, N_R and $N_R \cap N_V$ after each valid node is returned by the index.

3.1.1 Speeding up the DAG algorithm

We can speed up the DAG algorithm further by skipping in the underlying index. The following two lemmas show that we can always skip to the minimum element in the reachable set that is at least as big as the current node id returned by the index.

LEMMA 1. *Let m be the minimum node id in N_R. Then no node with id less than m can ever added to the result set.*

PROOF. Consider a node k whose id is less than m. Then when processing node k, we know that it is not in the reachable set and therefore the `reachable.contains(k)` statement will fail. □

LEMMA 2. *When processing node n, let m be the minimum id in N_R that is at least as big as n. Then no node with id less than m can ever be added to the result set.*

PROOF. Suppose by contradiction that some node with id less than m should be added to the result set, and let k be such a node with the smallest id. Clearly k must be a valid node, furthermore, one of its parents, $v \in P_k$ must be both valid and reachable. When processing v we add C_v to the reachable set. Therefore, since $k \in C_v$ it could not be skipped during the course of the algorithm. □

To implement skipping during retrieval we need to change only two lines in the original DAG algorithm (Figure 5). The changes from the previous algorithm are in line 2, where we set the next node to be retrieved by the index to be the minimum node id in the reachable set, and line 8, where we resume searching for valid nodes after the minimum node id from the reachable set that is greater than n.

Let us consider again the example from Figure 4 this time using the skip enabled algorithm shown in Figure 5. After the index returns node 2 and we verify that it is unreachable, we know that the next node with id greater than n that is in the reachable set is 4. Therefore we can avoid retrieving node 3 from the index completely (for $n = 2$, variable *skip* will be set to 4 in line 8 of the algorithm).

```
evaluate(s, Q)
  1. reachable.add(graph.children(s));
  2. skipIndex = min(reachable);
  3. while (n = index.getNextEntity(Q, skipIndex)) {
  4.    if (reachable.contains(n)) {
  5.       result.add(n);
  6.       reachable.add(children(n));
  7.    }
  8.    skipIndex = minMoreThan(reachable, n);
  9. }
 10.return result.nodes();
```

Figure 5: Query evaluation algorithm for DAGs with skipping.

3.2 General graphs

The crucial invariant in the case of DAGs ensured that when processing a node n all of its parents had already been processed. This allowed us to quickly decide whether n is reachable or not. This is not the case in general graphs, where no topological sort on the nodes exists. In this case, the evaluation algorithm explicitly maintains the *valid* set N_V, in addition to the set of reachable nodes N_R.

As before, the algorithm consumes the sequence of nodes returned by the index. Each of these nodes is valid, therefore when processing node n, the algorithm begins by adding it to the valid set N_V. If the node is already present in the reachable set ($n \in N_R$), it is then added to the result set. Moreover, since at this point we can conclude that n is valid and reachable we add all of its children to the reachable set. We recursively check whether any of these nodes were already valid, in which case they too are added to the result set and their children added to the reachable set.

The exact pseudocode for the algorithm for general graphs is presented in Figure 6. We begin by retrieving the valid nodes from the index starting from node id 0 (line 2). Once a node n is returned by the index, *evaluate* adds it to the valid set (line 4). It then checks if n is reachable (line 5). If n belongs to the reachable set we know it is both reachable and valid and we use the auxiliary function *updatePath* to update the status of n and its descendant nodes.

Function *updatePath* starts by adding n to the result set (line 1). Then it updates the status of n's children. At this point of the execution we can conclude that n's children have at least one valid path leading to them. This is done in lines 2–12. We only modify the status of a child node c if it is not already in the result set (line 4). This check guarantees that *updatePath* is called exactly once for each node in the result set. If c already belongs to the valid set, we know it is both valid and reachable and we update its status through a recursive call to *updatePath* (line 6). If c does not belong to the valid set, we just add it to the reachable set (line 9).

Let us consider the example in Figure 7. For this example we randomly assigned node ids to emphasize the fact that the algorithm does not make any assumption about the node id ordering. The start node s is 3 and the query labels are $Q = \{A, B, C\}$. A node is considered valid if its labels have a non-zero intersection with Q. The set of valid nodes N_V returned by the index is {1,2,5,6,8}. The table in Figure 7 shows the initial state of each of the node sets, as well as the state after each call to the index method *getNextEntity*().

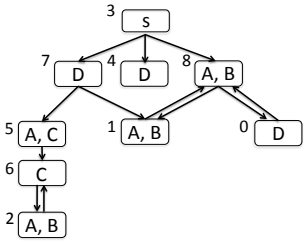
When nodes 1, 2, 5 and 6 are returned by the index they are not in the reachable set, so we simply add them to the valid set. When the index returns node 8, which is reachable,

```
evaluate(s, Q)
  // Returns the valid and reachable nodes.
  1. reachable.add(graph.children(s));
  2. skipIndex = 0;
  3. while (n = index.getNextEntity(Q, skipIndex)) {
  4.    valid.add(n);
  5.    if (reachable.contains(n)) {
  6.       updatePath(n);
  7.    }
  8.    skipIndex = n + 1;
  9. }
 10. return result.nodes();

updatePath(n)
  // Updates status of a node and its descendants.
  1. result.add(n);
  2. C = graph.children(n);
  3. foreach c in C {
  4.    if (not result.contains(c)) {
  5.       if (valid.contains(c)) {
  6.          updatePath(c);
  7.       }
  8.       else {
  9.          reachable.add(c);
 10.       }
 11.   }
 12.}
```

Figure 6: Query evaluation algorithm for the general case.

we add it to the valid set and we call *updatePath*, which adds 8 to the result set and its children 0 and 1 to the reachable set. Since node 1 is already valid, *updatePath* is called recursively and it is added to the result set as well.



n	N_V	N_R	*Result*
$s = 3$	$\emptyset$	{4,7,8}	$\emptyset$
1	{1}	{4,7,8}	$\emptyset$
2	{1,2}	{4,7,8}	$\emptyset$
5	{1,2,5}	{4,7,8}	$\emptyset$
6	{1,2,5,6}	{4,7,8}	$\emptyset$
8	{1,2,5,6,8}	{0,1,4,7,8}	{1,8}

Figure 7: Cyclic graph example. The table shows the state of N_V, N_R and $N_R \cap N_V$ after each valid node is returned by the index.

LEMMA 3. *The query evaluation algorithm returns node n in a result if and only if n is valid and reachable.*

PROOF. For n to be added to the result set, it must be returned by the index and therefore valid. Furthermore, since only the children of result nodes are added to the set of reachable nodes N_R, one of its parents was a result node, therefore n must be reachable as well.

To prove the converse, assume by contradiction that the lemma is false and let V be the set of valid and reachable nodes that is not returned by the algorithm. There exists

some node $n \in V$ such that one of its parents $v \in P_n$ must be returned by the algorithm (otherwise none of the nodes in V can be reached from s). If v was added to the result set before processing n, then when processing n it will appear in N_R and therefore be added to the result set. Otherwise, n is added to the valid set N_V, however when v is added to the result set, n will be marked reachable and added to the result set as well. Therefore no such n can exist. □

3.2.1 Speeding up the algorithm

In the case of DAGs the numbering of the nodes allowed us to conclude that some of the valid nodes cannot be reachable, and skip in the underlying index. At first glance, this is not true in the case of general graphs—without a full ordering on the nodes, one cannot skip a node simply because it is not currently in the reachable set. In order to maintain the skipping property, we first decompose the graph into strongly connected components (SCCs). Recall that if we contract each SCC into a single node, the resulting graph, called the condensation of G, is a DAG. Thus, we can combine the skipping component from the DAG algorithm and the recursive evaluation component from the general algorithm to enable skipping.

Before building the index we decompose the graph into the SCCs. Let us assume that node ids have two parts, the SCC id and the id of the node within the SCC. After decomposing the graph into SCCs, we assign the SCC ids in topological sort order. Inside each SCC we assign ids in arbitrary order. Figure 9 shows an example of this id assignment. Given two node ids $c_1.n_1$ and $c_2.n_2$, $c_1.n_1 > c_2.n_2 \Leftrightarrow c_1 > c_2 \vee (c_1 = c_2 \wedge n_1 > n_2)$. If required by the index API, this numbering scheme can be easily converted to simple integer ids, e.g. by using the most significative bits to represent the SCC id. We proceed by running the DAG algorithm on the SCC ids and the general graph algorithm on the combined ids. The former enables us to skip over unreachable entries in the graph, while the latter guarantees correctness within each SCC.

The full algorithm is given in Figure 8. We use variable *reachableSCCs* to store just the component ids from the nodes in N_R. The main changes from Figure 6's algorithm are in lines 6 and 16, where we set variable *skip* to the minimum SCC id in the reachable set. In line 16 we also make sure the component is greater than the current component, denoted by *scc*. For simplicity we assume that setting *skip* to a given component *comp* will cause the index to return the next valid node with id greater than *comp*.0. Another change is that we only add a node to the valid set if it belongs to a reachable component (line 8).

In Figure 9 we show a run of the *evaluate* algorithm with skipping enabled. We keep the same example as in Figure 7, but annotate the graph with new node id assignment scheme. The algorithm proceeds as before, keeping a set of valid and reachable nodes, as well as the reachable SCCs. When evaluating node 2.1 we note that the minimum reachable SCC has index 4, therefore we can set *skip* to 4.0. This allows us to completely skip over nodes 3.1 and 3.2, which would otherwise be retrieved by the index. Another subtle point is that although node 2.1 is valid, we do not add it to the valid set N_V since at the point that it is processed we already know it is not reachable.

To reason about the skipping behavior, we observe the following simple consequence of the labeling scheme.

```
evaluate(s, Q)
  // Returns the valid and reachable nodes.
  1. C = graph.children(s);
  2. foreach scc.v in C {
  3.    reachable.add(scc.v);
  4.    reachableSCCs.add(scc);
  5. }
  6. skip = min(reachableSCCs);
  7. while (scc.n = index.getNextEntity(Q, skip)) {
  8.    if (reachableSCCs.contains(scc)) {
  9.       valid.add(scc.n);
 10.       if (reachable.contains(scc.n)) {
 11.          updatePath(scc.n);
 12.       }
 13.       skip = scc.n + 1;
 14.    }
 15.    else {
 16.       skip = minMoreThan(reachableSCCs, scc);
 17.    }
 18. }
 19. return result.nodes();

updatePath(scc.n)
  // Updates status of a node and its descendants.
  1. result.add(scc.n);
  2. C = graph.children(scc.n);
  3. foreach comp.v in C {
  4.    if (not result.contains(comp.v)) {
  5.       reachableSCCs.add(comp);
  6.       if (valid.contains(comp.v)) {
  7.          updatePath(comp.v);
  8.       }
  9.       else {
 10.          reachable.add(comp.v);
 11.       }
 12.    }
 13. }
```

Figure 8: Query evaluation algorithm for the general case with skipping.

INVARIANT 2. *For any two nodes $v, w \in N$ if there exists a path from v to w in G, then either v and w lie in the same SCC, or the SCC id of v is strictly smaller than the SCC id of w.*

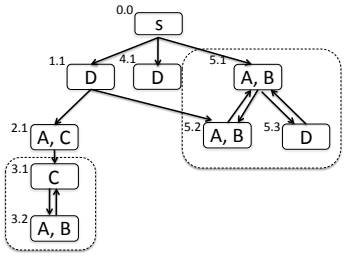
The invariant allows us to skip unreachable SCCs in the general graph in the same manner that we skipped unreachable nodes in DAGs. To ensure correctness we state the analogues of Lemmas 1 and 2. We omit their proofs since they are parallel to those in the DAG case.

LEMMA 4. *Let $c_m.n_m$ be the minimum node id in N_R. Then no node with id less than $c_m.0$ can ever added to the result set.*

LEMMA 5. *When processing node c.n, let $c_m.n_m$ be the minimum id in N_R that is at least as big as c.n. Then no node with id less than $c_m.0$ can ever be added to the result set.*

3.2.2 Node ordering within an SCC

Although the node id assignment within an SCC does not impact the algorithm correctness or the ability to skip in the index, it may affect query latency. In particular, it is easy to come up with examples where a suboptimal labeling may delay the emission of the first result by $O(n)$, where n is the number of nodes in the graph. The time for retrieving the first result is quite important in several applications,



scc.n	N_V	N_R	*SCCs*	*Result*
$s = 0.0$	$\emptyset$	{1.1,4.1,5.1}	{1,4,5}	$\emptyset$
2.1	$\emptyset$	{1.1,4.1,5.1}	{1,4,5}	$\emptyset$
5.1	{5.1}	{1.1,4.1,5.1,5.2,5.3}	{1,4,5}	{5.1}
5.2	{5.1,5.2}	{1.1,4.1,5.1,5.2,5.3}	{1,4,5}	{5.1,5.2}

Figure 9: Graph decomposed into strongly connected components (SCCs). Column SCCs is the set of reachable SCCs. After processing node 2.1 the next reachable SCC is 4, therefore the algorithm sets *skip* to 4.0 and nodes 3.1 and 3.2 are skipped during the processing.

including Web advertising exchanges, where the selection of valid subscribers is followed by other computations that can be pipelined [17].

In Figure 10(a) we show an example where we have m nodes in a single SCC. Let us consider for this example that the query labels include label A but not label X, which means that all of the nodes are valid except for node 1.1. When node 1.2 is returned by the index it is simply added to the valid set N_V. The same happens for every node $1.i$, i between 2 and $m-1$. Finally, when we see node $1.m$, we know it is reachable and we add it to the result set. At that point, *updatePath* is recursively called to add all nodes $1.i$, i between $m-1$ and 2 to the reachable and result sets. This means that we must retrieve $m-1$ nodes from the index before emitting the first query result.

Figure 10(b) shows exactly the same graph, but with different node ids. With these new ids no recursive call to *updatePath* is needed. Every node $1.i$, i from 1 to $m-1$ returned by the index is already reachable by the time it is evaluated. This means that we can start emitting results right away when we retrieve the first result from the index.

We note that for every fixed query a node assignment requiring no calls to *updatePath* always exists: simply label the nodes in order discovered by running breadth-first search from s. However, there is no universally optimal assignment — different queries yield different optimum assignments.

3.3 Handling updates in the system

Our proposed algorithm relies on the index for evaluating the targeting constraints, and on the graph, for checking node reachability. These two data structures are built offline and used during query processing. Both the graph and the index structure can be updated using standard techniques (for example maintaining a "tail" index for entities added since the last index build).

We must be careful, however, if the new updates to the graph change the global connectivity parameters. Problems arise if, due to the update, two previously distinct SCCs are now merged into one, as the skipping may now produce incorrect results. In this case, we must disable the SCC

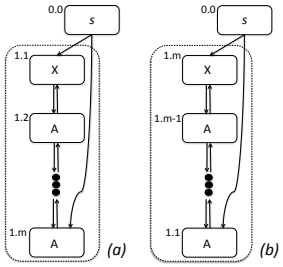


Figure 10: Two different node id assignments within an SCC and their impact on query latency: (a) illustrates the worst possible assignment while (b) illustrates the best possible assignment.

based skipping until the graph index is rebuilt and the node ids are updated to reflect the new graph structure.

3.4 Complexity analysis

The challenge of the problem we are trying to solve comes from the fact that even though the underlying graph G is given ahead of time, the graph induced by the valid nodes is given online, as it depends on the query Q. Before analyzing the running time of our proposed algorithm, we present a running time lower bound that applies to any algorithm for this problem. In what follows, denote by out_n the number of outgoing edges for node n. For any set of nodes S, $\text{out}(S) = \sum_{n \in S} out_n$.

THEOREM 2 (LOWER BOUND). *The worst-case running time lower bound for any algorithm $\mathcal{A}$ solving the networked pub/sub problem is $\Omega(|N| + |E|)$, where $|N|$ and $|E|$ are the number of nodes and edges in the system, respectively.*

PROOF. Consider a graph G where all nodes are valid and reachable and each node has only one incoming edge. Any algorithm $\mathcal{A}$ for the networked pub/sub problem will have to look at all nodes and edges of this graph in order to produce a valid response. □

Since the algorithm makes a check for every node returned by the index, the total running time of our algorithm is $\mathcal{O}(|N_V| + \text{out}(N_V \cap N_R))$. However, it is worth noting that the skipping allowed by the structure of the graph reduces the contribution of the N_V term as some valid nodes may be determined to be unreachable and be skipped by the index.

Besides the proposed algorithms, in Section 4 we show results for two baseline algorithms:

- *Index baseline* uses the index to retrieve all valid nodes and then applies a breath-first search (BFS) on the results of the index, to filter out nodes that are not reachable. The running time for this algorithm is also $\mathcal{O}(|N_V| + \text{out}(N_R \cap N_V))$. However, for this algorithm, we have to wait until the index returns all the results before the BFS can start, and therefore, the query latency for the first response is much higher than in our proposed algorithm, which can start emitting results during the index evaluation. Moreover, this algorithm cannot benefit from the graph structure to drive skipping in the index.
- *BFS baseline* does not use the index. It runs BFS from the start node s, and for each reachable node n

it calls $match(\cdot,\cdot)$ to validate if n is also valid. The running time for this algorithm is $\mathcal{O}(|N_R| + \text{out}(N_R \cap N_V))$. The downside of this baseline is the fact that evaluating $match(\cdot,\cdot)$ without an index does not scale well in practice, as it is shown in our experimental results.

Our proposed algorithm and the two baselines are worst-case optimal. Moreover, the number of edges accessed by our algorithm and the two baseline algorithms is also the same, as described below.

THEOREM 3. *Let $E_{BFS} \subset E$ be the set of edges examined by any BFS algorithm running on graph G. Then the number of edges examined by the evaluate algorithm is no more than $|E_{BFS}|$.*

PROOF. Consider the edges examined by *evaluate*. Each edge is examined at most once, and edges are only examined when a node v is determined to be both valid and reachable. Therefore, the number of edges examined by *evaluate* is at most $\text{out}(N_V \cap N_R)$. Now consider a BFS algorithm running on G. Every node added to the BFS queue is valid and reachable, and every time such a node is evaluated all of its children are added to the queue. Therefore, the number of edges examined by the BFS algorithm is $\text{out}(N_V \cap N_R)$. □

4. EXPERIMENTAL RESULTS

In this section we evaluate our query evaluation algorithms. We start by describing the data set we used in Section 4.1. We then evaluate the performance of the general version of our algorithm against the BFS and index baselines defined in Section 3.4. For this evaluation we varied the targeting selectivity (Section 4.2) and graph size (Section 4.3). We run our experiments on a 2.5GHz Intel(R) Xeon(R) processor with 16GB of RAM. In all experiments we run, both the index and the graph were already loaded into memory.

4.1 Data set

Our experiments are based on a subset of the graph from Yahoo!'s RightMedia advertising exchange currently in production. This graph has three types of nodes: *publishers* which are nodes with no incoming edges; *ad networks*, which are the intermediary nodes; and *advertisers*, which are nodes with no outgoing edges. The graph has 71,097 nodes and 87,799 edges. A summary of the the salient statistics of the data set is given in Table 1.

The average number of nodes reachable from each publisher node ignoring the targeting constraints is 4,802, which is about 43% from the total of 11,086 ad networks and advertiser nodes. We have also computed the strongly connected components (SCCs) and the average number of incoming and outgoing edges for each type of node in the graph. From the 282 ad network nodes, 126 of them form a large SCC. The remainder networks are isolated, except for another small SCC of size 4. Figure 11 shows the structure of the graph, which resembles the bow-tie structure of the Web graph [4]. As shown in the figure, there are about 17K edges that directly connect publishers to advertisers, without going through ad networks. We also computed the number of cycles in the graph with path size smaller than 12 and found over 3.7 million cycles.

In our setting, an event is the result of a user visiting one of the pages for a given publisher. The event contains a set

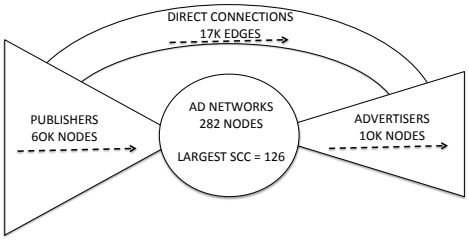


Figure 11: Graph structure of an advertising exchange.

PARAMETER	VALUE
NUMBER OF NODES	71,097
NUMBER OF EDGES	87,799
PUBLISHERS (PUB)	60,011
AD NETWORKS (NET)	282
ADVERTISERS (ADV)	10,804
AVG REACHABLE NODES	4,802
MAX REACHABLE NODES	5,998
MIN REACHABLE NODES	1
NUMBER OF SCCs	70,969
MAX SCC SIZE	126
AVG out(PUB)	1.3
AVG in(NET)	224.4
AVG out(NET)	42.1
AVG in(ADV)	2.3
NUMBER OF CYCLES	> 3,776,185

Table 1: Summary of graph parameters.

of attributes from the page and the user. A simple example is $\{topic \leftarrow sports, gender \leftarrow male, age \leftarrow 20, adsize \leftarrow 300 \times 200\}$. Each ad network and advertiser node in the system can define the types of events they are interested on by specifying a Boolean expression over the event attributes. For instance, they can specify $(topic = finance \wedge age = 20 \wedge geo = CA)$. We used the index algorithms described in [12, 22], which efficiently evaluate arbitrarily complex Boolean expressions using an inverted index.

4.2 Index selectivity

For the first set of experiments, we used the graph described in Table 1 and Figure 11 and we varied the index selectivity. We defined six attributes with different selectivity values and we randomly assigned DNF expressions over these synthetic attributes to the graph nodes, so that the index selectivity would be approximately $1/16$, $1/8$, $1/4$, $1/2$ and 1. Selectivity 1 means that no targeting was applied. Table 2 shows the size of the node sets and the number of edges evaluated by the algorithms for each index selectivity we tested. We tested 1,000 queries for each selectivity and the numbers are averaged.

Figure 12 compares the running time of our algorithm with the BFS baseline, which does not use the index and simply applies $match(\cdot,\cdot)$ for every reachable node during the BFS. Since the graph from the advertising exchange has cycles, for these experiments we used the generic version of *evaluate* described in Figure 6. The times shown in this and all figures in this section are in milliseconds. The figure also shows the time spent by the baseline in the $match(\cdot,\cdot)$ evaluation. This baseline performs well if $|N_R|$ is small, but it is not scalable. In our setting evaluating $match(\cdot,\cdot)$ takes 20μs without the index. Therefore, after $|N_R|$ is only 2,000 (selectivity is $1/2$) the index-based methods are more efficient.

	1/16	1/8	1/4	1/2	1
$\|N_V\|$	693	1,336	3,081	7,064	11,086
$\|N_R\|$	34	83	452	1,756	4,837
$\|N_R \cap N_V\|$	3	11	125	1,126	4,837
out($N_R \cap N_V$)	47	127	740	3,310	8,588

Table 2: Size of valid, reachable and result node sets and the number of edges evaluated for different selectivity values.

More complex evaluation functions which require more time to evaluate, would lead to a index based methods outperforming the BFS baseline even earlier. For the remainder of the experiments we do not show results for the BFS baseline since it is not scalable.

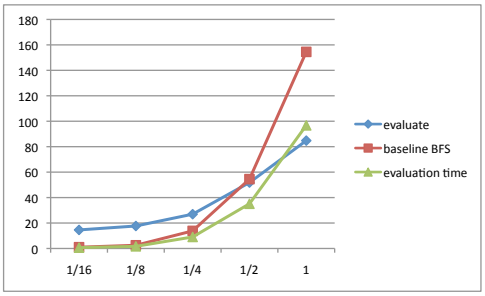


Figure 12: Runtime performance of our proposed algorithm and the BFS baseline for different selectivity values. Also shown is the cost of evaluation without the index.

Figure 13 compares our algorithm with the index baseline, which first takes all of the valid nodes from the index and then runs BFS to find the reachable subset. The figure also shows the total time spent in the index evaluation module, i.e., the time spent in calls to *getNextEntity*(). Figure 14 measures query latency by plotting the time each different algorithm takes for emitting the first result. Our proposed algorithm is always about 50% faster than the index baseline due to the fact that it can start emitting results before retrieving all of the valid nodes. In Web advertising exchanges, the time to retrieve the first result is quite important as there are several steps in the overall evaluation that can be pipelined [17].

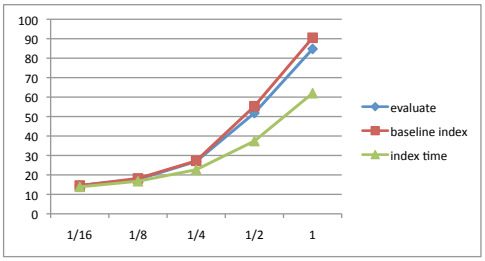


Figure 13: Runtime performance of our proposed algorithm and the index baseline for different selectivity values. Also shows index evaluation cost.

4.3 Graph size

For the experiments presented in this section we used a graph generator to produce graphs of different sizes. We generated graphs up to five times bigger than our original graph maintaining the same average number of incoming and outgoing edges for publishers, ad networks, and advertiser

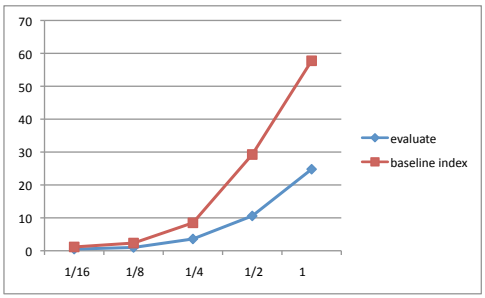


Figure 14: Time to first response of our proposed algorithm and the index baseline for different selectivity values.

	×1	×2	×3	×4	×5
$\|N_V\|$	2,751	5,489	8,263	10,968	13,677
$\|N_R\|$	596	1,153	1,675	2,141	2,806
$\|N_R \cap N_V\|$	166	320	466	593	782
out($N_R \cap N_V$)	1,000	1,934	2,833	3,613	4,746

Table 3: Size of valid, reachable and result node sets and the number of edges evaluated for different graph sizes.

nodes as shown in Table 1. For these experiments we kept the targeting selectivity to 1/4 and we tested 1,000 random queries for each graph size. Table 3 shows the average size of the different node sets and number of edges evaluated for each graph size.

Figure 15 compares the running time of our algorithm with the index baseline for different graph sizes. Figure 16 shows the time for the first response, highlighting that the latency of the proposed algorithm scales better than the baseline as the graph size grows.

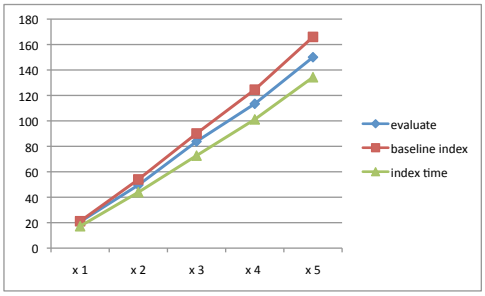


Figure 15: Runtime performance of our proposed algorithm and the index baseline for different graph sizes. Also shows index evaluation cost.

5. RELATED WORK

As mentioned in the introduction, the problem of evaluating graph constraints extends the semantics of matches in content-based publish/subscribe systems. Our proposed solution builds upon the work on indexing content-based subscriptions, including the use of hash-indices (e.g., [11]), trees (e.g., [2]) and inverted lists (e.g., [22]), and extends them to work on graph constraints. There has also been a large body of work on pub/sub systems with complex predicates and events, such as those that support predicates over multiple events (e.g., [1, 8]) and tree-structured events (e.g., [9]), which is again complementary to our proposed ap-

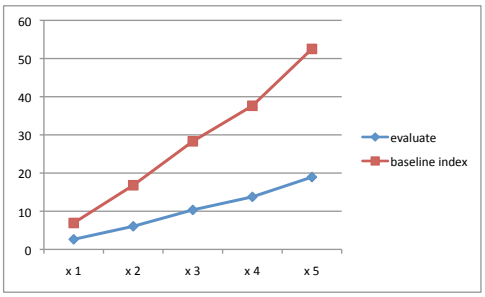


Figure 16: Time to first response for different sizes.

proach since we focus on evaluating graph constraints. Another related body of work is that of content-dissemination networks (e.g., [5, 20, 21], where the goal is to efficiently route the results to subscribers over a physical network based on matching results. These do not change the semantics of matches themselves, as is the case with graph constraints.

There has also been a vast body of work on indexing graphs in order to efficiently support reachability queries (e.g., [7, 13, 15, 19]). These techniques are designed to handle queries such as "is node A reachable from node B" given the directed edges of the graph. Using our terminology, existing reachability techniques essentially support only the case where function $match(\cdot, \cdot)$ evaluates to "true" for every node. They are not designed to handle predicates on the nodes of the graph which depend on the query assignment, which is the main focus of the techniques we propose.

More recently, [14] showed how to handle the case when the $match(\cdot, \cdot)$ function is the subset relation. Their algorithms are exponential in Σ, the total number of unique labels on the edges. In this paper we consider arbitrary $match(\cdot, \cdot)$ functions, which may encode Boolean expressions over hundreds of potential labels, as long as they are efficiently supported by the index infrastructure (e.g. [12]).

Another related body of work is that of indexing the structure and content of XML documents, and efficiently processing queries over such documents (e.g., [16]). While XML queries can indeed specify restrictions on valid paths over XML documents, these are designed to work over tree-structured data, and the set of predicates are limited to simple predicates on XML tags, as opposed to arbitrary predicates over nodes and edges.

6. CONCLUSIONS AND FUTURE WORK

In this paper, we introduced the problem of evaluating graph constraints in content-based pub/sub systems, and have proposed efficient evaluation algorithms over arbitrary directed graphs. Our experimental results show that the proposed algorithms lead to significant performance gains in a realistic Web advertising exchange application scenario. The proposed algorithms are currently deployed in production at the core of Yahoo!'s RightMedia Web advertising exchange. An interesting direction for future work is a ranked version of this problem, in which the ranking function may be dependent on the paths from the publisher to the different subscribers (e.g., longer paths may incur in higher cost, so subscribers reached through them might have lower scores).

7. REFERENCES

[1] J. Agrawal, Y. Diao, D. Gyllstrom, and N. Immerman. Efficient pattern matching over event streams. In *SIGMOD*, 2008.

[2] 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*, 1999.

[3] G. Banavar, T. D. Chandra, B. Mukherjee, J. Nagarajarao, R. E. Strom, and D. C. Sturman. An efficient multicast protocol for content-based publish-subscribe systems. In *ICDCS*, 1999.

[4] A. Z. Broder, R. Kumar, F. Maghoul, P. Raghavan, S. Rajagopalan, R. Stata, A. Tomkins, and J. L. Wiener. Graph structure in the web. *Computer Networks*, 33(1-6):309–320, 2000.

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

[6] B. Chandramouli, J. Yang, P. K. Agarwal, A. Yu, and Y. Zheng. Prosem: scalable wide-area publish/subscribe. In *SIGMOD*, 2008.

[7] E. Cohen, E. Halperin, H. Kaplan, and U. Zwick. Reachability and distance queries via 2-hop labels. *SIAM J. Comput.*, 32(5), 2003.

[8] A. J. Demers, J. Gehrke, M. Hong, M. Riedewald, and W. M. White. Towards expressive publish/subscribe systems. In *EDBT*, 2006.

[9] Y. Diao, M. Altinel, M. J. Franklin, H. Zhang, and P. M. Fischer. Path sharing and predicate evaluation for high-performance xml filtering. *ACM TODS*, 28(4), 2003.

[10] Y. Diao, S. Rizvi, and M. J. Franklin. Towards an internet-scale xml dissemination service. In *VLDB*, 2004.

[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*, 2001.

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

[13] H. Hwang, V. Hristidis, and Y. Papakonstantinou. Objectrank: a system for authority-based search on databases. In *SIGMOD*, 2006.

[14] R. Jin, H. Hong, H. Wang, N. Ruan, and Y. Xiang. Computing label-constraint reachability in graph databases. In *SIGMOD*, pages 123–134, New York, NY, USA, 2010. ACM.

[15] R. Jin, Y. Xiang, N. Ruan, and D. Fuhry. 3-hop: a high-compression indexing scheme for reachability query. In *SIGMOD*, 2009.

[16] R. Kaushik, R. Krishnamurthy, J. F. Naughton, and R. Ramakrishnan. On the integration of structure indexes and inverted lists. In *ICDE*, page 829, 2004.

[17] K. Lang, B. Ghosh, J. Delgado, D. Jiang, S. Das, A. Gajewar, S. Jagadish, A. Seshan, M. Binderberger-Ortega, C. Botev, S. Nagaraj, and R. Stata. Efficient online ad serving in a display advertising exchange. In *WSDM*, 2011.

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

[19] R. Schenkel, A. Theobald, and G. Weikum. Efficient creation and incremental maintenance of the hopi index for complex xml document collections. In *ICDE*, pages 360–371, 2005.

[20] S. Shah, S. Dharmarajan, and K. Ramamritham. An efficient and resilient approach to filtering and disseminating streaming data. In *VLDB*, 2003.

[21] A. C. Snoeren, K. Conley, and D. K. Gifford. Mesh based content routing using xml. In *SOSP*, 2001.

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