Delta: Scalable Data Dissemination under Capacity Constraints *

Konstantinos Karanasos
IBM Almaden Research Center
San Jose, CA, USA
kkarana@us.ibm.com

Asterios Katsifodimos
Inria Saclay and Université Paris-Sud
Orsay, France
asterios.katsifodimos@inria.fr

Ioana Manolescu
Inria Saclay and Université Paris-Sud
Orsay, France
ioana.manolescu@inria.fr

ABSTRACT

In content-based publish-subscribe (pub/sub) systems, users express their interests as queries over a stream of publications. Scaling up content-based pub/sub to very large numbers of subscriptions is challenging: users are interested in low *latency*, that is, getting subscription results fast, while the pub/sub system provider is mostly interested in *scaling*, i.e., being able to serve large numbers of subscribers, with low *computational resources utilization*.

We present a novel approach for scalable content-based pub/sub in the presence of constraints on the available CPU and network resources, implemented within our pub/sub system Delta. We achieve scalability by off-loading some subscriptions from the pub/sub server, and leveraging view-based query rewriting to feed these subscriptions from the data accumulated in others[1]. Our main contribution is a novel algorithm for organizing views in a multi-level dissemination network, exploiting view-based rewriting and powerful linear programming capabilities to scale to many views, respect capacity constraints, and minimize latency. The efficiency and effectiveness of our algorithm are confirmed through extensive experiments and a large deployment in a WAN.

1. INTRODUCTION

Publish/subscribe (*pub/sub*, in short) is a popular model for disseminating content to large numbers of distributed subscribers. The literature distinguishes *topic-based* pub/sub, where users subscribe to a set of predefined topics, from *content-based* pub/sub, where users express their subscriptions as custom complex-structured queries on the published data. Topic-based pub/sub offer better scalability at the expense of subscription expressiveness, while in more complex systems, the increased expressive power of content-based pub/sub makes it preferable. For instance, within a large company ACME, *"senior positions representing ACME in Singapore"* should be pushed to the senior staff which may be interested, while *"sales seminar in Singapore"* interests the sales department plus the administrative staff that must make the travel arrangements.

Pub/sub *subscribers* are interested in *low latency*, that is, getting all the results to their subscriptions, as soon as possible after the data is published. The *publisher* of a pub/sub system faces several performance challenges in order to meet subscriber requirements. The first is matching published items against the set of subscriptions, a CPU-intensive task. Then, the publisher's outgoing bandwidth is another physical limitation, as more and more updates must be sent to the interested subscribers. Third, the speed of the network connecting the publisher to the subscribers imposes a lower bound on the dissemination latency.

Both centralized and distributed approaches have been proposed to address the above issues, while aiming at latency minimization. The centralized ones [4, 7] mostly rely on efficient filtering algorithms for matching the data against subscriptions. However, for more expressive and numerous subscriptions, subscription matching remains an onerous task. To this end, distributed pub/sub systems have been proposed [8, 10, 20, 26], providing solutions for serving thousands or millions of subscribers with minimum resources utilization and low latency. In most cases, they focus on distributed filtering and design overlay networks in the form of logical multicast trees. Those trees are formed by specialized nodes, called *brokers*, able to efficiently filter and move the data from the publisher to the subscribers, or by the subscribers themselves. Nevertheless, as the amount of subscribers and data increases, the publisher's (or broker's) resource capacity becomes insufficient.

Problem statement. To overcome the above resource constraints, we allow the subscribers to *take part in the dissemination of data* (i.e. serve other subscribers that have similar interests) in order to offload the data publisher. Due to their similarity of interests, the subscribers can form a logical overlay network, over which subscription results can flow from the data publisher to the subscribers. Since subscribers have to use their resources to serve others, the problem we consider is how to (i) minimize the total resource utilization (e.g., CPU and bandwidth), while (ii) keeping the subscription latency as low as possible, and (iii) respecting the given resource capacity constraints.

The key idea on which we build our approach is that subscriptions often overlap, completely or partially, when user interests are close. In such a case, results of several subscriptions can be combined to compute the results of other subscriptions. For instance, from the subscriptions s_1: *"open positions in Asia"* and s_2: *"open positions in Sales"*, one can compute s_3: *"open Sales positions in Asia"* by joining s_1 and s_2.

*This work has started while the first author was in Inria Saclay, France. It has been partially funded by *Agence Nationale de la Recherche*, decision ANR-08-DEFIS-004 and EIT ICTLabs Europa activity (CLD12115).

[1]This can be seen as organizing subscriptions in a dissemination network where data flows from the source through a network of subscriptions, similarly to water flow in a river delta.

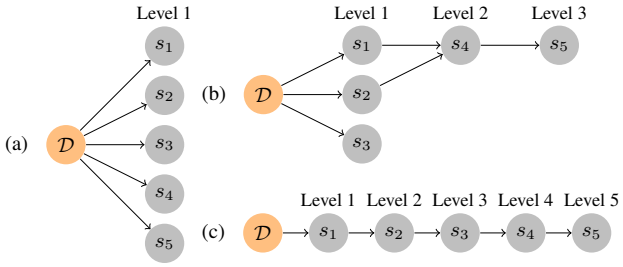


Figure 1: Sample dissemination networks.

Rewriting subscriptions. More formally, a subscription can be *rewritten based on other subscriptions*, by filtering their results, e.g., through classic database selections and projections, combining them through joins, etc. For instance, rewriting and serving s_3 based on s_1 and s_2 instead of the publisher, relieves the publisher from the effort of computing s_3 against the published data, and saves bandwidth between the publisher and the site of s_3. At the same time, rewriting s_3 from s_1 and s_2 incurs computations to the sites of s_1, s_2 and/or s_3 to evaluate the rewriting, and also bandwidth consumption from the sites of s_1 and s_2, to the site of s_3. Notice that if we consider subscriptions as queries (or views), deciding how to serve a subscription based on others, can be turned to a problem of *view-based query rewriting*, which has been extensively studied in the database literature (e.g. [21, 18]).

Multi-level subscriptions. Moving a subscription from being served directly by the publisher (we call this a *level 1* subscription), to being served from other subscriptions by rewriting (we call this a *level 2*, *level 3* subscription, etc.), changes the data transfer and processing paths, with many possible consequences on subscription latency and resources utilization for data dissemination.

For illustration, Figure 1 shows three possible dissemination networks. At left (a), there is only one level and all subscriptions are filled from the publisher $\mathcal{D}$. The data paths from $\mathcal{D}$ to all subscriptions are as short as possible, however all the load is on $\mathcal{D}$. At (b), the subscription s_5 gets its data from s_4 instead of the publisher, while s_4 results are computed based on s_1 and s_2. At (c), only s_1 is filled from $\mathcal{D}$, while s_2 gets data from s_1, s_3 from s_2, etc. The load on the publisher is minimal, but the four hops from $\mathcal{D}$ to s_5, increase the latency of this subscription.

More generally, dissemination effort decreases at the publisher, at the expense of subscribers joining this effort. A less-loaded publisher will likely match data against the rest of the subscriptions faster, which may reduce the total latency for all the subscriptions. However, moving a subscription to a higher level lengthens the data path from the publisher to that subscription, which may increase its latency. Finally, pushing some processing at the subscribers require taking into account a new set of capacity constraints, since subscriber resources should be sparingly used, to keep the respective sites willing to participate in the system.

Contributions and outline. Given a set S of subscriptions and a data publisher $\mathcal{D}$, we term *configuration* a choice for each subscription $s \in S$ of filling s either (i) directly from $\mathcal{D}$ or (ii) by rewriting s over some other S subscriptions and thus computing s results from these other subscriptions' results. The *cost* of a configuration is a weighted sum of the resource utilization and subscription latencies incurred by the configuration. This work makes the following contributions:

- We show how to model the problem of finding a minimum-cost configuration under some resource capacity constraints as a graph problem, related to the known Degree-bounded Arborescence problem [1], but departing from it through our interest in minimizing both resource utilization and latency. As we will explain, resource utilization and latency differ in fundamental ways, making existing solutions inapplicable in our setting.
- Based on this insight, we provide a novel two-step algorithm for selecting a configuration. First, we employ an Integer Linear Programming (ILP) approach to find a resource utilization-optimal solution (ignoring latency); second, we provide a latency optimization algorithm which starts from the configuration found by the ILP solver and modifies it to reduce latency.
- We have implemented all our algorithms and performed extensive experiments, including a deployment of Delta on a significant-size pub/sub scenario on a WAN. Our experiments demonstrate the efficiency and effectiveness of our algorithms and the practical interest of multi-level subscriptions in large data dissemination networks.

The paper is organized as follows. Section 2 introduces our problem and presents its graph-based formalization. Section 3 describes our algorithms for selecting an efficient configuration, based on the graph models previously introduced. Section 4 details our view-based approach for rewriting subscriptions based on other subscriptions, given the large number of subscribers. Section 5 describes our experiments, we then discuss related works and conclude.

2. PROBLEM MODEL

We now describe our multi-level subscription problem model.

Let $\mathcal{D}$ denote a data source publishing a set of data items $i_1, i_2, \ldots$ and $S = \{s_1, s_2, \ldots, s_n\}$ be a finite set of subscriptions, each defined by a query and established on some network site. The semantics of a subscription s defined by query q_s and issued at site n_s is that s must receive the results of $q_s(i)$ for any data item i published by the data source $\mathcal{D}$ after s was created. From now on, for simplicity, whenever possible we will simply use s to denote both a subscription and the query defining it.

At the core of our work is the observation that it may be possible to compute results of a subscription out of the results of others. We say subscription s can be **rewritten** based on subscriptions $s_1, s_2, \ldots, s_k$, if there exists a query r, which, evaluated over the results of $s_1, s_2, \ldots, s_k$, produces exactly the results of subscription s, regardless of the actual data items published by $\mathcal{D}$: $r(s_1(\mathcal{D}), s_2(\mathcal{D}), \ldots, s_k(\mathcal{D})) = s(\mathcal{D})$ for any $\mathcal{D}$, or more simply, $r(s_1, s_2, \ldots, s_k) \equiv s$, where $\equiv$ denotes query equivalence.

Subscriptions = views. Observe that we are interested in complete rewritings only, that is, we do not assume that r can rely directly on the data source, but only on the results of subscriptions $s_1, s_2, \ldots, s_k$. This is because our goal is to off-load subscriptions from the data source and serve them from other subscriptions instead. In turn, a subscription s rewritten based on $s_1, \ldots, s_k$ as above, may be used to rewrite another subscription s'. This shows that every subscription may be considered as a (materialized) view, based on which to rewrite the others. Thus, from now on, for conciseness, we will simply use *view* to designate a subscription.

In the sequel, we introduce the central concepts and data structures of our work. We define rewritability graphs (RGs) and configurations in Section 2.1. Section 2.2 presents the basic metrics we use to gauge the interest of a configuration, namely utilization and latency, and shows how to incorporate load balancing in the discussion under the form of constraints over the configurations. Based on these notions, Section 2.3 formalizes our problem statement.

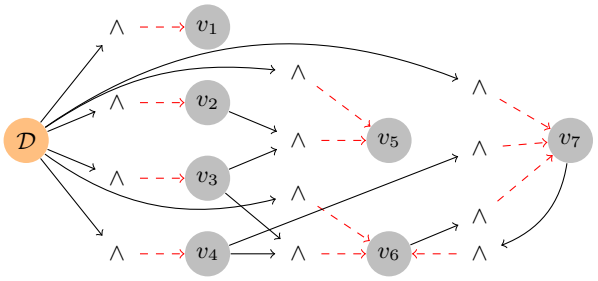


Figure 2: Rewritability Graph (RG).

2.1 Rewritability Graph (RG)

A rewritability graph (RG) indicates which views can be rewritten based on other views. Its simplest representation is an AND-OR rewritability graph as in, e.g., [11]. For each view v at site s, there is a corresponding node in the AND-OR graph (if the same v is declared at n distinct sites $s_1, s_2, \ldots s_n$, there are n corresponding nodes in the graph). Moreover, for every view set $v_1, v_2, \ldots, v_k$, based on which v can be equivalently rewritten, there exists a $\wedge$ (AND) node a_v such that: (i) each of the nodes corresponding to $v_1, v_2, \ldots, v_k$ points to a_v, and (ii) a_v points to the v node. If v can be rewritten based on several view sets, there will be one $\wedge$ node pointing to v for each such rewriting possibility.

A sample RG over seven views is depicted in Figure 2. Each view can always be evaluated directly from the data source $\mathcal{D}$, thus, for each view v, there is a $\wedge$ node through which $\mathcal{D}$ is connected to v. Further, in Figure 2, v_2 and v_3 can be used to rewrite v_5, as shown by the lower $\wedge$ node pointing to v_5; v_3 and v_4 can be used to rewrite v_6, etc. Observe that there may be cycles in the RG: v_6 can be used to rewrite v_7 and vice versa. This entails that v_6 and v_7 are equivalent.

Formally, given a view set S, an RG is a directed graph, defined by the pair $(V \cup \{\mathcal{D}\} \cup A, E)$, such that:

- $V \cup \{\mathcal{D}\} \cup A$ is the set of nodes:
  - For each view $s_i \in S$, there exists a corresponding node $v_i \in V$.
  - $\mathcal{D}$ is the node corresponding to the data source.
  - A is the set of $\wedge$ nodes, each of which represents a rewriting of a view $s \in S$ based on a set of other views $\{s_1, s_2, \ldots, s_k\} \subseteq S \setminus \{s\}$.
- $E \subseteq ((V \cup \{\mathcal{D}\}) \times A) \cup (A \times V)$ is the set of directed edges that connect the graph's nodes as follows:
  - V nodes (as well as $\mathcal{D}$) can only point to A nodes, while A nodes can only point to V nodes.
  - Each node $a \in A$ has an indegree of at least one, and an outdegree equal to one.
  - For each view $v \in V$, there exists a $\wedge$ node $a_v \in A$ such that (i) $\mathcal{D} \rightarrow a_v \rightarrow v$ and (ii) $\mathcal{D}$ is the only node pointing to a_v.
  - For each view set $\{s_1, s_2, \ldots, s_k\}$ based on which another view s can be rewritten, there exists a $\wedge$ node $a_v \in A$ such that the edges (v_1, a_v), (v_2, a_v), $\ldots$, (v_k, a_v), $(a_v, v) \in E$.

Size of RG. The number of nodes in an RG is $|V| + |A| + 1$ (where 1 corresponds to $\mathcal{D}$). We have $|V| = |S|$, which is the number of views (subscriptions). As for the A nodes, there is one for every V node v, connecting $\mathcal{D}$ to v (thus, $|S|$ such A nodes). Moreover, we have one A node for every view set that can rewrite a view v. Since there are $|S| - 1$ views that can be used to rewrite v (we exclude v itself), we can have at most $2^{|S|-1}$ such A nodes for v. Thus, we have $|A| \leq |S| \times (2^{|S|-1} + 1)$.

We now turn to the number of edges. Since by definition the outdegree of each A node is one, there are $|A|$ edges from A to V nodes. Furthermore, an A node has at most $|S| - 1$ incoming edges (a rewriting can involve at most that many views), leading to at most $|A| \times (|S| - 1)$ edges from V to A nodes. Hence, we have $|E| \leq |S|^2 \times (2^{|S|-1} + 1) \approx |S|^2 \times 2^{|S|}$.

Clearly, an RG may be very large when there are many views. Therefore, it is also of interest to develop *partial rewritability graphs*, each of which can be seen as the RG from which some $\wedge$ nodes (and their corresponding input and output edges) have been erased.

Configuration (CFG). Given an RG, a configuration (CFG) is a subgraph of RG encapsulating a concrete choice of how to rewrite every view $v \in V$. Specifically, in a configuration, only a single $\wedge$ node points to each view. Moreover, there exists a directed path from $\mathcal{D}$ to each view of the RG[2].

Formally, given an RG $rg = (V \cup \{\mathcal{D}\} \cup A, E)$, a CFG $cfg = (V \cup \{\mathcal{D}\} \cup A', E')$ is a subgraph of rg such that:

- $A' \subseteq A$ and $E' \subseteq E$;
- for any $v \in V$, there exists exactly one $a \in A'$ such that $a \rightarrow v$;
- there exists a path from $\mathcal{D}$ to any view $v \in V$;
- for each node $a \in A'$, if edge $(v_i, a) \in E$ (for each $v_i \in V$), then $(v_i, a) \in E'$.

The last point in the above definition guarantees that when we select an A node to be included in cfg, we also select all its incoming edges that constitute the rewriting. Observe that a CFG completely specifies the paths along which data is disseminated to all the subscribers. Moreover, multiple data dissemination paths starting from the source $\mathcal{D}$ may meet, for instance, when two views v_1 and v_2, together, rewrite another view v_3.

The number of CFGs which may be derived from an RG is $\Pi_{v \in V}(in(v))$ where in denotes the indegree of a view node. It follows from the RG size estimations that the upper bound for the number of CFGs is $|S|^{2^{|S|}}$, which is extremely high.

2.2 Characteristics of a Configuration

We now discuss how to quantify the cost of a CFG.

For each rewriting ($\wedge$) node in a CFG, there can be several ways of distributing the effort entailed by the rewriting (typically selections and joins) across the network nodes in which the views reside. For example, consider the views v_2, v_3 and v_5 of Figure 2. Assume that v_2 resides on site n_2, v_3 on n_3 and v_5 on n_5. To join v_2 and v_3, they could both be shipped to the site n_5 and joined there. Alternatively, v_3 could be shipped to n_2, the join could be evaluated at n_2 and the results shipped to n_5, at a different resources utilization. More generally, the utilization incurred by the operations of a $\wedge$ node depend on the operations' types and ordering, where each operation runs etc.

Distributed resources utilization. To estimate the resources *utilization* of a given $\wedge$ node, we quantify the resources (e.g., I/O, CPU, bandwidth) needed for its execution over the various sites.

Let N be the set of network sites on which work can be distributed (we assume for simplicity N is the set of all the sites having subscriptions), and k be the number of distinct resources

[2]This also guarantees that a configuration is acyclic.

considered for each site, such as: I/O at that site, CPU, incoming and outgoing bandwidth, etc. Let $P_\wedge$ be the set of all physical plans for a given $\wedge$ node. We define the utilization function $u : P_\wedge \rightarrow \mathbb{R}^{|N| \times k}$, assigning to each plan $p \in P_\wedge$, the estimated resources utilization, along different resource dimensions, entailed by the evaluation of p. Observe that each result of u is a matrix stating the consumption along each dimension and at each site.

To enable comparing utilizations, we rely on a single *utilization aggregator* $\mathcal{U} : \mathbb{R}^{|N| \times k} \rightarrow \mathbb{R}$, which combines the utilization of all the different resource components of the sites involved in the execution of a plan, and returns a single (real) number. The aggregator may for instance sum up all the utilization components, possibly assigning them various weights depending on the metric and/or the site involved. In the sequel, for a plan $p \in P_\wedge$, we will simply write $\mathcal{U}(p)$ to denote the scalar aggregation $\mathcal{U}(u(p))$ of p's multidimensional utilization.

Finally, for a given $\wedge$ node $a \in A$, we denote by $\mathcal{U}(a)$ the smallest value of $\mathcal{U}(p)$, over all the plans $p \in P_\wedge$. Moreover, the utilization of a CFG $cfg = (V \cup \{\mathcal{D}\} \cup A', E')$ is:

$$\mathcal{U}(cfg) = \sum_{a \in A'} \mathcal{U}(a).$$

Latency. In a CFG, given a data item i and subscription v such that $v(i) \neq \emptyset$, the data dissemination *latency* of v with respect to i, denoted $\lambda(v, i)$, is the time interval between the publication of i and the moment when $v(i)$ reaches the site of v. In the sequel, we may simply use $\lambda(v)$ to denote v's latency.

Clearly, $\lambda(v)$ is determined by the paths in CFG followed by the data that is moving from $\mathcal{D}$ to v. Each $\wedge$ node a encountered along these paths adds to the latency its contribution, which we term *local latency* of a. That reflects the delays introduced on the propagation of data in the rewriting graph, by evaluating that rewriting. For instance, if the best physical plan for a $\wedge$ node requires shipping data across the network from n_1 to n_2 and performing a join at n_2, the local latency of this node will reflect the data transfer and the processing time in the join. We assume available a *local latency estimation function* l, which estimates the local latency introduced by a. We stress that $l(a)$ characterizes only the operations at the rewriting node a, and not the behaviour of its input(s).

Given that for every subscription v there is a single $\wedge$ node a_v pointing to v (see RG definition, Section 2.1), v's latency is equal to the *total latency* of a_v (denoted $\lambda(a_v)$), thus $\lambda(v) = \lambda(a_v)$. This latency can be computed by adding a_v's local latency $l(a_v)$ to the maximum latency of the subscriptions $\{v_i\}$ that are inputs to a_v. Denoting by $v_i \rightarrow a_v$ the fact that node v_i points to a_v in the RG, we have:

$$\lambda(a_v) = \lambda(v) = max_{v_i \rightarrow a_v}(\{\lambda(v_i)\}) + l(a_v)$$

Note that the latency of $\mathcal{D}$ is defined as 0. We also define the latency of a CFG $cfg = (V \cup \{\mathcal{D}\} \cup A', E')$ as follows:

$$\lambda(cfg) = \sum_{v \in V} \lambda(v).$$

Cost. We define the cost of a $\wedge$ node a in a CFG as a linear combination of its utilization and latency:

$$\mathcal{C}(a) = \alpha \times \mathcal{U}(a) + \beta \times \lambda(a)$$

where α and β are coefficients controlling the importance given to the utilization and latency. A high α prioritizes solutions of low utilization, incurring a low consumption of resources across the network, while a high β prefers solutions having a low latency, favoring quick dissemination of data to the subscribers. Finally, we define the cost of a CFG $cfg = (V \cup \{\mathcal{D}\} \cup A', E')$:

$$\mathcal{C}(cfg) = \sum_{a \in A'} \mathcal{C}(a).$$

Constraints. In practice, resources such as CPU, memory, incoming and outgoing network bandwidth, are limited on each site. This has to be taken into account when deciding whether to use a view v_1 to feed another view v_2 with data, since doing so incurs some consumption of resources on the site of v_1: such resource consumption should be kept within the capacity limits. Each site may have different such *capacity constraints*, according, for instance, to its specific infrastructure or available bandwidth.

We make the simplifying assumption that there is a single view published in each network site. We model capacity constraints by a single integer B_v^{out}, which is the maximum number of views that can be served by v (and which coincides with the maximum number of views served by a network site, since there is one view per site), and design our algorithms to operate within these constraints. This can be easily extended to more (and more complex) constraints.

2.3 Problem Statement

Given an RG $rg = (V \cup \{\mathcal{D}\} \cup A, E)$, a cost function $\mathcal{C}$, a limit B_v^{out} for each $v \in V$, as well as a limit $B_\mathcal{D}^{out}$ for the data source, the problem we address is to find a CFG $cfg = (V \cup \{\mathcal{D}\} \cup A', E')$, such that:

1. Capacity constraints are respected:
$$\forall v \in V \cup \{\mathcal{D}\},\ out(v) \leq B_v^{out}$$
where $out(v)$ denotes the outdegree of node v in the CFG;
2. The cost of CFG $\mathcal{C}(cfg)$ is minimized.

3. CONFIGURATION SELECTION

We now describe our approach for selecting a low-cost configuration. We start by discussing RG construction in Section 3.1. Section 3.2 provides an overview of the CFG selection, a two-step process described in detail in Section 3.3 and 3.4, respectively. Section 3.5 shows how we treat with CFG updates (view addition/removal).

3.1 Rewritability Graph Generation

Given a set of views, we show how to construct the corresponding RG, modelling the ways to rewrite views based on other views.

Naive RG generation. Assume we initially create a graph that contains the nodes $(V \cup \{\mathcal{D}\})$, as well as the $\wedge$ nodes that are needed to connect $\mathcal{D}$ with each view $v \in V$ (along with the corresponding edges). Based on this graph, the most direct way of building the RG is by calling the view-based rewriting algorithm exhaustively, and adding, each time a rewriting is found, the corresponding $\wedge$ nodes and edges. This simple method requires calling the rewriting algorithm $|V|$ times, using each time $|V| - 1$ views. Given the typically high complexity of view-based query rewriting algorithms, this method is unlikely to scale to large problems. Moreover, even if we optimize the calls to the rewriting algorithm (e.g., by reducing the number of views we use as input each time, as discussed in Section 4), the resulting *complete* RG is usually too dense, hampering in turn the process of choosing a CFG from RG.

Partial RG generation. In the interest of efficiency, one can limit the search performed during each call to the rewriting algorithm to at most k rewritings. In other words, we only consider the first (at most) k alternative ways we find to rewrite a given query. Clearly, the internals of the rewriting algorithm affect the order in which rewritings are explored and, thus, the first k rewritings found; we will revisit this issue in Section 4. Algorithm 1 outlines the construction of the partial RG, obtained through this limited exploration of rewritings. When a view cannot be rewritten based on the others, Algorithm 1 connects it directly to the data source $\mathcal{D}$.

Algorithm 1: Partial RG Generation

Input : View set V, maximum number k of rewritings per view
Output: RG of V with at most k rewritings per view

```
// RG initially contains only V and D
A ← ∅, E ← ∅, G ← (V ∪ {D} ∪ A, E)
foreach v ∈ V do
    rewrNo ← 0
    while hasNextRewriting(v, V \ {v}) and (rewrNo < k) do
        // Get next rewriting
        rw ← nextRewriting(v, V \ {v})
        A ← A ∪ {rw}    // Add rewriting (∧) node rw
        E ← E ∪ {(u_i, rw)}, ∀u_i ∈ rw  // Add edges to rw
        E ← E ∪ {(rw, v)}               // Add edges to v
        rewrNo++
// All views are also fed by D
E ← E ∪ {(D, u)}, ∀u ∈ V
return G
```

3.2 Configuration Selection Overview

We now turn to the problem of selecting out of a (possibly partial) RG, a CFG that minimizes the cost as a weighted sum of *utilization* and *latency*, under *capacity constraints* (as per our problem statement in Section 2.3).

Complexity and relationship with known problems. We now discuss how our problem relates to already studied graph problems.

First, consider *resources utilization optimization alone*, that is, ignore the latency and capacity constraints. This simplified problem can be solved in linear time, by selecting for each view v in an RG, the lowest resources utilization $\wedge$ node pointing to v, together with the corresponding edge and the $\wedge$ node's incoming edges.

Now assume given bounds on the number of views that can be fed (i) from $\mathcal{D}$ and (ii) from each view, and consider the problem of finding a CFG that respects these *capacity constraints, without considering the cost*. This version of the problem is more complex than the previous one, as choosing $\wedge$ nodes is no longer a local decision for each view v in the RG: selecting an $\wedge$ node can break the capacity constraints of any of the nodes that are serving it.

This last problem of selecting a CFG under capacity constraints is largely connected to the problem of finding a *Degree-bounded Arborescence* (DBA, for short) in a given graph. An arborescence is a spanning tree of a directed graph rooted at a given root node. Finding a DBA is NP-hard [1]; the NP-hardness is due to the fact that, in order to respect the degree bounds, the edge-selection decisions cannot be local. We have shown that the DBA problem can be reduced in polynomial time to finding a capacity-constrained CFG, which is already a specialization of the general problem we consider (Section 2.3), since it does not take into account the cost. This leads us to the following proposition:

PROPOSITION 3.1. *Finding a minimum-cost CFG under capacity constraints is NP-hard.*

The proof is given in the extended version of this paper [14].

Importantly, the latest effective techniques for solving DBA and even more general network design problems, rely on solving linear relaxations of *Integer Linear Programs* [17]. The idea is to use one boolean variable x_i to encode whether a node (or edge) is part of the solution, and to formulate the total utilization (objective function) as a weighted sum of all the variables, with the weights being the respective node (or edge) utilizations. Such an ILP formulation can be handed to an ILP solver, which takes advantage of advanced techniques that enable it to solve large-size problems corresponding in our context to many views and many rewritings.

Two-steps optimization approach. Although our problem (Section 2.3) is naturally expressed as an ILP when one considers capacity constraints and optimizes for utilization (ignoring latency), and can thus be delegated to an ILP solver, it turns out that one cannot rely on an ILP solver to also reduce *latency* (as explained in Section 3.3). Thus, our approach for addressing the problem is organized in two steps:

1. Formulate our optimization problem *considering utilization and constraints only* as an ILP and delegate it to an efficient ILP solver. We describe this next in Section 3.3.
2. Post-process the utilization-optimal configuration returned by the solver (if one exists under the given constraints) to reduce latency in a heuristic fashion, as described in Section 3.4.

3.3 CFG Utilization Optimization With ILP

Integer Linear programming (ILP) is a well-explored branch of mathematical optimizations. A wide class of problems can be expressed as: given a set of linear inequality constraints over a set of variables, find value assignments for the variables, such that a target expression on these variables is minimized. Such problems can be tackled by dedicated *ILP solvers*, some of which are by now extremely efficient, benefiting from many years of research and development efforts. Inspired by the model for directed graphs of [17] (with some changes), we formulate our problem as an Integer Linear Program as follows.

Variables. For each node $n \in V \cup \{\mathcal{D}\} \cup A$, we denote by E_n^{in} and E_n^{out} the sets of its incoming and respectively outgoing edges. Selecting a CFG amounts to selecting *one way to compute each view*, which is equivalent to selecting for each view v, one of the $\wedge$ nodes pointing to v, or, equivalently, one edge from E_v^{in}. Thus, for each $v \in V$ and $e \in E_v^{in}$, we introduce a variable x_e, taking values in the set $\{0, 1\}$, denoting whether or not e is part of the CFG.

Coefficients. Our problem model attached rewriting evaluation utilization to the rewriting nodes, through the utilization function U returning for each $\wedge$ node $a \in A$, the associated utilization $\mathcal{U}(a)$ which aggregates various types of utilizations (CPU, I/O, network, etc.) Further, as explained in Section 2.2, $\mathcal{U}(a)$ is the smallest over the utilizations of all physical plans that could be used for this rewriting. To simplify the presentation, and since there is a bijection between A, the set of $\wedge$ node sets, and the set of edges entering view nodes, namely $\cup_{v \in V} E_v^{in}$, we *move the utilization of each rewriting, to the edge going from the rewriting $\wedge$ node, to the corresponding rewritten view*. The other edges, in particular all those entering $\wedge$ nodes, are assumed to have zero utilization. Thus, for each rewriting node $a \in A$ and edge $e \in E_a^{out}$ (recall that $E_a^{out} = \{e\}$, that is, each a node has exactly one outgoing edge), we denote by $\mathcal{U}_e$ the utilization $\mathcal{U}(a)$. Our final ingredient is the B_v^{out} bounds on the views fan-out, introduced in Section 2.2.

Putting it all together. Our problem's ILP statement is given in Table 1. Equation (1) states that each x_e variable takes values in $\{0, 1\}$, (2) ensures that every view is fed exactly by one rewriting, (3) states that if the (only) outgoing edge of a $\wedge$ node is selected, all of its inputs are selected as well, and finally (4) ensures the respect of the B_v^{out} constraint.

ILP example. Consider the RG shown at the top of Figure 3, where for illustration we have added to each $\wedge$ node leading to the view v_i, the subscript $_i$ and a superscript ^{j} with $j = 0, 1, \ldots$. For each edge (n, m) in the RG, where n and m are two RG nodes, we introduce a variable $x_{n \to m}$ stating whether that edge is part of the chosen configuration. For simplicity, for each node $\wedge_i^j$ pointing to the view v_i, we write x_i^j instead of $x_{\wedge_i^j \to v_i}$. Thus, x_i^j is a boolean variable whose value 1 indicates that the view v_i is filled by its

Minimize: $\mathcal{U} = \sum_{e \in E} \mathcal{U}_e x_e$

subject to:

$x_e \in \{0,1\}$	$\forall e \in E$	(1)
$\sum_{e \in E_v^{in}} x_e = 1$	$\forall v \in V$	(2)
$\sum_{e \in E_a^{in}} x_e = x_{E_a^{out}} \times \lvert E_a^{in} \rvert$	$\forall a \in A$	(3)
$\sum_{e \in E_v^{out}} x_e \leq B_v^{out}$	$\forall v \in V \cup \{\mathcal{D}\}$	(4)

Table 1: Utilization optimization problem as an ILP.

rewriting $\wedge_i^j$. Moreover, for each $\wedge_i^j$, let c_i^j be the utilization of the processing incurred by that rewriting.

The linear program whose solution is a minimum-utilization CFG for this graph is shown in the lower part of Figure 3. Equation numbers at the left refer to the generic equations in Table 1.

Non-linearity of latency. Still on the RG in Figure 3, we now turn to quantifying the latency of each view. Let l_i^j be the latency of each rewriting $\wedge_i^j$; for simplicity we include therein the impact of all the transfers and processing incurred by the rewriting.

We consider that $\mathcal{D}$ implements an efficient algorithm allowing it to match *simultaneously* all the subscriptions it serves, against each newly published document. This is the case in state-of-the-art algorithms such as [7], and also in our simpler implementation. Thus, the latency component that is due to *subscription matching at* $\mathcal{D}$ (as opposed to latency incurred by shipping data from $\mathcal{D}$ and possibly further processing and shipping of data) is the same for all views, and we ignore it without loss of generality.

Applying our formulas defining latency, we obtain $\lambda(v_2) = l_2^0$, $\lambda(v_3) = l_3^0$, since v_2 and v_3 are fed directly from the publisher. Since v_1 can be fed either through $\wedge_1^0$ or $\wedge_1^1$, its latency is:

$$\lambda(v_1) = x_1^0 l_1^0 + x_1^1(l_1^1 + max(\lambda(v_2), \lambda(v_3))) = x_1^0 l_1^0 + x_1^1(l_1^1 + max(l_2^0, l_3^0))$$

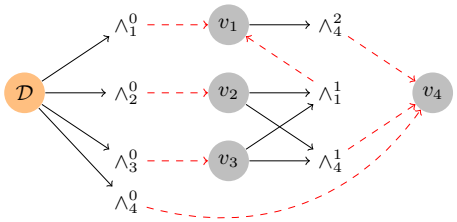
Similarly, given that v_3 can be fed through three different $\wedge$ nodes, we have:

$$\lambda(v_4) = x_4^0 l_4^0 + x_4^1(l_4^1 + max(\lambda(v_2), \lambda(v_3)) + x_4^2(l_4^2 + \lambda(v_1)) = x_4^0 l_4^0 + x_4^1(l_4^1 + max(l_2^0, l_3^0)) + x_4^2(l_4^2 + x_1^0 l_1^0 + x_1^1(l_1^1 + max(l_2^0, l_3^0)))$$

Observe that the above expression unfolds into a sum having among its terms $x_4^2 x_1^0 l_1^0$ and $x_4^2 x_1^1 l_1^1$, which is *non-linear in the problem's variables* x_i^j; in contrast, the latencies of v_1, v_2 and v_3 are *linear* combination of these variables. As a consequence, in these examples and in general, *configuration latency cannot be pushed into the ILP objective function*, which only admits linear combinations of variables.

The intuition behind this non-linear behavior is easy to trace on the RG in Figure 3. The variables which end up multiplied correspond to paths of length 2, leading to v_4 *through* v_1. If $x_1^0 = x_4^2 = 1$, v_1 is fed from the source and v_4 from v_1. If $x_1^1 = x_4^2 = 1$, v_1 is fed from v_2 and v_3 and v_4 from v_1. The multiplication of variables corresponds to the logical conjunction of the edge selection decisions they correspond to.

Concluding this discussion, we will rely on ILP to solve efficiently and exactly the utilization optimization problem, and reduce in a second step the latency of the configuration thus obtained.



Minimize: $\mathcal{U}_1^0 x_1^0 + \mathcal{U}_1^1 x_1^1 + \mathcal{U}_2^0 x_2^0 + \mathcal{U}_3^0 x_3^0 + \mathcal{U}_4^0 x_4^0 + \mathcal{U}_4^1 x_4^1 + \mathcal{U}_4^2 x_4^2$

subject to:

eq.(1)	$x_i^j \in \{0,1\}, \forall i,j$
eq.(2)	$x_1^0 + x_1^1 = 1;\ x_2^0 = 1;\ \ x_3^0 = 1;\ \ x_4^0 + x_4^1 + x_4^2 = 1;$
eq.(3)	$x_{\mathcal{D}\to\wedge_1^0} = x_1^0;\ x_{\mathcal{D}\to\wedge_2^0} = x_2^0;\ x_{\mathcal{D}\to\wedge_3^0} = x_3^0;$ $x_{\mathcal{D}\to\wedge_4^0} = x_4^0;\ x_{v_1\to\wedge_4^2} = x_4^2;$ $x_{v_2\to\wedge_1^1} + x_{v_3\to\wedge_1^1} = 2x_1^1;\ x_{v_2\to\wedge_4^1} + x_{v_3\to\wedge_4^1} = 2x_4^1;$
eq.(4)	$x_{v_1\to\wedge_4^2} \leq B_{v_1}^{out};\ x_{v_2\to\wedge_1^1} + x_{v_2\to\wedge_4^1} \leq B_{v_2}^{out};$ $x_{v_3\to\wedge_1^1} + x_{v_3\to\wedge_4^1} \leq B_{v_3}^{out};$ $x_{\mathcal{D}\to\wedge_1^0} + x_{\mathcal{D}\to\wedge_2^0} + x_{\mathcal{D}\to\wedge_3^0} + x_{\mathcal{D}\to\wedge_4^0} \leq B_{\mathcal{D}}^{out};$

Figure 3: Sample RG and corresponding ILP model.

3.4 CFG Latency Optimization

In this second stage, we seek to improve the latency of the CFG obtained by solving the LP problem (corresponding to the utilization minimization under constraints), by incremental changes on this CFG. We start by introducing a helper notion:

Impact of a view on CFG latency. Given a CFG cfg, we define the *impact* of a view v, denoted by $\mathcal{I}(v)$, as an estimation of v's impact on the latency of all of the views that are fed with data by v, directly or indirectly. Formally:

$$\mathcal{I}(v) = \lambda(v) \times |\text{nodes of } rg \text{ reachable from } v|$$

In the above, we consider that any rg node reachable from v is potentially impacted by the latency introduced by v, and, thus, multiply v's latency by the number of such nodes. We also define the impact of a rewriting rw_v pointing to view v to be equal to the impact of v: $\mathcal{I}(rw_v) = \mathcal{I}(v)$.

The LOGA algorithm. We have devised a Latency Optimization Greedy Algorithm (LOGA, in short), given in Algorithm 2, which incrementally tries to improve the latency of a CFG cfg obtained from an RG rg. The algorithm uses the original rg in order to replace a rewriting in cfg with another one that leads to a CFG with a globally smaller latency. It initially orders the rewritings of cfg in descending order of impact, and then tries to replace first the rewritings with the biggest impact. Such replacements are made (i) without violating the B^{out} bounds, and (ii) without assigning views again to $\mathcal{D}$, since the goal of our work is precisely to spread the data dissemination work.

Incremental re-computation of latency. As explained above, a change in the latency of a view v in a CFG cfg might affect the latency of every view in cfg accessible from v. Therefore, when the latency of v changes as a consequence of a replacement, LOGA performs a traversal in topological order of the cfg sub-DAG rooted at v, to recompute the latency only of the affected views.

Recomputing impact of views. As the CFG changes through rewriting replacements, the number of nodes reachable from any given view node v must be recomputed. This number is needed in order to update the impact $\mathcal{I}(v)$, at line 5 of Algorithm 2. The number of

Algorithm 2: Latency Optimization Greedy Algorithm (LOGA)

Input : CFG cfg, RG rg
Output: Latency optimized version of cfg

$newLat \leftarrow \lambda(cfg)$
repeat
  $prevLat \leftarrow \lambda(cfg)$
  $rwList \leftarrow \{rw \in cfg \mid \nexists \text{ edge } (\mathcal{D}, rw)\}$
  $rwList \leftarrow reorder(rwList)$ in desc. order of interest $\mathcal{I}(rw)$
  foreach $rw \in rwList$ **do**
    $minLat \leftarrow \lambda(cfg)$; $best_{rw} \leftarrow null$
    `// Replace rw with its latency-optimal alternative (if any)`
    foreach $rw' \in rg$ *s.t.* rw, rw' *feed the same view* **do**
      replace rw with rw' in cfg
      if $(\forall v \in cfg, outdegree(v) \leq B_v^{out})$ **and** $(\lambda(cfg) < minLat)$ **then**
        $minLat \leftarrow \lambda(cfg)$
        $best_{rw} \leftarrow rw'$
      replace rw' with rw in cfg `// leave cfg intact`
    if $best_{rw} \neq null$ **then**
      replace rw with $best_{rw}$ in cfg
      $newLat \leftarrow \lambda(cfg)$
until $prevLat = newLat$
return cfg

nodes reachable from v is determined by the rewriting opportunities, which in turn depend on the actual views etc. In the worst case this may require a costly traversal of the whole CFG, however, as our experiments show (Section 5), much fewer nodes are traversed and thus this operation is not expensive in practice.

3.5 Incremental CFG Computation

Adding a new view v to an existing configuration cfg, goes as follows: we compute v's rewritings and add them to the existing RG. We then search the RG for a rewriting rw with the least cost $\mathcal{C}(rw)$ such that no bounds are violated in cfg. If such a rewriting rw exists, we add it to cfg; otherwise, v is assigned to the data source. After a certain number of new subscriptions have been added, or when the data source's are been reached, the solver and LOGA are re-invoked and a full CFG selection takes place.

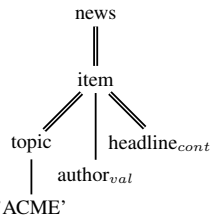
When a subscription v is withdrawn or its site fails, the views depending on v, that is those to whose cfg rewritings v contributes, are treated as new and the above incremental process is followed for each of them.

4. VIEW-BASED REWRITING

We now describe the view-based rewriting framework underlying Delta. Section 4.1 presents some preliminary notions on views and rewritings, whereas Section 4.2 describes an auxiliary structure, the embedding graph, which is used for building the RG. Then, Section 4.3 presents our algorithm for efficiently rewriting a subscription (view) based on the others. Its novelty resides in its capability to produce a specified number of solutions, crucial in our setting where not all rewriting opportunities are explored. Finally, Section 4.4 discusses how other view-based rewriting algorithms could be substituted to ours, to port the Delta architecture in other distributed dissemination contexts.

4.1 Views and Rewritings

Since our target applications concern the dissemination of structured text news, and in order to leverage our previous system development [15, 18], we built our system for disseminating XML documents to a network of subscriptions expressed in a rich flavor of XML queries.



Each view is defined by a *tree pattern query*, where nodes are labeled with XML element or attribute names, while edges encode parent-child (single) or ancestor-descendant (double) relationships. Unlike XPath 1.0, and close to XPath 2.0 and to simple XQuery for-let-where-return (FLWR) expressions, our tree patterns may return content from multiple nodes. For instance, the subscription at left requests the *author and headline of all published news about company "ACME"*. Observe that the subscription requires the XPath *text value* (denoted val) of the author, while for each matching headline, the complete XML subtree rooted at the ⟨headline⟩ element is returned (denoted $cont$). Finally, each pattern node can be annotated with the token ID, denoting that the identifiers of XML nodes matching this pattern node are part of the pattern query result.

Node IDs are implemented by virtually all efficient XML engines. Therefore, we include IDs in our views, since, as we have shown in [16], view joins based on such IDs may lead to very efficient rewritings. As a simple example, consider the query q defined as $//a[//c]//b$ and the views $v_1 = //a$, $v_2 = //a_{ID}[//c]$ and $v_3 = //a_{ID}//b$, where v_2 and v_3 store IDs for the a nodes. One can rewrite q as $v_2 \bowtie_{a.ID} v_3$, or alternatively as $v_1[//c]//b$. The former is likely to be much more efficient than the latter, because v_2 and v_3 are more selective than v_1, especially if few a elements have b and/or c descendants.

The full tree pattern language is described in [18], which also provides an equivalent view-based rewriting algorithm for this language. Unsurprisingly, this algorithm has high complexity, therefore, it is not applicable in a setting like ours with a very large numbers of views. Therefore, we consider here a sub-language of the one considered in [16, 18], that is, we assume *all nodes are annotated with ID*. Moreover, to increase the possibilities of view-based rewriting, we assume *IDs are structural*: by comparing two node IDs one can decide if the node corresponding to the one is a parent/ancestor of the node corresponding to the other. Node IDs are invisible to the user; they are added by the system to the user-issued tree patterns. Storing IDs in subscription data brings a space overhead, but not a very significant one, especially if one relies on space-efficient encodings of such views [28]. *Restricting the view language to endow all nodes with ID reduces view-based rewriting to a set-cover problem*, as we explain shortly below.

View embedding. It has been shown [18, 25] that a tree pattern view v may participate in an equivalent rewriting of another tree pattern view q only if there exists an embedding $\phi : v \rightarrow q$ respecting (1) node labels, i.e., for any node $n \in v$, $label(n) = label(\phi(n))$, and (2) structural relationships between nodes, that is, for any two nodes $n, m \in v$, if n is a /-child (resp., //-child) of m, then $\phi(n)$ is a /-child (resp., descendant) of $\phi(m)$. Finally, ϕ must not contradict value predicates from the query, i.e., for any node $n \in v$, such that $m = \phi(n) \in q$, if m is annotated with predicate $[val = c_1]$ for some constant c_1, then n must not be annotated with predicate $[val = c_2]$ for some constant $c_2 \neq c_1$. It follows readily from the above properties of embeddings that:

COROLLARY 4.1. *If a view v embeds into a query q, the labels of v are a subset of the labels of q.*

View coverage. We say that a set of views V *covers* a given view q, iff, for every attribute att of a node $n_q \in q$, there exists a node n_v belonging to a view $v \in V$ and an embedding $\phi : v \rightarrow q$ such that $\phi(n_v) = n_q$ and n_v is also annotated with att. We call such a view set V an *embedded attribute set cover* (EAC) for q.

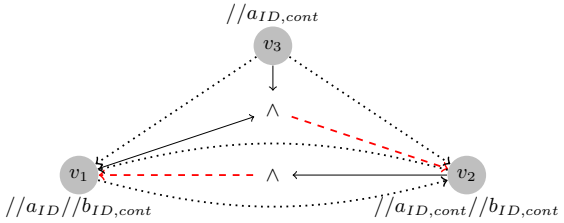


Figure 4: Superposed EG and RG over three views.

If we restrict the rewriting algorithm [18] to the case when all view nodes are annotated with ID, it can be shown that the existence of an EAC V for q is a necessary and sufficient condition for an equivalent rewriting of q based on V to exist. Indeed, given an EAC V for q, the rewriting can be built using structural joins (based on the node IDs) between all the involved views, and adding all required structural predicates (imposing structural relationships present in the query but not in the views), as well as possible value selection predicates still needed. We formalize this as follows:

PROPOSITION 4.1. *A query q can be equivalently rewritten based on a set of views V, iff V is an EAC for q.*

Observe that such a rewriting may be non-minimal; we revisit this issue in Section 4.3.

4.2 Embedding Graph (EG)

Given a view set V, in order to build the corresponding RG, we must solve $|V|$ view-based rewriting problems, one for each view based on the others. To speed up the rewriting process, we can exploit Proposition 4.1 to attempt to rewrite a given view v, only using those views that embed into v. Thus, we are interested in all view pairs (v_1, v_2) such that v_1 embeds into v_2. We encode this embedding information in an *embedding graph* (EG, in short), which is a directed graph having a node for each view $v \in V$ and an edge (v_1, v_2), with $v_1, v_2 \in V$, iff v_1 embeds in v_2. Figure 4 depicts a sample EG (view nodes, dotted edges), along with the corresponding RG (view and $\wedge$ nodes, solid and dashed edges). Next to each view node, we give its view definition. For instance, v_3 embeds in v_1 and v_2 (as shown by the dotted edges).

Testing whether v embeds into v' takes at most $|v| \times |v'|$ operations [18], leading to a total complexity of $O(|V|^2 \times |v|^2_{max})$ for creating the EG, where $|v|_{max}$ is the size of the largest view in V. Such tests may get quite expensive for large V sets.

To improve performance, we pre-filter views, based on Corollary 4.1: for v to embed into v', the labels of v must be among the labels of v'. We organize the view definitions in a prefix trie specifically designed to support subset queries [12]. Using this trie, given a view v, we can efficiently identify all the views u_i such that $labels(u_i) \subseteq labels(v)$.

Algorithm 3 shows how to construct an EG given a set of views V. The algorithm starts by constructing a trie as explained above. Then, it uses the trie as an index to efficiently build the EG: for a given view v, the trie returns all views whose labels are a subset of v's labels. Only the views thus obtained are tested for embedding into v. Since our pre-filtering has no false negative, Algorithm 3 generates the complete EG.

EG cycles and their consequences. It is possible for two views to embed into each other, as for example v_1 and v_2 in Figure 4, leading to cycles in the EG. In some cases, cycles in the EG lead to cycles in the RG. For instance, in Figure 4, although the EG cycle between v_1 and v_2 does not directly translate to an RG cycle, view v_3 enables some additional rewritings (such as the one represented by the upper $\wedge$ node), and in turn these lead to an RG cycle (involving v_1, v_2 and the two $\wedge$ edges).

Algorithm 3: Trie-based EG Construction Algorithm

```
Input  : View set V
Output: EG of V
1  E ← ∅; EG ← (V, E)        // Initially empty edge set
2  T ← createTrie(V)          // Create the trie for V
3  foreach v ∈ V do
       //Retrieve from T all u s.t. labels(u) ⊆ labels(v)
         and add edges corresponding to embeddings
4      foreach u ∈ {T.lookUp(v)} do
5          if u embeds into v then E ← E ∪ {(u, v)}
6  return EG
```

Algorithm 4: Cover-based greedy rewriting (CGR)

```
Input  : View v, EG eg = (V_eg, E_eg), max. number k of rewritings
Output: List with at most k rewritings of v based on the views of eg
// Get from eg all views embeddable in v
1  V ← {u_i | (u_i, v) ∈ E_eg}
2  rwList ← ∅                 // List with rewritings for v
3  visited ← ∅                // Set of already visited EACs
4  if ∃ attribute att ∈ v, not covered by any u ∈ V then return ∅
5  crtEAC ← ∅                 // Current EAC view set
6  backtrackFindEAC(v, V, crtEAC)
7  return rwList
8  Procedure backtrackFindEAC(v, V, crtEAC)
9      if crtEAC covers all v's attributes and crtEAC ∉ visited then
10         visited ← visited ∪ {crtEAC}
           // Get rewriting from EAC and add to rwList
11         rwList.add(EACtoRw(crtEAC))
12         if (rwList.size = k) then return
       // Get views not yet used in crtEAC
13     remainViews ← V \ crtEAC
14     if remainViews = ∅ then return
15     remainViews ← sort(altViews) in desc. order of interest i
16     foreach v_alt ∈ altViews do
17         crtEAC ← crtEAC ∪ {v_alt}
           backtrackFindEAC(v, V, crtEAC)
18         crtEAC ← crtEAC \ {v_alt}
```

RGs featuring such cycles pose an issue since the ILP solver may return a CFG with cycles, e.g., feeding v_1 from v_2 and v_2 from v_1 in this example, without using the publisher $\mathcal{D}$ at all. Such CFGs do not make sense from the application perspective, since the data path feeding each view must start at the publisher $\mathcal{D}$.

It can be shown that an RG has cycles only if the EG it has been built from had cycles. To avoid RGs (and CFG) cycles, we break EG cycles using the cycle removal algorithm [9].

4.3 View-based Rewriting Algorithm

We now describe our rewriting algorithm (Algorithm 4). As stated in Proposition 4.1, to find rewritings of v it suffices to find all embedded attribute set covers (EACs) of v, and to build an efficient rewriting from each such EAC.

The novelty of our algorithm is that it generates solutions *incrementally on-demand*, a useful feature given that we only consider k alternative rewritings for each subscription (recall Section 3.1). Since some rewritings may never be developed, Algorithm 4 strives to develop the most promising rewritings first, that is those whose evaluation utilization is likely to be low. This is done by ordering candidate views in decreasing order of their *interest* w.r.t. rewriting (covering) a given view v: the more v attributes *currently uncovered* by a partial rewriting are covered by a view v', the more interesting it is to add v' to (join it with) the respective partial rewriting. Clearly, as views are added to the rewriting, view interests have to be recomputed. The algorithm is based on depth-first exploration and backtracks to move from one rewriting to the next one.

View Set Metric	Value
Number of views (unique)	100,000
Avg. number of predicates per view	0.72
Avg. number of predicates per node	0.11
Avg. number of nodes per view	6.13
Avg. number of return nodes per view	2.52
EG Metric	**Value**
Number of edges	10,592,053
Number of edges deleted to remove cycles	18,665
% of views in which at least one view is embedded	99.95
Generation time (sec)	452
RG Metric	**Value**
Number of rewritings ($\wedge$ nodes)	2,692,139
Number of edges	8,589,822
Generation time (sec)	127
Views rewritten by other views	94,835
Avg. number of views used in a rewriting	2.15
Avg. $\|E^{out}\|$	57.9

Table 2: Experiment settings and EG/RG statistics.

First, the algorithm uses the EG to retrieve the view set V containing only the views embeddable in v. The EAC exploration starts with an empty EAC, and at each point the highest-interest view not already in the current EAC is added to it. We compute the interest of adding a candidate view u to the EAC, given that a subset of V has already been selected, by counting *how many attributes of v not covered by the EAC views, are covered by the candidate u*.

For example, when rewriting view v $/a_{ID,cont}/b_{ID,cont}$ and considering a candidate view $u_1 = /a_{ID}/b_{ID,cont}$, the interest of u_1 is 3, since u_1 covers the attribute ID in two nodes of v as well as $b.cont$. Once u_1 is selected, the interest of another candidate view $u_2 = /a_{ID,cont}/b_{ID}$ is 1, since the only attribute of v not previously covered by u_1 and covered by u_2 is $a.cont$. When several views have the same interest, the tie is broken by picking the one that covers attributes from *the largest number of v nodes*. Once an EAC for v is found, we transform it to a rewriting expression and add it to the list of rewriting solutions.

In the worst case, Algorithm 4 will develop all subsets of V. However, in practice, since we only seek k rewritings, the number is typically much less, as we verified through our experiments.

Rewriting minimization. Algorithm 4 may generate rewritings which include redundant views. These views may be removed from the rewriting while leaving it still equivalent to the target view. Non-minimality is due to the greedy nature of Algorithm 4: after a view u was included in a rewriting, another set of views $\{u_1, u_2, \ldots, u_k\}$ may be added such that, together, the views in the set cover all attributes that u was selected for. This makes u redundant although it was not when initially added. To build efficient (non-redundant) rewritings, we minimize them in a post-processing fashion as in [25]: remove a random view from a non-minimal rewriting, then check if this has compromised the rewriting. If yes, the view is put back in the rewriting, another view is removed, etc.

4.4 Generality of our Approach

The core concepts and framework of Delta, discussed in Section 2, are independent of the concrete underlying data model, query language and query rewriting algorithm. While Delta is currently implemented and deployed for XML subscriptions, it can be easily adapted to another data model and subscription language. We briefly discuss the rewriting-related components needed to do so.

First, an algorithm for equivalent view-based query rewriting is needed, such as proposed in the literature, e.g., for relational [21] or XML data [25, 18]. In particular, the set-cover-based algorithm described above can be used as-is if we model subscriptions simply as *key-value pairs*, e.g., "topic=sport and location=England", as considered in many publish-subscribe data management settings (e.g., [4]). We rely on this algorithm to build the RG.

B^{out}	30	50	100	∞
% rewritten views	94.3	94.7	94.7	94.7
CFG utiliz. ($\times 10^{13}$)	3.49	3.32	3.31	3.13
Avg. views per rewriting	1.77	1.78	1.79	1.8

Table 3: Impact of B^{out} on the selected CFGs.

Second, while building the EG is optional, for many-view settings it is likely to significantly improve performance, by limiting the view set input to the rewriting algorithm. The embedding criterium we used to build the EG has natural counterparts in other data models, e.g., the classical containment mappings [3]. If these are not implemented or their computational cost is high, the EG can be approximated using any non-lossy pruning. For instance, if one considers relational queries as subscriptions, we could add an edge (v_1, v_2) in the EG as soon as the tables in v_1 are a subset of those in v_2, and for each table, the constants used in selections on that table in v_1 are used in selections over the same tables in v_2.

5. EXPERIMENTAL EVALUATION

In this Section we present the experimental evaluation of our system. We describe our setup in Section 5.1, and discuss the construction of EGs and RGs in Section 5.2. Section 5.3 studies the utilization-based selection of CFGs through ILP, while Section 5.4 discusses how to improve the latency of such CFGs. Finally, Section 5.5 presents the deployment of Delta in a wide area network.

5.1 Experimental Setup

We implemented all our algorithms in Java, except for the utilization based CFG selection algorithm (Section 3.3), for which we made use of the Gurobi ILP solver [29].

We relied on YFilter [7] to generate our views, based on the XMark DTD [23]. We generated a view set of 100,000 *unique* views[3], the characteristics of which are shown in Table 2. We opted for unique views in order to examine the scalability and efficiency of our algorithms in the absence of trivial rewritings (where equivalent views rewrite one another) and force our utilization and latency optimizations algorithms to consider more complicated CFGs (rather than chains of equivalent views that can be easily optimized). All our experiments ran on an 8-core server (2 CPUs, Intel Xeon @2.93GHz), with 16GBs of RAM and running CentOS Linux 6.4.

5.2 EG and RG Generation

We have generated the EG using Algorithm 3, then removed cycles from it, and finally generated the RG using Algorithm 4. Algorithm 4 was instructed to generate no more than $k = 30$ rewritings for each view. The sizes and generation times for the EG and RG appear respectively in the middle and bottom of Table 2. Every time Algorithm 4 finds a rewriting, we create the corresponding $\wedge$ node, with an outgoing edge toward the rewritten view, and with an incoming edge from each view used in the rewriting. Table 2 shows that the number of rewritings (and thus, the size of the unrestricted RG) is very high, more than 2.5 millions.

5.3 CFG Utilization Optimization with ILP

We have set the upper bound of the data source as $B^{out}_{\mathcal{D}} = 6,198$, that is, the number of views that cannot be rewritten by other views (see Table 2) plus a 20% margin. We did this in order to push to the data source $\mathcal{D}$ the least possible load, while giving the ILP solver some margin to assign some extra views to $\mathcal{D}$ if needed. We have also set a common $B^{out} = \{30, 50, 100, \infty\}$ for all views (to see the effect of bounds on the shape of the resulting CFGs).

[3]We provide more experiments with a view set containing non-unique views in [14].

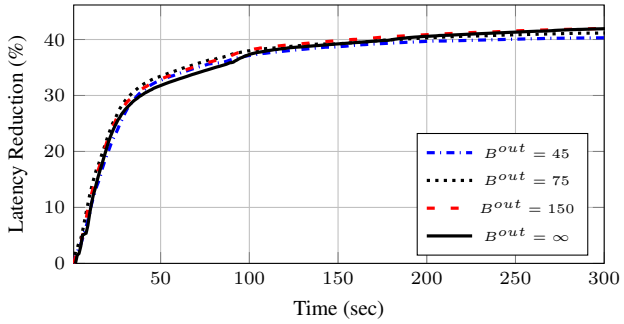


Figure 5: Latency reduction while running LOGA.

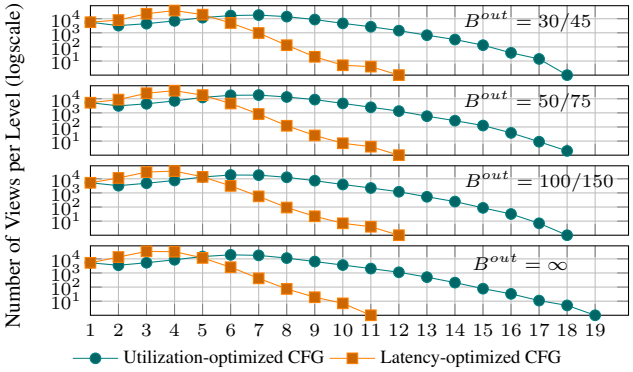


Figure 6: Distribution of views across CFG levels.

The Gurobi solver was then used to select utilization-optimal CFGs. A first observation was that the running time *decreases* as B^{out} increases, from about four minutes for $B^{out} = 30$ to less than two minutes for $B^{out} = \infty$. The reason is that a small B^{out} corresponds to highly restricted settings where the solver must search longer in order to find acceptable solutions.

Table 3 depicts the percentage of views rewritten using other views (and not filled from the data source $\mathcal{D}$) in the CFGs returned by the ILP solver, as well as the utilization of the CFGs and the average number of views that take part in the rewritings. First, notice that even when we keep the load on the views under tight control ($B^{out} = 30$), we achieve a high degree of off-loading (94.3%) from the data publisher $\mathcal{D}$. Moreover, as can be seen, by decreasing B^{out}, the utilization of the CFG increases (due to tighter constraints), while the number of views participating in a rewriting decreases (since each view is allowed to serve less views).

5.4 Greedy CFG Latency Optimization

We now study the performance of Algorithm 2 (LOGA, Section 3.4), applied on CFGs obtained through ILP optimization. Our initial experiments did not show significant latency improvement, because the ILP-selected CFGs exploited most of the freedom we gave them (almost every view was feeding B^{out} other views). Hence, there was very little leeway for LOGA to make changes. To circumvent this problem, we allowed LOGA to use as bound 1.5 times the B^{out} given to the ILP solver. Thus, where the ILP solver had $B^{out} = 30, 50, 100$, LOGA used $45, 75, 150$, respectively.

Latency optimization. Figure 5 depicts the latency improvement as a function of the LOGA running time. We see that LOGA is very effective, achieving a 43% reduction with respect to the latency of the CFG returned by the initial ILP solver. Moreover, such savings are obtained within 150-200 seconds. They stabilize when the data propagation paths to all the high-impact views have been altered and there is not much room for further optimization.

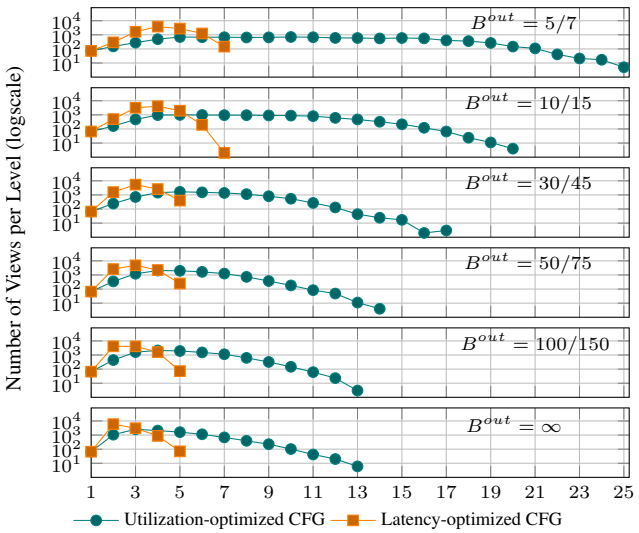


Figure 7: Distribution of views across deployed CFG levels.

Distribution of views into levels. Figure 6 depicts the distribution of views into levels in the CFGs for varying B^{out}, as produced (i) by the ILP solver, and (ii) after LOGA optimization. Note the logarithmic vertical axis. We see that the latency-optimized CFGs have less than 2/3 of the number of rewriting levels of the CFGs produced by ILP. Moreover, in the latency-optimized CFGs, most of the views lie in levels 1-6, leaving approx. only 1.5% of the views on levels 6-12. Thus, most views are only 4-5 hops away from the data source. This "flattening of rewriting levels" is an expected result of LOGA, since the more levels the data passes through from the publisher to a view, the more latency is added.

Utilization vs. latency. Although one may expect latency optimization (that reached 50%) to re-increase utilization, the increase was very moderate (5-7%). LOGA is only making greedy incremental fine-tuning over utilization-optimized CFGs (whose bounds were already attained), and therefore, the changes in the graph could not significantly change utilization.

5.5 Experiments in a WAN Deployment

We deployed Delta's algorithms on top of the distributed query execution engine of ViP2P [15], a large Java-based platform previously developed in our team. ViP2P provides a full set of continuous physical operators (structural joins, selections, etc.) which are used in Delta's rewritings. We report here on experiments we carried deploying Delta in a WAN.

Infrastructure. We conducted our experiments in the Grid5000 infrastructure (https://www.grid5000.fr), using 300 machines distributed over nine major cities across France and Luxembourg. The hardware of Grid5000 machines varies from dual-core machines with 2GBs of RAM to 16-core machines with 32GBs of RAM. This heterogeneous hardware distribution is likely to occur with real settings as subscribers have varied-capacity machines.

Views and documents. We have generated a set of 10,000 views, along with a set of 200 small (10-40KB) XMark [23] documents, in a way such that each document matches almost all of the views. Unlike our previous experiment, this view set has only ~3,000 unique views, which is more representative of real-life scenarios where some subscription topics are popular.

We have created the corresponding EG and RG and invoked the ILP solver to generate utilization-optimized configurations for $B^{out} \in \{5, 10, 30, 50, 100, \infty\}$ and $B^{out}_{\mathcal{D}} = 72$. The resulting

CFGs were optimized for latency with the LOGA Algorithm with bounds $\{7, 15, 45, 75, 150, \infty\}$.

The distribution of views into levels is depicted in Figure 7. A first observation is that in the presence of duplicate views, the latency optimized CFGs can have less than half of the levels of their utilization-optimized counterparts. A CFG with duplicate views is easier to optimize through the LOGA Algorithm since equivalent views may be served from one another.

We now move to presenting our results from deploying the generated CFGs. To characterize the performance of Delta, we have measured two important metrics, namely the *observed latency* and the *document delivery time*.

Observed view latency. We measured the latency of a view v for a document d as the time elapsed between: (i) the moment when the first tuple of d leaves the data source, and (ii) the instance when the last tuple of d reaches the view v. Note that in the observed view latency we do not include the time needed to extract the level 1 view tuples from a document. We do not include this[4] since this extraction step is not the main scope of the paper and has been studied in other works [7, 13].

Figure 8 depicts the average observed view latency for all pairs of views and documents in our CFGs. A first observation is that on average, views get their results in just 1.6 seconds after a document is published. This translates to a throughput of many thousands of subscriptions served per second, with a data source having to serve only ~0.7% (72 out of 10.000) of the views. This demonstrates how Delta makes it possible to serve large numbers of subscribers using very little publisher computing resources.

Our second remark regards the minimum/maximum latencies for $B^{out} = 5$ in utilization-optimized CFGs. Some views in the network receive their results extremely fast (~30ms) while some others considerably slower (~3.7s). This is an inherent feature of Delta: views that are close to the data source receive their data faster than the ones that reside in deeper levels.

The LOGA algorithm reduces the observed latency of views up to ~20% ($B^{out} = 5$) compared to the utilization-optimized CFGs. This also shows that our latency estimation models (used by our algorithms) are quite accurate.

An interesting phenomenon is the following: in the utilization-optimized CFG where $B^{out} = \infty$ we notice a very large increase in the maximum latency (~4.7s) while the CFG is not too deep (13 levels) compared to other CFGs that showed lower latency. This is explained by the fact that when a view serves a very large number of other views, it can be overloaded and the data processing/transmission throughput is reduced. This shows the importance of the bounds B^{out} in Delta: for optimal performance, B^{out} must be set in the "sweet spot" between values too large (to avoid overloading) and too low (to avoid very deep CFGs). In practice, a simple test can be performed at each subscriber machine to tailor its B^{out} to its observed hardware performance.

Document Delivery Time. For a view v and a document d that matches v, we term *document delivery time*, or simply DDT, the total time needed for all the matching tuples of document d to reach the view v. For a set of views V, the DDT is measured as the interval between: (i) the moment when the first tuple of the document d *leaves* the data source and (ii) the instance when the last tuple of the document d has reached the slowest view $v \in V$. In other words, this metric captures the time it takes for a document to reach its slowest interested view.

[4]For completeness: our view matcher took an average of ~100ms to extract from each document the tuples for the 72 first-level views.

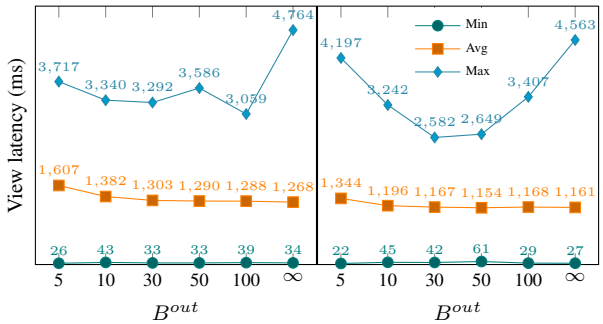


Figure 8: View latency for utilization-optimized CFGs (left) and latency optimized CFGs (right).

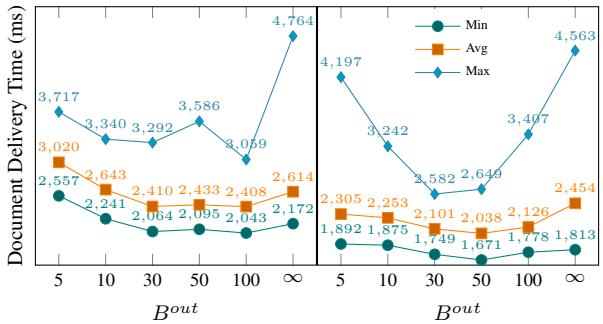


Figure 9: Document delivery time for utilization-optimized CFGs (left) and latency optimized CFGs (right).

Figure 9 shows the average, minimum and maximum DDT over all published documents in our experiment. In general, in all CFGs, a document is delivered to all views in the network, in an average of 2-2.5 seconds. Note that the maximum observed latency coincides with the maximum DDT (see Figures 8 and 9) as the slowest view in the network actually defines the DDT. Thus, we observe the same phenomenon as in the observed latency: DDT slows down for the extreme $B^{out} = \{5, \infty\}$ values.

5.6 Experiment Conclusion

Our experiments have demonstrated the efficiency and effectiveness of Delta's multi-level dissemination approach. With respect to efficiency, for 100,000 distinct subscriptions, the full graph generation, optimization for utilization and then latency took less than 13 minutes. As for effectiveness, the configurations retained have low cost scores. This is confirmed by the WAN deployment of 10,000 subscriptions, which showed a high message delivery throughput and low latency: documents are propagated to 10,000 subscriptions, which are fed with data within 1.5 seconds on average.

6. RELATED WORK

Our work belongs to the class of content-based publish subscribe systems, disseminating to users the results of their specified subscriptions over a stream of published data. This paper is related to several themes of existing works.

Filtering systems. A large part of the literature addresses the problem of optimizing the publisher so that it handles the filtering of incoming data for very large numbers of subscribers.

YFilter [7] stands out as a widely-known system for XML publish-subscribe. It is able to feed many XPath 1.0 subscriptions very efficiently by matching them simultaneously against documents through a single automaton. NiagaraCQ [4] relies on multi-query optimization for continuous queries, taking advantage of the similarity of subscriptions in order to share operators during evaluation. Similarly, [13] addressed the same problem but for a more expressive

subscription language, supporting joins over multiple documents. Finally, [27] proposes a pub/sub system where the evaluation of subscriptions is done inside a relational database.

The above do not consider distributed data dissemination. Instead, they focus on optimizing the publisher task, to support very large numbers of subscribers. Our work can be seen as complementary since we focus on the design of a logical overlay network (CFG), that exploits the subscribers in order to scale up. Any efficient filtering at the publisher can be adopted in our setting.

Distributed publish/subscribe. Onyx [8] connects multiple publishers and subscribers by employing multiple YFilter instances running on connected brokers. Recently, FoXtrot [19] has distributed YFilter automata on top of a DHT network. Other DHT-based pub/sub systems are, e.g., [5, 10]. Closer to our work, SemCast [20] leverages commonalities between subscriptions and creates logical *channels* between brokers and subscribers to form multicast trees of low utilization and latency. However, the system relies on a network of brokers, and the subscribers do not help in the dissemination of data. Finally, [26] builds one multicast tree per broker aiming at redundancy and fault tolerance.

Contrariwise, in [2], *every peer* can forward messages to its neighbors if the message matches its own interests. Peers are organized in an hierarchy tree based on subscription similarity. However, by design, the peers do not know the subscriptions of their neighbors, and as a result, their routing protocol allows for false positives (peers may receive messages which do not interest them).

In contrast with these works, Delta builds multi-level dissemination networks involving the subscribers, leveraging query rewriting to determine whether some subscriptions can be used to compute results of other subscriptions. One of the consequences unique to Delta is the ability to combine the results of *multiple* subscriptions in order to serve another one.

View-based data management. As explained in Section 4.4, any efficient view-based rewriting algorithm (e.g., [21]) can be used instead of our Algorithm 4. View maintenance has been investigated in the centralized context of data warehousing [24, 22]. In [6], the authors consider "stacked" views, specified as queries over other defined views, study their maintenance and the efficient evaluation of queries using such views; these resemble our multi-level configurations, but in [6] the connections between views are given, whereas we choose them for performance through our algorithms.

7. CONCLUSION

We considered the problem of scaling up content-based publish/subscribe systems under resource constraints (such as finite CPU and network capacity) by off-loading some of the data publisher's effort on the subscriber sites. This is achieved by organizing subscriptions in a rewritability graph which materializes the ways in which one subscription could be served from others, through view-based rewriting. We provide a novel two-step algorithm for organizing the views in a network minimizing a combination of resource utilization and data dissemination latency. First, we express the utilization minimization problem as a linear program and solve it exactly; as we show, latency cannot be included in the ILP formulation due to its non-linear nature. We reduce latency in a second step based on the result obtained from the ILP solver. Our configuration choice algorithm scale well to 100.000 unique subscriptions, whereas in a WAN deployment, Delta succeeds in filling in 10.000 subscriptions with a latency of under 2 seconds.

Acknowledgments. The authors would like to thank Cédric Bentz for his valuable help and discussions on the ILP modeling and Yannis Manoussakis for his input in the proof of NP-Hardness.

8. REFERENCES

[1] N. Bansal, R. Khandekar, and V. Nagarajan. Additive guarantees for degree-bounded directed network design. *SICOMP*, 2009.

[2] R. Chand and P. Felber. Semantic peer-to-peer overlays for publish/subscribe networks. In *Euro-Par*, 2005.

[3] A. K. Chandra and P. M. Merlin. Optimal implementation of conjunctive queries in relational data bases. In *STOC*, 1977.

[4] J. Chen, D. DeWitt, F. Tian, and Y. Wang. NiagaraCQ: A scalable continuous query system for internet databases. In *SIGMOD Rec.*, 2000.

[5] P. Chirita, S. Idreos, M. Koubarakis, and W. Nejdl. Publish/Subscribe for RDF-based P2P networks. In *ESWS*, 2004.

[6] D. DeHaan, P.-A. Larson, and J. Zhou. Stacked Indexed Views in Microsoft SQL Server. In *SIGMOD*, 2005.

[7] Y. Diao, M. Altinel, M. J. Franklin, H. Zhang, and P. Fischer. Path sharing and predicate evaluation for high-performance XML filtering. *TODS*, 2003.

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

[9] P. Eades, X. Lin, and W. Smyth. A fast and effective heuristic for the feedback arc set problem. *Information Processing Letters*, 1993.

[10] A. Gupta, O. Sahin, D. Agrawal, and A. Abbadi. Meghdoot: Content -based Publish/Subscribe over P2P networks. In *Middleware*, 2004.

[11] H. Gupta. Selection of views to materialize in a data warehouse. In *ICDT*, 1997.

[12] J. Hoffmann and J. Koehler. A new method to index and query sets. In *JCAI*, 1999.

[13] M. Hong, A. Demers, J. Gehrke, C. Koch, M. Riedewald, and W. White. Massively multi-query join processing in publish/subscribe systems. In *SIGMOD*, 2007.

[14] K. Karanasos, A. Katsifodimos, and I. Manolescu. Delta: Scalable Data Dissemination under Capacity Constraints. Inria Research Report N°8385, October 2013.

[15] K. Karanasos, A. Katsifodimos, I. Manolescu, and S. Zoupanos. ViP2P: Efficient XML management in DHT networks. In *ICWE*, 2012.

[16] A. Katsifodimos, I. Manolescu, and V. Vassalos. Materialized view selection for XQuery workloads. In *SIGMOD*, 2012.

[17] L. C. Lau, J. S. Naor, M. R. Salavatipour, and M. Singh. Survivable network design with degree or order constraints. *SICOMP*, 2009.

[18] I. Manolescu, K. Karanasos, V. Vassalos, and S. Zoupanos. Efficient XQuery rewriting using multiple views. In *ICDE*, 2011.

[19] I. Miliaraki and M. Koubarakis. Foxtrot: Distributed structural and value XML filtering. *ACM TWEB*, 2012.

[20] O. Papaemmanouil. SemCast: Semantic multicast for content-based data dissemination. In *ICDE*, 2005.

[21] R. Pottinger and A. Y. Halevy. MiniCon: A scalable algorithm for answering queries using views. *VLDB J.*, 10(2-3), 2001.

[22] K. A. Ross, D. Srivastava, and S. Sudarshan. Materialized view maintenance and integrity constraint checking: trading space for time. In *SIGMOD*, 1996.

[23] A. Schmidt, F. Waas, M. L. Kersten, M. J. Carey, I. Manolescu, and R. Busse. XMark: A Benchmark for XML Data Management. In *VLDB*, 2002.

[24] A. Segev and W. Fang. Currency-based updates to distributed materialized views. In *ICDE*, 1990.

[25] N. Tang, J. X. Yu, M. T. Özsu, B. Choi, and K.-F. Wong. Multiple Materialized View Selection for XPath Query Rewriting. In *ICDE*, 2008.

[26] W. W. Terpstra, S. Behnel, L. Fiege, A. Zeidler, and A. P. Buchmann. A peer-to-peer approach to content-based publish/subscribe. In *DEBS*, 2003.

[27] F. Tian, B. Reinwald, H. Pirahesh, T. Mayr, and J. Myllymaki. Implementing a scalable XML publish/subscribe system using relational database systems. In *ACM SIGMOD*, 2004.

[28] X. Wu, D. Theodoratos, and W. H. Wang. Answering XML queries using materialized views revisited. In *CIKM*, 2009.

[29] Gurobi Optimizer. `http://www.gurobi.com`, 2013.