Optimizing large collections of continuous content-based RSS aggregation queries

Jordi Creus[1], *Bernd Amann*[1], *Vassilis Christophides*[2], *Nicolas Travers*[3], *and Dan Vodislav*[4]

[1]*LIP6, CNRS – Université Pierre et Marie Curie, Paris, France*
[2]*ICS-FORTH – University of Crete, Crete, Greece*
[3]*Cedric/CNAM – Conservatoire National des Arts et Métiers, Paris, France*
[4]*ETIS, CNRS – University of Cergy-Pontoise, Cergy, France*

Abstract

In this article we present *RoSeS* (Really Open Simple and Efficient Syndication), a generic framework for content-based RSS feed querying and aggregation. *RoSeS* is based on a data-centric approach, using a combination of standard database concepts like declarative query languages, views and multi-query optimization. Users create personalized feeds by defining and composing content-based filtering and aggregation queries on collections of RSS feeds. Publishing these queries corresponds to defining views which can then be used for building new queries / feeds. This naturally reflects the publish-subscribe nature of RSS applications. The contributions presented in this article are a declarative RSS feed aggregation language, an extensible stream algebra for building efficient continuous multi-query execution plans for RSS aggregation views, a multi-query optimization strategy for these plans and a running prototype based on a multi-threaded asynchronous execution engine.

1 Introduction

Web 2.0 technologies have transformed the Web from a publishing-only environment into a vibrant information place where yesterday's passive readers have become active information collectors and content generators themselves. Web 2.0 contents is generally evolving very rapidly in time and can best be characterized by a stream of information entities. The proliferation of content generating Web 2.0 applications obviously yields many new opportunities to collect, filter, aggregate and share information streams on the web. Given the amount and diversity of the information generated on a daily base in Web 2.0, it is unprecedented and creates a vital need for efficient continuous information processing methods which allow users to effectively follow personally interesting information.

Web syndication formats have emerged as a popular mean for timely delivery of frequently updated Web content. According to the RSS (or Atom) format, information publishers provide brief summaries (i.e. textual snippets) of the content they deliver on the Web, while information consumers subscribe to a number of RSS/Atom feeds (i.e., streams) and get informed

The authors acknowledge the support of the French Agence Nationale de la Recherche (ANR), under grant ROSES (ANR-07-MDCO-011) "Really Open, Simple and Efficient Syndication"

about newly published information items. Today, almost every personal weblog, news portal, or discussion forum and user group employs RSS/Atom feeds for enhancing traditional *pull-oriented* searching and browsing of web pages with *push-oriented protocols* of web content. Furthermore, social media (e.g., Facebook, Twitter, Flicker) also employ RSS for notifying users about the newly available posts of their preferred friends (or followers).

RSS/Atom intermediaries such as GoogleReader[1], Yahoo Pipes[2] as well as several other aggregators (feedrinse.com, newsgator.com, bloglines.com, or pluck.com) allow users to assign a set of keywords to a specific RSS feed they have already subscribed to. Whenever a new item is fetched for that specific feed, the aggregator tests if all of the terms specified are also present in the item, and notifies the user accordingly. In a similar way, Google[3] and Yahoo[4] alerting services enable to filter out syndicated content that is not of interest to users and notify them by email. However, both RSS/Atom aggregators and alerters provide content filtering facilities on the items essentially in a pull mode by periodically downloading the corresponding RSS documents (e.g. every 2-5 hours) according to a predefined refreshing policy (i.e. they are not continuous queries). Clearly, this functionality can be used only when the number of feeds is limited to a few authority sites (such as news agencies and newspapers) and not to a myriad of feeds bounded to citizen journalism (such as personal blogs, discussion forums and social media). It is estimated that more than 60M blogs (without considering discussion forums and user groups) actually exist compared to around 5000 professional news sources tracked periodically by Google News, Yahoo! News and MSN News. Moreover, modest estimations places the total number of user accounts on various social media around 230M. On the other hand, news search engines (or even blogs search engines) enable content filtering without enforcing users to know a-priori the information sources offering RSS/Atom feeds. However, storing web content locally in a warehouse implies heavy intellectual property rights management especially for professional sources (e.g., press organizations) whose cost could be prohibiting for small and medium enterprises.

In this paper we present *RoSeS* (Really Open Simple and Efficient Syndication), a generic framework for content-based RSS feed querying and aggregation. It relies on a data-centric approach, supporting expressive continuous queries (selection, join, union) over textual and factual information streams, along with RSS publishing views capable of merging and filtering RSS information items originating from a potential large number of feeds (for example from all major French journals). Following a database approach, these views can be reused for building other streams and the final result is an acyclic graph of union/filtering queries on all the involved feeds. These query graphs generally grow very rapidly and call for efficient multi-query optimization strategies to reduce the evaluation cost.

RoSeS is based on a simple but expressive data model and query language for defining continuous queries on RSS streams. Combined with efficient web crawling strategies and multi-query optimization techniques, *RoSeS* can be used as a continuous RSS stream middle-ware for dynamic information sources. The main contributions presented in this article are:

- A declarative RSS feed aggregation language for publishing large collections of structured queries/views aggregating RSS feeds,

[1] www.google.com/reader
[2] pipes.yahoo.com
[3] www.google.com/alerts
[4] alerts.yahoo.com

- An extensible stream algebra for building efficient continuous multi-query execution plans for RSS streams,
- A flexible and efficient cost-based multi-query optimization strategy for optimizing large collections of publication queries,
- A running prototype based on multi-threaded asynchronous execution of query plans.

The rest of the article is organized as follows. Section 2 describes the overall *RoSeS* architecture. The *RoSeS* data model, language and algebra are presented in Section 3 and correspond to the first important contribution of our article. Section 4 is devoted to query processing and our cost-based multi-query optimization approach based on finding Steiner minimal-cost Trees in weighted filter subsumption graphs. In Section 5 we present an algorithm (VCA) and data structure (VCB) for computing such trees efficiently. VCA and VCB are experimentally evaluated in Section 6. Related work is presented in Section 7 and Section 8 discusses future work.

2 *RoSeS* Architecture

The *RoSeS* system is composed of five modules for processing RSS feeds and managing meta-data about users, publications and subscriptions. As shown in Figure 1, RSS feeds are processed by a three layered architecture where the top layer (acquisition) is in charge of crawling the collection of registered RSS feeds (in the following called source feeds), the second layer (evaluation) is responsible for processing a continuous *query plan* which comprises all publication queries and the third layer (dissemination) deals with publishing the results according to the registered subscriptions (see Section 3). The remaining two modules (catalog and system manager) provide meta-data management services for storing, adding, updating and deleting source feeds, publication queries and subscriptions.

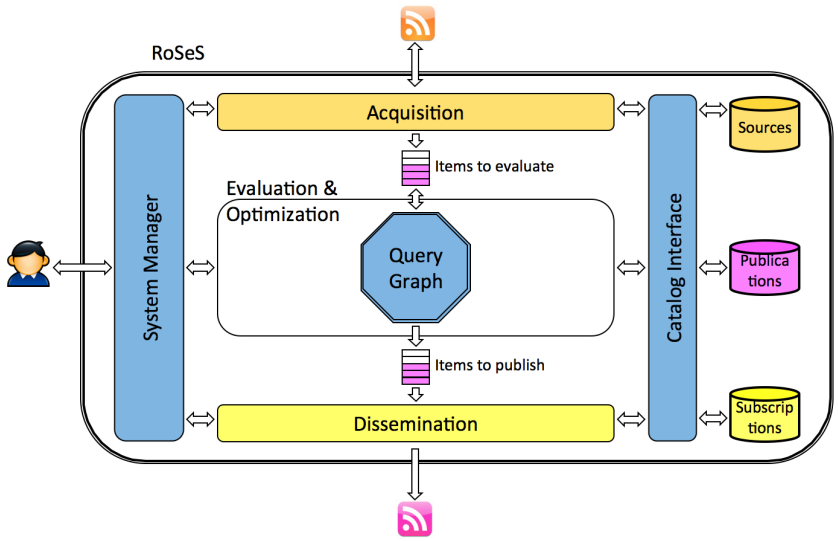

Figure 1: RoSeS System architecture

Acquisition: The main task of this module is to transform evolving RSS documents into a continuous stream of RoSeS items which can be processed by the evaluation module. This transformation includes an efficient refresh strategy optimizing the bandwidth usage. In [**?**], we

propose a best-effort strategy for refreshing RSS documents under limited bandwidth, which introduces the notion of saturation for reducing information loss below a certain bandwidth threshold. The freshness efficiency is not in the scope of this paper.

Evaluation: The core of the RoSeS system is an *algebraic multi-query plan* which encodes all registered publication queries. The evaluation of this query plan follows an asynchronous pipe-lined execution model where the evaluation module (1) continuously evaluates the set of algebraic operations according to the incoming stream of RoSeS items, and (2) manages the addition, modification and deletion of publication queries.

Dissemination: This module is responsible for transforming RoSeS items into different output formats and notifying new items to corresponding user subscriptions. The goal of this module is to define the way items are rewritten, in given formats (SMS, email, RSS/Atom feed...).

3 *RoSeS* Data Model and Language

3.1 The *RoSeS* language

The *RoSeS* data manipulation language provides the main functionality for *registering* new source feeds, for *publishing* virtual feeds through queries, and for *declaring* subscriptions to feeds. We focus in a first place on the feeds publication language, and then briefly describe how we can register source feeds and subscribe to both source and virtual feeds.

RoSeS publication language has been designed to address several desiderata: to be expressive but simple to use, to facilitate most common aggregation and filtering operations while be amenable to optimization in order to support real scale web syndication systems. The most frequently used form of aggregation is to union items originating from a large number of RSS feeds and then to filter them using Boolean conditions especially in textual content. Additionally, a unique feature of the *RoSeS* publication language is its ability to (semi-)join the items of different feeds by keeping track of the matching items under the form of annotations. In a nutshell, a publication query contains three clauses: (a) A mandatory `from` clause, which specifies the primary feeds that will produce the items of the so-defined *virtual feed*; (b) Zero, one or more `join` clauses, each one specifying a join with a secondary feed; (c) An optional `where` clause for filtering conditions on primary or secondary feeds.

Consider, for example, that Bob is interested in organizing a social event with his friends which includes attending a rock concert. He publishes a virtual feed named *RockConcertStream*, with items about concerts originating from his friends feeds (*FriendsFacebookStream* and *FollowedTwitterStream*), as well as rock concert announces from feed *EventAnnounces*. Notice that the `from` clause allows users to build nested named groups of feeds (unions identified by variables), which can be referenced in the `where` clause for adding some filtering conditions. Here filtering conditions are expressed on an individual feed *EventAnnounces* represented by variable $ca (title contains "rock") and on the group of all named feeds in the `from` clause represented by variable $r (description contains "concert"):

```
create feed RockConcertStream
from (FriendsFacebookStream | EventAnnounces as $ca | FollowedTwitterStream) as $r
where $ca[title contains 'rock'] and $r[description contains 'concert'];
```

Then Bob, who is a fan of the Muse rock band, publishes a virtual feed *MusePhotoStream* with items about this band annotated with related photos. Items come from feeds *RockCon-*

certStream (those talking about "Muse") and *MuseNews*, while photos originate from two secondary feeds: *FriendsPhotos* with photos published by his friends and *MusicPhotos* (only for category "rock"). Feeds annotation is realized by the join operation, where in our example each item from the primary feed ($main) is annotated with the photos from items of the secondary feed having similar titles during the last three months. Notice that a join specifies a window operation on a group of secondary feeds, a primary feed (through a variable) and a join predicate.

```
create feed MusePhotoStream
from (RockConcertStream as $r | MuseNews) as $main
join last 3 months on (MusicPhotos as $m | FriendsPhotos)
                    with $main[title similar window.title]
where $r[description contains 'Muse'] and $m[category = 'rock'];
```

The *RoSeS registration language* allows declaring source feeds coming either from existing RSS/Atom feeds on the Web, or virtual feeds internally materialized. In particular, for virtual feeds published by queries is also possible to employ XSLT stylesheets in order to transform the structure of items. Transformations may also use annotations produced by joins, e.g. include corresponding links to photos of feed *MusePhotoStream* at materialization time. To simplify optimization, transformation operations were not included in the *RoSeS* publication language for specifying the virtual feeds. The following example illustrates how we can register in *RoSeS* a source feed (*http://muse.mu/rss/news*) and materialize a virtual feed (*MusePhotoStream*) featuring a transformation ("IncludePhotos.xsl").

```
register feed http://muse.mu/rss/news as MuseNews;
register feed MusePhotoStream apply 'IncludePhotos.xsl' as MuseWithPhotos;
```

The *RoSeS subscription language* allows declaring subscriptions to source or virtual feeds. A subscription essentially specifies a feed, a notification mode (RSS, mail, etc.), a periodicity and possibly an item transformation. Subscription transformations are also expressed by XSLT stylesheets but unlike transformations during feeds' registration, the output format is free. The following example depicts two subscriptions to the *RockConcertStream* publication: the first one extracts item titles ("Title.xsl" transformation) and sends them by mail every three hours, the second one simply outputs an RSS feed refreshed every ten minutes.

```
subscribe to RockConcertStream apply ''Title.xsl'' output mail ''me@mail.org'' every 3 hours;
subscribe to RockConcertStream output file ''RockConcertStream.rss'' every 10 minutes;
```

3.2 Data model and algebra

The *RoSeS* data model borrows from state-of-the-art data stream models, while proposing specific modeling choices adapted to RSS/Atom syndication and aggregation.

A RoSeS feed corresponds to either an existing RSS/Atom (*source feed*) published on the Web, or to a *virtual feed* published through queries in *RoSeS*. Formally, a registered feed is a couple $f = (d, s)$, where d is the *feed descriptor* and s is a stream of *RoSeS items*. Feed descriptor d is a tuple, representing usual RSS/Atom feed properties: title, description, URL, etc. *RoSeS* items represent the information content conveyed by RSS/Atom feeds. Despite the adoption of an XML syntax, RSS and Atom formats are mainly used with flat text-oriented content. Extensions and deeper XML structures are very rarely used, therefore we made the choice of a flat representation, as a set of typed attribute-value couples, including common RSS/Atom item

properties like title, description, link, author, publication date, etc. Extensibility may be handled by including new, specific attributes to *RoSeS* items – this enables both querying any feed through the common attributes and addressing specific attributes (when known) of extended feeds.

A RoSeS feed is a fully-fledged data stream of annotated *RoSeS* items. Formally, a *RoSeS* stream is a (possibly infinite) set of *RoSeS elements* $e = (t, i, a)$, where t is a timestamp, i a *RoSeS item*, and a an annotation, the set of elements for a given timestamp being finite. Annotation a refers to all items that have been joined with element e: an *annotation* is a set of couples (j, A), where j is a *join identifier* and A is a set of items. The semantics of annotation is further detailed in the join operator below.

A RoSeS window captures subsets of a stream's items that are valid at various time periods. Formally, a window w on a stream s is a finite set of couples (t, I), where t is a timestamp and I is the set of items of s valid at t. Note that (i) a timestamp may occur only once in w, and (ii) I contains only items that occur in s before (or at) timestamp t. We note $w(t)$ the set of items in w for timestamp t. Windows are used in *RoSeS* only for computing joins between streams. *RoSeS* uses two types of sliding windows: time-based (last n units of time) and count-based (last n items).

Publication queries of virtual feeds are based on *five operators* for composing streams of *RoSeS* items. We distinguish between *conservative* operators (filtering, union, windowing, join), that do not produce new items or change the contents of input items, and *altering* operators (transformation) on items. A central design choice for the *RoSeS* publication language is to *rely only on conservative operators*. This choice is justified by the fact that conservative operators dispose good query rewriting properties (commutativity, distributivity, etc.) and thus favor both query optimization and language declarativeness (any algebraic expression can be rewritten in a normalized form corresponding to the declarative clauses of the language). Publication queries are then translated to filtering, union, windowing and join operators. Next, transformation can be applied over materialized virtual feeds. Notice that transformations may use join annotations, and consequently enrich (indirectly) the expressive power of joins. Publication queries (for which subscriptions exist) are translated into algebraic expressions as shown in the following example for *RockConcertStream* and *MusePhotoStream* (for space reasons abbreviated feed names and a simplified syntax are used).

$$\begin{aligned} RConcert &= \sigma_{'concert' \in desc}(FFacebook \cup \sigma_{'rock' \in title}(EAnnounces) \cup FTwitter) \\ MPhoto &= (\sigma_{'Muse' \in desc}(RConcert) \cup MNews) \bowtie_{title \sim w.title} \\ &\quad \omega_{last\,3\,m}(\sigma_{category='rock'}(MPhotos) \cup FPhotos) \end{aligned}$$

The algebra is defined in the rest of this section and the evaluation of algebraic expressions is detailed in section 4. The *filtering* operation σ_P outputs only the stream elements that satisfy a given item predicate P:

$$\sigma_P(s) = \{(t, i, a) \in s |\ P(i)\}$$

Item predicates are Boolean expressions (using conjunctions, disjunctions, negation) of atomic item predicates that express a condition on an item attribute; depending on the attribute type, atomic predicates may be:

- for simple types: comparison with value (equality, inequality).
- for date/time: comparison with date/time values (or year, month, day, etc.).

- for text: operators *contains* (word(s) contained into a text attribute), *similar* (text similar to another text).
- for links: operators *references/extends* (link references/extends an URL or host), *shares-link* (attribute contains a link to one of the URLs in a list).

Note that *RoSeS* allows applying text and link predicates *to the whole item* - in this case the predicate considers the whole text or all the links in the item's attributes. Observe also that it is not possible to filter stream elements by their timestamp or annotation attributes.

Union outputs all the elements in the input streams:

$$\bigcup(s_1, ..., s_n) = \{i |\ i \in s_1 \vee ... \vee i \in s_n\}$$

Windowing produces a time-based or a count-based sliding window on the input stream according to the window specification:

$$\omega_{spec}(s) = \{i |\ i \in s \wedge spec(i, s)\}$$

where $spec$ expresses an upper bound on the items publication date (time-based window), respectively on the items position in the stream (count-based window).

Join takes a primary stream and a window on a (secondary) stream. *RoSeS* uses a conservative variant of the join operation, called *annotation join*, that acts like a semi-join (primary stream filtering based on the window contents), but keeps trace of the joining items under the form of an annotation entry. A join $\bowtie_P (s, w)$ of an item j outputs the elements of s for which the join predicate P is satisfied by a non-empty set I of items in the window, and adds to them the annotation entry (j, I). More precisely,

$$\bowtie_P (s, w) = \{(t, i, a') |\ (t, i, a) \in s, I = \{i' \in w(t) |\ P(i, i')\}, |I| > 0, a' = a \cup \{(j, I)\}\}$$

Transformation modifies each input element following a given transformation function:

$$\tau_T(s) = \{T(t, i, a) |\ (t, i, a) \in s\}$$

τ_T is the only altering operator, whose use is limited to produce subscription results or new source feeds, as explained above.

4 Query Processing

4.1 Query graphs

Query processing consists in continuously evaluating a collection of publication queries. This collection is represented by a *multi-query plan* composed of different physical operators reflecting the algebraic operators presented in Section 3.2 (union, selection, join and window). *RoSeS* adopts a standard execution model for continuous queries [**?**, **?**] where a query plan is transformed into a graph connecting sources, operators and publications by inter-operator queues or by window buffers (for blocking operators like join).

A query plan for a set of queries Q can then be represented as a directed acyclic graph $G(Q)$. Figure 2 illustrates one out of the several possible query plans for three publications $p_1 = \sigma_1(s_1 \cup s_2)$, $p_2 = (s_3 \cup s_4) \bowtie_1 \omega_1(s_5)$, $p_3 = \sigma_2(p_1 \cup s_6)$. As we can see, window operators produce a different kind of output, namely window buffers, which are consumed by

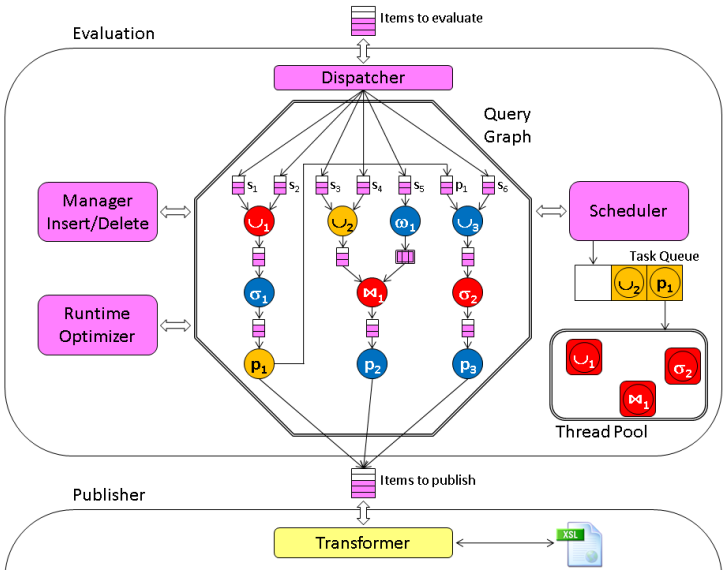

Figure 2: Architecture of the *RoSeS* Evaluation Engine

join operators. View composition by publish/subscribe is illustrated by an arc connecting a publication operator to an algebraic operator (p_1 is used as input by publication p_3). Observe also that all transformations are applied after publication.

The next section presents query processing and the underlying cost-model. Section 4.3 introduces our optimization techniques to improve processing performances.

4.2 Query evaluation and cost-model

A multi-query plan is composed of operators connected by read/write queues or by window buffers. New items are continuously arriving to this graph and have to be consumed by the different operators. In this respect, we have adopted a multi-threaded, pipelining execution mode of continuous queries. Query execution is performed as follows. The query graph is observed by a *scheduler* that continuously decides which operators (tasks) must be executed (see Figure 2). The scheduler has at his disposal a pool of threads for executing in parallel a fixed number of tasks[5]. The choice of an inactive operator to be evaluated is influenced by different factors depending on the input buffer of each operator (the number and/or age of the items in the input queue).

In this setting, we rely on a cost model for estimating the resources (memory, processor) necessary for the execution of a query plan. Compared to the cost estimation of a snapshot query plan which is based on the size of the input data, the estimation parameters of a continuous query plan must reflect the streaming nature of the data. We adapt a simplified version of the model presented in [**?**] and define the cost of each operator op as a function of the publishing rate $R(b)$ of its input buffer(s) b (and the size $S(w)$ of input windows w, for join operators).

As we can see in Table 1, the cost of each operator mainly depends on the publishing rate $R(b)$ of its input buffer(s) b (b_i). We assume a constant execution cost per item (independent of the item contents) for the filter operator since items are in general small text fragments of similar size. Memory cost of filtering is also constant because it processes only one item at a time. The publishing rate of the selection operator reduces the rate of its input buffer by the selectivity factor $sel(p) \in [0, 1]$ depending on the selection predicate p. Union generates an

[5]The naive solution of attaching one thread to each operator rapidly becomes inefficient / impossible due to thread management overhead or system limitations.

Operator	Output rate	Memory	Processing cost
$\sigma_p(b)$	$sel(p) * R(b)$	$const$	$const * R(b)$
$\cup(b_1, ..., b_n)$	$\sum_{1 \leq i \leq n} R(b_i)$	0	0
$\bowtie_p (b, w)$	$sel(p) * R(b)$	$const$	$R(b) * S(w)$
$\omega_d(b)$	0	$S = const$ or $S = d * R(b)$	$const$

Table 1: The *RoSeS* Cost Model

output buffer with a publishing rate corresponding to the sum of its input buffer rates[6]. We assume zero memory and processing cost since, compared with the other operators, union is simply implemented by a set of buffer iterators, one for each input buffer. The join operator generates an output buffer with a publishing rate of $sel(p) * R(b)$ where $R(b)$ is the publishing rate of the primary input stream and $sel(p) \in [0, 1]$ corresponds to the probability that an item in b joins with an item in window w using join predicate p. This is due to the behavior of the annotation join: one item is produced by the join operator when a new item arrives on the main stream and matches at least one item in the window. Then the item is annotated with all matching window items. So the processing cost of the join operator corresponds to the product $R(b) * S(w)$, where $S(w)$ is the size of the join window w. The window operator transforms the stream into a window buffer where the size depends on the specified number of items (count-based window) or on the time-interval d and the input buffer rate $R(b)$ (time-based window). It is easy to see that the global cost of the execution plan (the sum of the cost of all operators) is mainly influenced by the operator ordering and the input buffer rate of each operator. We will describe in the following section how we can reduce this cost by pushing selections and joins towards the source feeds of the query plan.

4.3 Query optimization

The main novelty of our framework with respect to other multi-query optimization solutions lies in the explicit usage of a cost model for continuous queries. The *RoSeS* cost model (Table 1) indicates that the execution cost of most operators is proportional to the input rate. We exploit two main ideas, common to other optimization approaches, but guided in *RoSeS* by the cost model: (i) rapidly decrease the input rate by first applying filtering (then join), and (ii) factorize common operators among publications.

The optimization process is decomposed into two main phases:

- a *normalization* phase which applies rewriting rules for pushing all filtering operators towards their source feeds,
- a *factorization* phase of the selection predicates of each source based on a new cost-based factorization technique not previously reported in the literature.

We will describe the whole optimization process in the following.

Query normalization: The goal of the first phase is to obtain a normalized query plan where all filtering selections are evaluated first to each source before evaluating joins and unions. This is possible by iteratively applying (a) distributivity of selections on unions and (b) commutativity between filtering and join (observe that filtering operations cannot be applied on annotations generated by joins). It is possible to show that under *snapshot semantics* and by

[6]In our current implementation union does not remove duplicates. For removing duplicates we have to add a sliding window of fixed size and check for each new item if it already appears in this window.

applying rewriting rules (a) and (b) we can obtain an equivalent query plan which is a four level tree: the first level (leaves) of the tree are the source feeds involved in the query, the second level nodes involves filtering operators which have to be applied to each source feed, the third level comprises the window/join operators evaluated over the results of the selections and the final (fourth) level have the unions evaluated over the results of the selections/windowed-joins. Normalization also flattens all cascading filtering paths into elementary filtering predicates under a conjunctive normal form (CNF). It should be stressed that normalization might increase the cost of the resulting query graph with respect to the original one. However, as we will see, this happens only temporary since the subsequent factorization phase after query normalization has more optimization opportunities.

A simple example for the normalization of three publications without joins is shown in Figures 3 and 4. Publication p_3 ($p_3 = \sigma_d(p_2 \cup s_5)$) is defined over another publication p_2, with a filtering operation ($\sigma_{a \wedge c}$). Here, the normalization process consists in pushing all filtering operations through the publication tree to the sources for obtaining a normalized plan as shown in Figure 4.

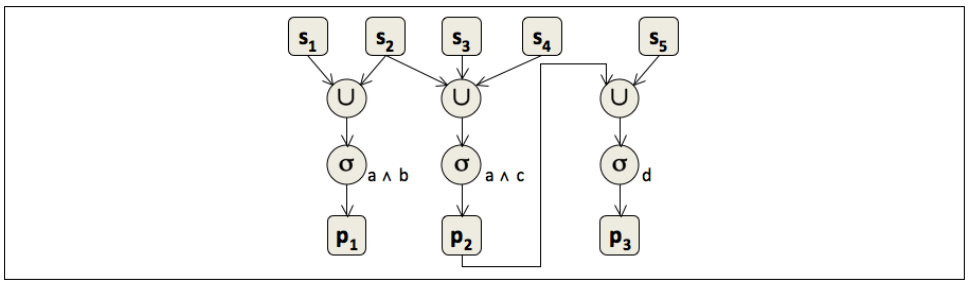

Figure 3: A Query Plan for Publications p_1, p_2 and p_3

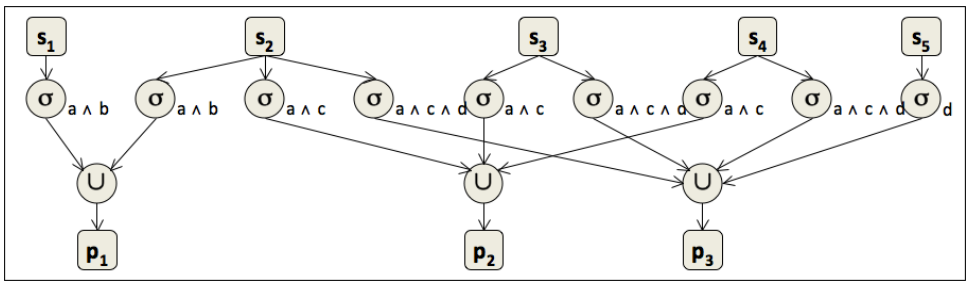

Figure 4: Normalized Query Graph in *RoSeS*

Query factorization: In this paper we focus on *filtering factorization*, the most effective optimization technique in our context. Normalization generates a global query plan where each source s is connected to a set of selection predicates $P(s)$. Factorization considers each source separately and consists in building an efficient *filtering plan* for it. To find the best operator factorization we proceed in two steps: we first generate for each source feed s a *predicate subsumption graph* which contains *all* predicates subsuming the set of predicates in $P(s)$. The weight of each subsumption edge in this graph corresponds to the output rate of the node it originates (source or filtering operation) and expresses *the cost* of the node it targets. It is easy to see that any sub-tree of this graph covering the source s (root) *and all predicates in* $P(s)$ corresponds to a filtering plan that is equivalent to the original plan. The cost of this plan is the sum of the costs of the tree edges. The optimization problem in this context is to find

a Steiner Minimal-Cost Tree (SMT) rooted at s which covers $P(s)$, given an edge-weighted graph $G = (V, E, w)$ and a subset $S \subseteq V$ of required vertices (corresponding to $P(s)$). A Steiner tree is a tree t in G that spans all vertices of S. The optimization problem associated with Steiner trees is to find a minimal-cost Steiner tree. If $S = V$, a minimal-cost Steiner tree corresponds to a minimum-weight spanning tree of G.

Figure 5 illustrates a subsumption graph for the source feed s_2 (transitive edges are omitted) and a minimal *Steiner* tree of this graph (in bold). The filtering operators involving s_2 are

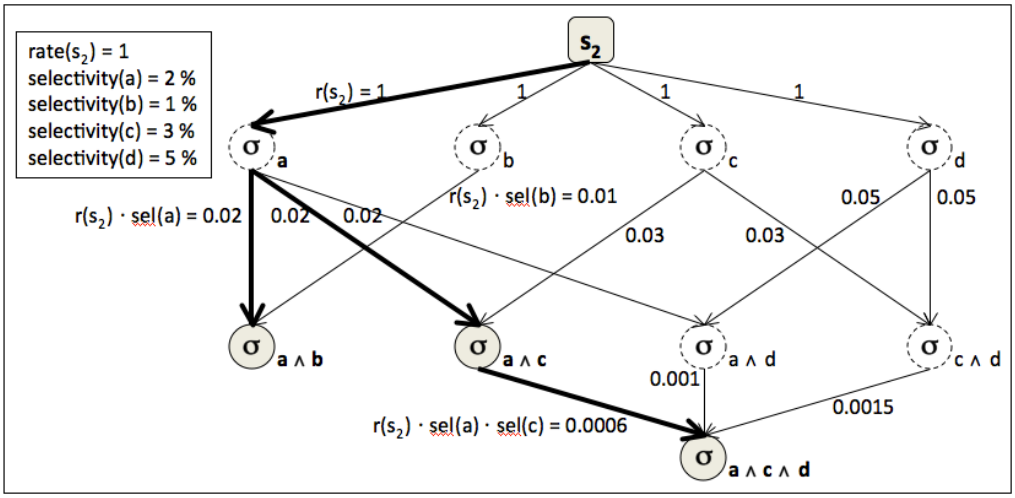

Figure 5: Subsumption Graph for s_2 and Steiner Tree

$\sigma_{a\wedge b}$, $\sigma_{a\wedge c}$, $\sigma_{a\wedge c\wedge d}$ (see Figure 4). The subsumption graph is composed of these three operators, as well as of all their sub-expressions: σ_a, σ_b, σ_c, σ_d, $\sigma_{a\wedge d}$ and $\sigma_{c\wedge d}$, while the edges express subsumption between predicates: $\sigma_a \rightarrow \sigma_{a\wedge b}$, $\sigma_b \rightarrow \sigma_{a\wedge b}$, etc. Note that only "direct" subsumption edges are represented in Figure 5, actually the graph also includes edges from the transitive closure of the subsumption relation, e.g. $\sigma_a \rightarrow \sigma_{a\wedge c\wedge d}$. Edge weight is represented by *the selectivity* of the originating predicate, proportional to the rate of items coming from the source.

We can easily observe that the resulting filtering plan is less costly than the initial one. And this is true despite the fact that normalization might increase the cost of the filtering plan since it replaces cascading selection paths by a conjunction of all predicates on the path. As a matter of fact, it can be demonstrated that the subsumption graph regenerates all these paths and the original plan is always a sub-tree of this graph. Since the Steiner tree is a minimal sub-tree for evaluating the initial set of predicates, its cost will be at most the cost of the initial graph. For example, the filtering cost for source s_2 in the original plan (Figure 3) roughly is twice the publishing rate of s_2 (both selections $\sigma_{a\wedge b}$ and $\sigma_{a\wedge c}$ are applied to all items generated by s_2). This cost is reduced to its half in the final Steiner tree by introducing the additional filter a.

5 Factorization Algorithms

The VCA predicate factorization algorithm: The Steiner Minimal-Cost Tree (SMT) problem is known to be NP-complete [**?**, **?**] and many approximation algorithms have been proposed in the literature. We have initially implemented in *RoSeS* a modified version of the algorithm introduced in [**?**] for edge-weighted directed graphs. Our variation of this general-purpose SMT algorithm lies on the fact that predicate subsumption graphs in *RoSeS* are acyclic (all equivalent

predicates are merged into a single node). Despite this simplification, our Algorithm 2 (given in Appendix A) still needs to exhaustively search Steiner-trees of minimal cost for various possible subsets of the query predicates and thus do not scale for large predicate graphs (see also experimental evaluation in Section 6).

Algorithm 1 VCA

Input: a set of filtering predicates *Preds* and a selectivity function sel_S which returns the selectivity of any predicate on source S

Output: minimal-cost filter plan $Tree$ for predicates $Preds$ on source s

1: // build an initial filter plan where each predicate is evaluated independently
2: $Tree \leftarrow \{(true, pred) |\, pred \in Preds\}$
3: $Border \leftarrow Preds$
4: // $Cands$: all predicates p subsuming at least two predicates in $Border$
5: $Cands \leftarrow FindCandidates(Border)$
6: **while** ($Cands \neq \{true\}$) **do**
7: // $bestCand$: the most selective candidate
8: $bestCand \leftarrow \arg\min_{cand \in Cands}\, sel_S(cand)$
9: // $Factorized$: all predicates in the Border subsumed by $bestCand$
10: $Factorized \leftarrow \{p |\, p \in Border \wedge p \models bestCand\}$
11: **if** $benefit(bestCand, true) > 0$ **then**
12: // Expand : insert bestCand as child of the root
13: **if** $bestCand \notin Border$ **then**
14: $Tree \leftarrow Tree \cup \{(true, bestCand)\}$
15: **end if**
16: **for all** $fact \in Factorized - \{bestCand\}$ **do**
17: // $fact$ becomes a child of $bestCand$
18: $Tree \leftarrow (Tree - \{(true, fact)\}) \cup \{(bestCand, fact)\}$
19: $k \leftarrow outdegree(fact)$
20: **if** $fact \notin Preds \wedge k \neq 0 \wedge benefit(fact, bestCand) < 0$ **then**
21: // Reduce : replace $fact$ by $bestCand$ as parent of all children of $fact$
22: **for all** $child \in Children(fact)$ **do**
23: $Tree \leftarrow (Tree - \{(fact, child)\}) \cup \{(bestCand, child)\}$
24: **end for**
25: **end if**
26: **end for**
27: $Border \leftarrow Children(true)$ // recompute $Border$
28: $Cands \leftarrow FindCandidates(Border)$ // recompute $Cands$
29: **else**
30: $Cands \leftarrow Cands - \{bestCand\}$ // remove $bestCand$ from $Cands$
31: **end if**
32: **end while**
33: **return** $Tree$

For these reasons we have devised a new approximate algorithm for minimal-cost Steiner Trees, called *VCA* (see Algorithm 1), that takes into account the peculiarities of predicate subsumption graphs and the corresponding cost-model in *RoSeS*. It is a greedy algorithm, iteratively improving an existing filter plan according to a local heuristic which estimates the benefit of adding a new intermediary predicate. VCA starts from an initially correct plan where the

root (predicate $true$) is directly connected to each predicate in the set $Pred$ of target predicates, e.g. $Pred = \{a \wedge b, a \wedge c, a \wedge c \wedge d\}$ in Figure 5. This corresponds to a plan where all filtering predicates are evaluated independently on the source feed and whose cost is proportional with the number of predicates (selectivity of root $true$ $\times$ the number of filter predicates).

We note $Border$ the set of children of the root in the current plan, i.e. $Border = Pred$ at the beginning. The $Border$ essentially contains the set of vertices to be potentially factorized at each iteration step of the VCA algorithm. For each $Border$ we consider a set of candidates for predicate factorization, denoted by $Cands$, containing nodes that subsume at least two predicates in the current border. Note that the intersection between $Border$ and $Cands$ is not necessarily empty, e.g. in Figure 5 $a \wedge c$ belongs at the beginning to both $Border$ and $Cands$ (it subsumes border nodes $a \wedge c, a \wedge c \wedge d$). Based on these two sets, VCA iteratively tries to replace with some node in $Cands$ all the nodes it subsumes in $Border$. The cost of the final tree obviously depends on the choice of these candidates and it is guided by two measures: (1) the selectivity of the candidate predicates and (2) the number of existing predicates they factorize. More formally, we define a *function* $benefit(x, y)$ estimating the benefit of adding x as a child of an existing node y in the tree, where k is the number of children of y which are also subsumed by x. Inserting x means replacing edges from y to the k children with an edge from y to x and edges from x to the k children.

$$benefit(x, y) = (k - 1) \cdot selectivity(y) - k \cdot selectivity(x)$$

This function expresses the difference between the costs of the filter trees after and before the insertion of x. A positive $benefit$ means the filter tree is improved by the insertion of node x, a negative one means the plan may be improved by deleting x. This information is exploited by VCA in two ways. First, during the *expansion* phase, it is used to decide whether a candidate node should be inserted in the current filter tree, and moreover, to select the one with the maximal benefit. However, computing the benefit for all candidates includes computing the number of border predicates they subsume, which increases the computation cost of the algorithm. As verified experimentally in Section 6, we can quickly obtain a very good approximation of the minimal Steiner tree by choosing candidates only based on their selectivity.

During the expansion phase, new nodes are always added on top of the border, as children of the root, i.e. their benefit is computed as $benefit(x, true)$. However, adding new nodes changes the context of existing ones, whose utility may be rediscussed. For instance, consider the current plan already containing node x that had been added because $benefit(x, true) > 0$. A new node z is inserted and becomes the new parent of x. If selectivity of x and z are close, it is possible that $benefit(x, z) < 0$, i.e. the plan improves if x is discarded and z is directly connected to x's children. This is the second use of the benefit function, during what is called the *reduction* phase. It is also noteworthy that the reduction has not to be performed recursively, since the weight of the edges monotonically decreases with the depth of the predicate subsumption graph. Indeed, if children of x have not been reduced when x was added, they will not be reduced by z because $sel(z) > sel(x)$.

VCA always stops after a finite number of iterations: at each iteration step either the best candidate replaces at least one more specific (selective) node in the border or is removed from the border. Hence, the border corresponds to a bottom-up traversal of the subsumption predicate graph and since this graph is acyclic, the algorithm always ends with the trivial border $\{true\}$.

Finding the best candidates with VCB: VCA reduces drastically the computation cost of the factorization process of similar quality than the general-purpose SMT algorithm of [**?**]. Whereas we have no guarantees about the approximation error of VCA (compared to existing approximate SMT algorithms), our experiments show that the cost of the filter plans obtained by the different algorithms are very similar. This is probably due to the biased distribution of the weights in the graph.

One last open issue concerns the construction cost of the candidate set and the identification of the best candidate nodes in this set. A straightforward way to do this is to build the *subsumption graph* in order to find the most selective predicate subsuming at least two nodes in the border. There are two major drawbacks here. First, building the complete subsumption graph is inefficient, since most subsumption links are never accessed by the algorithm. Second, the candidate list is recomputed at each update of the border ($FindCandidates$) without considering previous computations. To reduce this cost we have designed a data structure, called *Very Clever Border* (VCB), which (1) computes on the fly the minimal number of subsumption links at each iteration step for choosing the most selective predicate and (2) updates the list of candidates in an incremental way (the candidates for each node are computed at most once).

The main idea of VCB is to generate in advance for each border node the list of its subsuming candidate predicates. This list is ordered by the selectivity of the corresponding candidates and continuously evolves without re-computing the set of candidates at each iteration step. We will detail its behavior through an example.

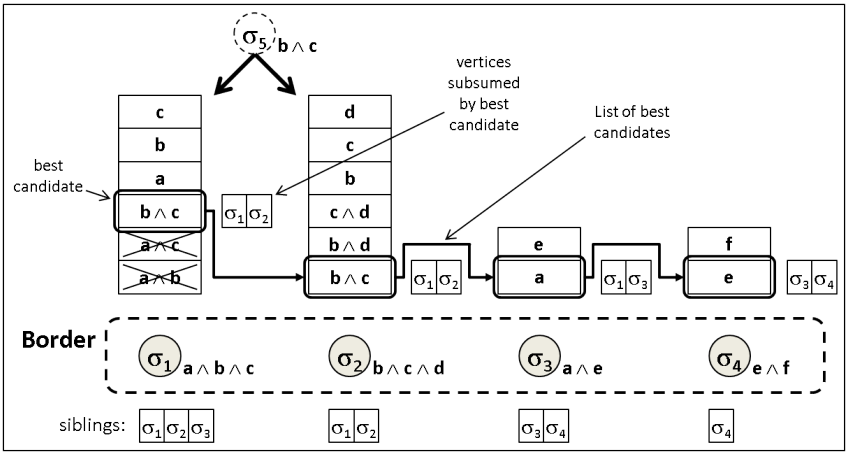

Figure 6: Very Clever Border

Consider a source feed s which is associated with four filter predicates: $a \wedge b \wedge c$, $b \wedge c \wedge d$, $a \wedge e$, $e \wedge f$ (see Figure 6). Initially, we generate for each filter predicate p in the border the list of candidates $candList(p)$. To find the most selective candidate we traverse each list to find the first candidate subsuming at least another predicate in the border. This candidate will be called the best candidate for p, denoted *bestCand(p)*. In our example, the best candidate for node $a \wedge b \wedge c$ is $b \wedge c$ ($a \wedge b$ and $a \wedge c$ can safely be removed from $candList(a \wedge b \wedge c)$. Additionally, we store for each best candidate $bestCand(p)$ the list of nodes in the border which are subsumed by $bestCand(p)$ (σ_1 and σ_2). Symmetrically, we also compute for each predicate p the list of nodes whose best candidate subsumes p. This list is called the *sibling list* of p. For instance, node σ_1 is subsumed by the best candidates of nodes σ_1, σ_2, σ_3. This list is used later to notify all nodes who should update their *best candidate* if p is removed from the border. The

best candidates found for each border predicate are inserted in a *list of best candidates* which is sorted by their selectivity. In our example $b \wedge c$ is more selective than a, which is more selective than e. Finally, the first *best candidate* in this list ($b \wedge c$) is chosen in order to perform the *expand* operation. The node σ_5 is generated, nodes σ_1 and σ_2 are removed from the *border* and σ_5 is added to the *border*. Then the siblings of σ_1 and σ_2 are notified to update their best candidate list (σ_3). Indeed, σ_1 is not anymore in the border, so a (best candidate of σ_3) subsumes only one node, himself. The new *best candidate* of σ_3 will be: e, subsuming σ_3 and σ_4. So, the next step of the iteration restarts with the updated new *border*.

6 Experimental Evaluation

An important issue for the experimental evaluation of our factorization algorithm was the generation of representative collections of filtering queries over RSS feeds. Having no real data set available, we have adapted an existing *RSS subscription generator* to our setting. This generator follows the heuristics that the probability (frequency) of a term to appear in a search query is strongly related to its document frequency (which also corresponds to the selectivity) in the searched document collection. The generated queries follow the following format which corresponds to the most simple kind of aggregation queries applying a filter to a union of sources:

```
create feed PublicationName
from (src1 | src2 | src3 | ... | srcN) as $var
where $var[predicate];
```

In order to generate a representative collection of queries, we have crawled 20 RSS feeds during a month and extracted for each feed a representative collection of about 200 keywords with their document frequencies (the percentage of items which contain this keyword). The keyword distribution is used by the *query generator* for generating the query scripts. More precisely, after having chosen the sources for a query, the generator merges the keyword distributions of these sources which defines the probability of a term to appear in the query. Besides the collection of sources and their statistics, the generator can be parametrized with other parameters controlling the number of generated queries in the script, the number of sources used by the generated queries and the syntactic form of the filtering predicates. All filtering predicates are in *conjunctive normal form* (CNF) composed of a random number $d \in [d^{min}, d^{max}]$ of disjunctions (length of the conjunction) picked from a range, where each disjunction contains a random number $a \in [a^{min}, a^{max}]$ of atoms (length of the disjunctions). This allows finally to generate all kinds of scenarios with general filters in CNF, with simple conjunctive queries ($a^{min} = a^{max} = 1$) and with simple disjunctive queries ($d^{min} = d^{max} = 1$). This technique also allows us to simulate scenarios where users define queries with predicates in *disjunctive normal forms* (*DNF*). We know that *DNF* to *CNF* transformation leads to an exponential explosion of the predicate size which can be simulated by some extent by defining big *maximum limits* (d^{max}, a^{max}).

In the following we will shortly describe some experimentation results concerning the scalability and effectiveness of our optimization approach. Our experiments are run on a 3.06 GHz computer with 4 physical cores and with 11.8 Gb of memory.

Conjunctive queries: The first experimentation scenario evaluates and compares the performance of our approximate Steiner algorithm (STA), the VCA algorithm with subsumption graph and the VCA algorithm with the VCB data structure. Our particular interest concerns

the scaling behavior in the number of queries. We know that the number of sources involved in the optimization process is independent of the quality of the solutions (due to the *normalization* process, each source is factorized independently). So we have generated different sets of queries over a single source. We limited the filtering predicates to simple conjunctions of one to three atomic filters (a realistic scenario for search engines). Remind that these predicates are generated according to the distribution of 200 keywords in the filtered feed. The obtained results are shown in Figure 7. The left figure shows the evolution of the computation cost with

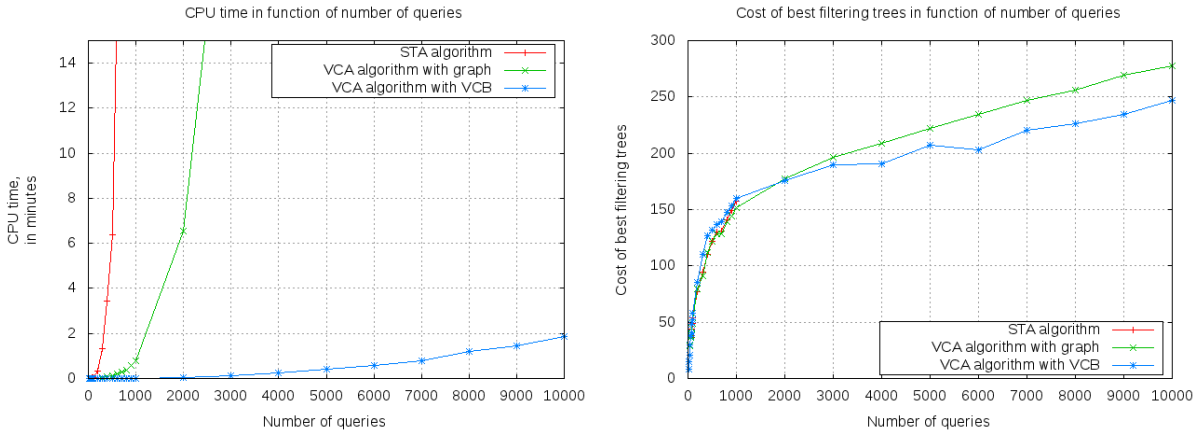

Figure 7: CPU time & Cost of trees for experience I

respect to the number of predicates. We can see that the *STA algorithm* does not even reach the number of 1000 queries in a reasonable time period (15 minutes). The *VCA algorithm with subsumption graph* is limited to 3000 queries, whereas *VCA algorithm + VCB* scales well and is able to optimize 10000 predicates in two minutes. On the right figure, we can see that the filter plans generated by all three algorithms have a similar *evaluation cost*, which is about 2% to 3% of a naive plan. We might expect that both versions of VCA (with graph and with VCB) should generate the same multi-query plans. We explain the difference by the non-determinism of the choice of the best candidate based on its selectivity.

Complex queries: In the second experiment we examine the scalability of the algorithms in function of the length of the filter predicates of the queries. For this purpose we have fixed the *query number per script* to 1000 and generated five sets of queries with an increasing number of disjunctions per predicate (from 1 to 5). The length of the disjunctive predicate is a random number in $[1, 5]$. Our results are shown in Figure 8. We can see that the cost *VCA with VCB* scales with the predicate complexity, while *VCA with subsumption graph* does note scale (*STA* already is out of the displayed limits for the most simple queries). The right figure shows the evaluation cost of the obtained filter plans which are again close. We also can see that an increasing number of conjunctions results in an increasing number of factorizations and the reduction of the obtained *evaluation cost*.

7 Related Work

There is a large amount of previous work on processing data streams and providing new continuous algebras and query languages [**?**, **?**, **?**, **?**, **?**] [**?**, **?**]. Most of these languages are based on a snapshot semantics (the result of a continuous query at some time instant t correspond to the result of a traditional query on a snapshot of its argument) and redesign relational operators

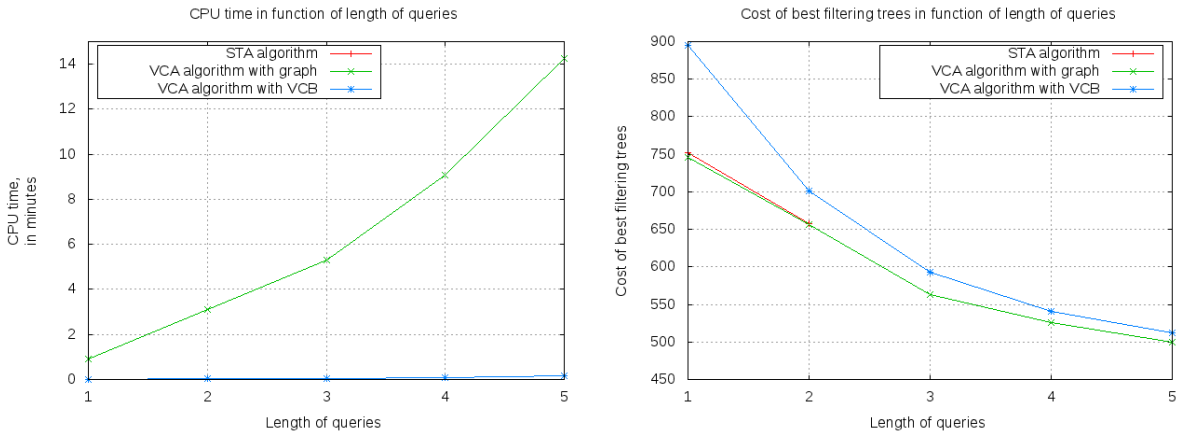

Figure 8: CPU time & Cost of trees for experience II

with a pipe-lining execution model using inter-operator queues. They also introduce different kinds of time-based and count-based window operators (sliding, tumbling, landmark) for computing aggregates and joins [**?**]. The semantic of our data stream model and algebra is strongly influenced by this body of work. The main contribution of our work with respect to existing algebraic frameworks lies on the definition of an annotation join operator which is expressive enough to associate items from different streams while supporting nice rewriting properties.

Multi-query optimization (MQO) has first been studied in the context of DBMS where different users could request complex (one-shot) queries simultaneously to the system which reuses same temporal results [**?**]. Most MQO techniques exploit the fact that multiple queries can be evaluated more efficiently together than independently, because it is often possible to share state and computation like in the system [**?**]. Solutions using this observations are based on several mutualizing methods like predicate indexing by ranked posting lists [**?**], indexing interval of predicate values for join queries [**?**], sharing states in global NFA with *YFilter* [**?**] and *Cayuga* [**?**], join graphs [**?**] or similarity joins hashing [**?**], and finally, sub-query factorization with the *HA* algorithm [**?**], *NiagaraCQ* [**?**], *TelegraphCQ* [**?**], *CQL* [**?**] and *RUMOR* [**?**]. We followed the factorization approach which appeared to be the most promising for a cost-based multi-query optimization solution since factorization decrease CPU and memory usage for operators doing exactly the same work.

MQO through predicate factorization brings difficulties in identifying similar operations with same parameters. It is a "matching problem" within a set of separate query plans where the goal consists in finding a rewriting containing a set of factorizing sub-queries which are shared in the processing of the original queries. This problem is known to be NP-hard [**?**] and has to be simplified to scale up those systems. *NiagaraCQ* [**?**] addresses this problem by factorizing sub-graphs from the execution plan, but it treats RFID and equity flows which is very different from textual requirement of our context. *TelegraphCQ* [**?**] brings an adaptable system for which operators are not connected together by a graph but by tuples that are routed by the stream processor *Eddies* [**?**]. *RUMOR* [**?**] proposes to use multi-operators which combines several feeds in one operator, but contrary to our approach, this solution requires a synchronous treatment of the sub-graph. Our approach addresses an original approach by focusing on maximizing efficiently the Steiner Minimal-Cost Tree (SMT) approximation in a multi-query graph, by maximizing subsumption of predicates. Since web queries provide a large amount of iden-

tical predicates, this solution brings an efficient technique to decrease CPU and memory use with a finest factorization than previous techniques.

Steiner Minimal-Cost Tree (SMT) is an NP-hard problem[**?**, **?**]. Several approximation algorithms have been proposed in the literature for directed and undirected graphs [**?**, **?**, **?**, **?**, **?**, **?**, **?**, **?**] for various application contexts (*e.g.* computer networks, keyword-based database queries). There algorithms comes with different memory requirements and approximation guarantees depending on the way they essentially prune the search space for SMT solutions (*e.g.* Distance Network Heuristics, Dynamic Programming, Local search Heuristics) and the general characteristics of the graph they are applied (*e.g.* directed or undirected, cyclic or acyclic, weighted edges/nodes). VCA, is a local search algorithm for acyclic edge-weighted graphs which exploits the peculiarities of predicate subsumption graphs and in particular the *RoSeS* cost model for expanding or shrinking the Steiner Tree computed in each iteration step. We are currently trying to qualify the quality of the approximation achieved by VCA.

8 Concluding remarks and future work

In this article we have presented *RoSeS*, a large-scale RSS feed aggregation system based on a continuous multi-query processing and optimization. Our main contributions are a simple but expressive aggregation language and algebra for RSS feeds combined with an efficient cost-based multi-query optimization technique. The whole *RoSeS* architecture, feed aggregation language and continuous query algebra have been implemented [**?**]. This prototype also includes a first Steiner tree-based multi-query optimization strategy as described in Section 4.3.

The most important issue we are addressing now concerns two intrinsic dimensions of dynamicity in continuous query processing systems like *RoSeS*. Users who continuously modify the query plan by adding, deleting and updating publication queries introduce the first dimension of dynamicity. The second one is brought by sources, which generally have a time-varying publishing behavior in terms of publishing rate and contents. Both dimensions strongly influence the evaluation cost of a query plan and need a continuous re-optimization strategy in order to compensate performance loss.

A standard approach [**?**, **?**] to this problem consists in periodically replacing a query plan P by a new optimized plan O (query plan migration). In order to control the trade-off between optimization cost and execution cost, the optimization of a plan only occurs when the difference of cost between P and O exceeds a certain threshold. The problem here is to estimate this difference efficiently without rebuilding the complete optimal plan. We are currently studying such a cost-based threshold re-optimization approach adapted to our context. The basic idea is to keep the subsumption graphs generated for each source during optimization. These subsumption graphs can be maintained at run-time by adding and deleting source predicates and updating the statistics concerning each source (selectivity, publishing rate). Using an appropriate threshold measure, it is then possible to recompute the optimal *Steiner* trees and update the corresponding query plan fragments. A particular sub-problem here is to define an incremental approximate *Steiner* tree algorithm for reducing optimization cost.

Acknowledgments: Thanks to Nelly Vouzoukidou for her RSS feed query generator which helped us much for our experiments.

A Algorithms

Algorithm 2 STA

Input: a directed acyclic labeled weighted graph $Graph$, a *set of terminal nodes* $Term \subseteq N$,
Output: a minimal-weight *sub-tree* $Tree$ of $Graph$ rooted at predicate *true* covering all terminals in $Term$

$Tree \leftarrow \varnothing$
$k \leftarrow |Term|$ // size of $Term$
while $(k > 0)$ **do**
  // $Tree$ does not cover all terminals
  $Tree_{best} \leftarrow \varnothing$
  $density(Tree_{best}) \leftarrow \infty$
  for all $vertex \in Nodes(Graph)$ **do**
    for all $k', 1 \leq k' \leq k$ **do**
      $Tree' \leftarrow STA(Graph, vertex, Term, k')$
      $Tree' \leftarrow Tree' \cup \{(root, vertex)\}$ // add root $root$
      $density(Tree') \leftarrow \omega_S(Tree')/|N' \cap Term|$
      if $(density(Tree_{best}) > density(Tree'))$ **then**
        $Tree_{best} \leftarrow Tree'$
      end if
    end for
  end for
  $Tree \leftarrow Tree \cup Tree_{best}$
  $Term \leftarrow Term - Nodes(Tree_{best})$
  $k \leftarrow |Term|$
end while
return $Tree$

$Graph$ is a directed labeled graph where each node $n \in Nodes(Graph)$ is labeled by filtering predicate $\lambda(n)$ and there exists an arc $(a, b) \in E$ between two nodes a and b if and only if $\lambda(a)$ subsumes $\lambda(b)$. We assume that $Graph$ is acyclic (all logically equivalent nodes are represented by a single node in $Graph$) and closed under subsumption: for any node $a \in Graph$ and any predicate P subsuming $\lambda(a)$, either $P \equiv \lambda(a)$ or there exists an ancestor b of a in $Graph$ such that $\lambda(b) \equiv P$ (for the root r of $Graph$: $\lambda(r) \equiv true$). Labeling function $\omega_S : E \rightarrow \Re$ returns for each edge in $(a, b) \in E$ a weight which corresponds to the selectivity of a on source feed S. The weight $\omega_S(t)$ of a tree t is the sum of the weights of its edges. The principle of Algorithm 2 is to recursively iterate over all successors for each vertex in the graph in order to find and optimal sub-tree of $Graph$ with respect to the weight function. The choice of a sub-tree t' in the inner loops not only takes account of the weight of t', but also of the number of terminals (initial predicates) covered by t (*tree density*).