RoSeS : A continuous content-based query engine for RSS feeds

Jordi Creus[1], Bernd Amann[1], Nicolas Travers[2], and Dan Vodislav[3]

[1] LIP6, CNRS – Université Pierre et Marie Curie, Paris, France
[2] Cedric/CNAM – Conservatoire National des Arts et Métiers, Paris, France
[3] 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

In its origins the Web was a collection of semi-structured (HTML) documents connected by hypertext links. This vision has been valid for many years and the main effort for facilitating access to and publishing web information was invested in the development of expressive and scalable search engines for retrieving pages relevant to user queries. More recently, new web content publishing and sharing applications that combine modern software infrastructures (AJAX, web services) and hardware technologies (handheld mobile user devices) appeared on the scene. The web contents published by these applications is generally evolving very rapidly in time and can best be characterized by a stream of information entities. *Google News*, *Facebook* and *Twitter* are among the most popular examples of such applications, but the list of web applications generating many different kinds of information streams is increasing every day.

In our work we are interested in RSS and ATOM as standard formats for publishing information streams. Both formats can be considered as the continuous counterpart of static HTML documents for encoding semi-structured data streams in form of dynamically evolving documents called *feeds*. They both use very similar data models

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”

and follow the design principles of web standards (openness, simplicity, extensibility, genericity) for generating advanced web applications.

Figure 1 illustrates the usage of RSS[4] in the context of social networks. User Bob is registered to various different social media sites like *Facebook*, *Twitter*, *Flickr*, *YouTube* etc. as a publisher and a reader. All of these sites propose different kinds of services for publishing and annotating any kind of web contents (web pages, images, videos). Bob is also an active member of different groups publishing photos on *Flickr*. He rapidly feels the need of a unique interface for observing the different information streams at a glance. This kind of service can be provided by social web sites or other mashup interfaces like *NetVibes* or *iGoogle* in form of widgets for building mashup pages (see the upper left box of figure 1).

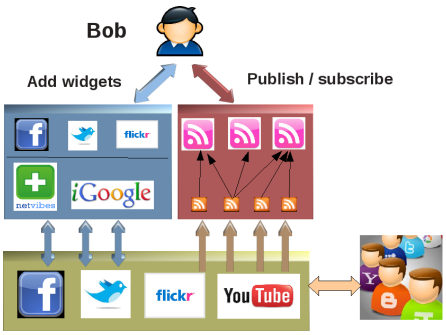

Fig. 1: Social network streams scenario

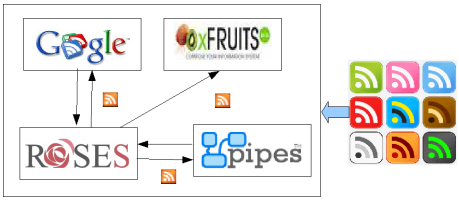

Fig. 2: RSS aggregation infrastructure

A more flexible data-centric solution consists in aggregating RSS streams. As shown in figure 1, Bob can use an RSS aggregator to subscribe to different kinds of RSS services proposed by the various social networks and media sites. This aggregator provides a uniform interface for creating new personalized information streams composing and filtering items published by collections of source feeds. A more detailed description of how RSS aggregation can be used for combining social information streams is described in Section 3.1.

RSS is also widely used by professional information publishers like journals and magazines for broadcasting news on the web. Combined with an adapted publish-subscribe middle-ware, this offers an efficient way for the personalized delivery of news. RSS aggregators like *GoogleNews* enable the personalization by defining and publishing RSS views over collections of news feeds. Each such view is defined by a declarative query which merges and filters news of a possibly important number of source feeds (for example all major French journals). Following the publish-subscribe principle, these views can be reused for building other streams and the final result is an acyclic graph of union/filtering queries on all the sources. One of the issues tackled in this article concerns the optimization of such plans.

[4] In the following we will use the term *RSS* for both formats, RSS and ATOM.

In this article we present RoSeS (Really Open Simple and Efficient Syndication), a generic framework for content-based RSS feed querying and aggregation. Our framework is based on a data-centric approach, based on the 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.

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, it can be used as a continuous RSS stream middle-ware for dynamic information sources. This is illustrated in Figure 2. 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,
- 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 the second important contribution about query processing and multi-query optimization. Related work is presented in Section 5 and Section 6 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 3, 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 (diffusion) 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.

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 [HAA10], 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.

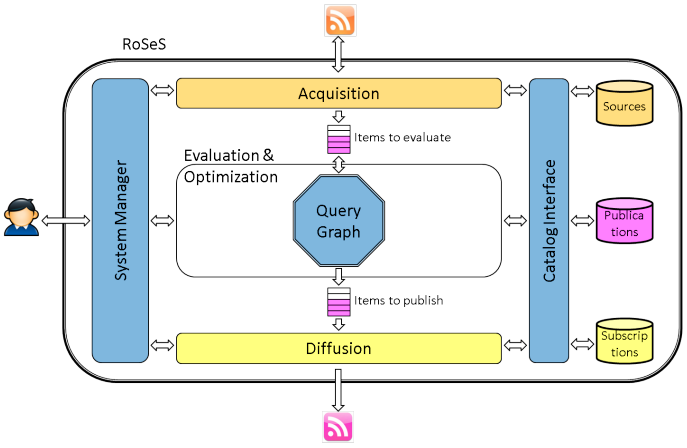

Fig. 3: RoSeS System architecture

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.

Diffusion: 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 with annotations, in given formats (SMS, email, RSS/Atom feed. . .).

3 RoSeS Data Model and Language

3.1 The RoSeS language

The RoSeS language provides three main functionalities: *registering new source feeds* (acquisition level), *defining new publications* (evaluation level) and *creating subscriptions* (diffusion level). We focus here on the description of the main component, the publication language, completed with few details about registering and subscriptions.

The *publication language* has been designed to respond to several desiderata: to be expressive but simple to use, to facilitate most common operations and to be appropriate for large scale multi-query optimization. In RSS syndication, the most commonly used aggregation is based on large unions combined with filtering. RoSeS enforces the simplicity in use of its declarative publication language, by favoring the expression of large unions, combined with targeted filtering and joins.

A publication query contains three clauses:

– A mandatory **from** clause, which specifies the input feeds that produce output items, called *main feeds*.

– Zero, one or several **join** clauses, each one specifying a join with a *secondary feed*. Secondary feeds only produce annotations (no output) to main feed elements.
– An optional **where** clause for filtering conditions on main or secondary feeds.

For instance, suppose Bob organizes with his friends an outing to a rock concert. He defines a publication *RockConcertStream*, including items about concerts from his friends messages (feeds *FriendsFacebookStream* and *FollowedTwitterStream*), and rock concert announces from feed *EventAnnounces*. Notice that the **from** clause allows defining groups (unions) of feeds, identified by variables and used then to express conditions on the group in the **where** clause. Here filtering conditions are expressed on *EventAnnounces* (title contains 'rock') and on grouped feeds (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 group, creates feed *MusePhotoStream* with items about Muse, annotated with photos. Items come from feeds *RockConcertStream* (those talking about Muse) and *MuseNews*, while photos come from two secondary feeds: *FriendsPhotos* with photos published by his friends and *MusicPhotos* (only for category 'rock'). Annotation is realized by join, each item from the main feed ($main) is annotated with photos in the last 3 months from secondary feed items having similar titles. Notice that a join specifies a window on a group of secondary feeds, a main 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 registering language allows defining new source feeds coming either from external RSS/Atom feeds, or from internal materialized publications. Note that RoSeS does not allow item transformation in publications (virtual feeds). However, transformations are possible if the resulting feed is materialized and registered as a new source feed at the acquisition level. Transformations are expressed through XSLT stylesheets that transform a RoSeS item structure into an other. Transformations may use annotations produced by joins, e.g. include corresponding links to photos of feed *MusePhotoStream* at materialization time.

The examples below illustrate the registering of an external RSS feed and of a materialized publication.

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

The subscription language allows defining subscriptions to existing publication / source feeds. A subscription specifies a feed, a notification mode (RSS, mail, etc.), a periodicity and possibly a transformation. Subscription transformations are expressed by XSLT stylesheets, but unlike registering transformations, the output format is free.

The following example shows two subscriptions to the *RockConcertStream* publication: the first one extracts item titles ('Title.xsl' transformation) and sends them by mail every 3 hours, the second one simply outputs an RSS feed refreshed every 10 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 a registered external RSS/Atom (source) feed, or to a publication (virtual) feed. A feed is a couple $f = (d, s)$, where d is the feed description and s is a RoSeS stream. Description is a tuple, representing usual RSS/Atom feed properties: title, description, URL, etc.

A RoSeS stream is a data stream of annotated RoSeS items. More precisely, a RoSeS stream is a (possibly infinite) set of 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 links joined items to an element. An annotation is a set of couples (j, A), where j is a join identifier and A is a set of items - the annotation is further detailed in the join operator below.

RoSeS items represent data content conveyed by RSS/Atom items. Despite the adoption of an XML syntax, RSS and Atom express rather flat text-oriented content structure. Extensions and deeper XML structures are very rarely used, therefore we made the choice of a flat structure, as a set of typed attribute-value couples, including common RSS/Atom properties: 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 window expresses subsets of a stream's items valid at various moments. More precisely, a window w on a stream s is a 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. Window are used in RoSeS only for computing joins between streams. RoSeS uses sliding windows of two types: time-based (last n units of time) and count-based (last n items).

Publication definition is based on *five main operators* for composing RoSeS streams. We distinguish *conservative* operators (filtering, union, windowing, join), that do not change the content of the input items, i.e. they output only already existing items, from *altering* operators (transformation), that produce new items.

Subscribed publications 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).

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

The algebra is defined in the rest of this section and the evaluation of algebraic expressions is detailed in section 4.

A central design choice for the RoSeS language is to express *only conservative operators in the publication language*. The advantage is that conservative operators have good query rewriting properties (commutativity, distributivity, etc.), which 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). Expressiveness is preserved first by allowing join, the most powerful operator, filtering, union and windowing in the publication language. Next, transformation can be used in defining new materialized source feeds. Notice that transformations may use join annotations, and consequently improve (indirectly) the expressive power of joins.

Filtering outputs only the stream elements that satisfy a given item predicate, i.e. $\sigma_P(s) = \{(t, i, a) \in s | P(i)\}$. *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), *shareslink* (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.

Union outputs all the elements in the input streams, i.e. $\bigcup(s_1, ..., s_n) = s_1 \cup ... \cup s_n$. Union may be expressed explicitly, by enumerating input streams, or implicitly, through a query over the existing feeds.

Windowing produces a window on the input stream conforming to the window specification, i.e. $\omega_{t,spec}(s)$ and $\omega_{c,spec}(s)$ define a time-based, respectively a count-based sliding window, where *spec* expresses a duration, respectively a number of items.

Join takes a (main) 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 (main stream filtering based on the window contents), but keeps a trace of the joining items by adding an annotation entry. A join $\bowtie_P (s, w)$ of identifier 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, i.e. $\tau_T(s) = \{T(t, i, a) | (t, i, a) \in s\}$. It 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 presented by a *multi-query plan* composed of different physical operators reflecting the algebraic operators presented in Section 3.2 (union, selection, join and window). Query execution is based on a pipe-lined execution model 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 $G(Q)$ as shown in Figure 4. Graph in Figure 4 represents one possible physical query plan for the following set of 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, window buffers, which are consumed by join operators. View composition is illustrated by an arc connecting a publication operator to an algebraic operator (p_1 is connected to union $\cup_3$). Observe also that transform operators are applied after publication operators.

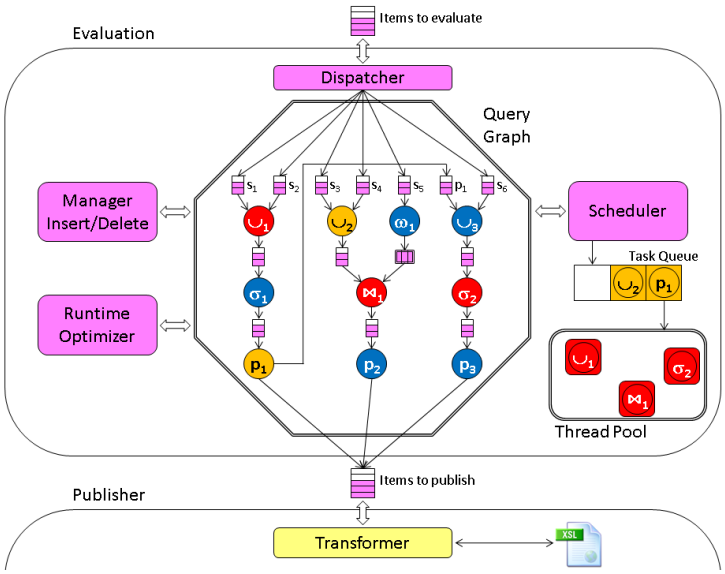

Fig. 4: Evaluation module architecture

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

4.2 Query evaluation and cost-model

As explained in the previous section, a set of publication queries is translated into a multi-query plan 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. We have adopted a multi-threaded pipe-lining execution model which is a standard approach in continuous query processing architectures. Query execution is done as follows. The query graph is observed by a *scheduler* that continuously decides which operators (tasks) must be executed (see Figure 4). The scheduler has at his disposal a pool of threads for executing in parallel a fixed number of threads (the naive solution of attaching one thread to each operator rapidly becomes inefficient / impossible due to thread management overhead or system limitations). 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).

Based on this execution model, we define a cost model for estimating the resources (memory, processor) necessary for the execution of a query plan. Compared to the cost estimation of a traditional query plan which is based on the size of the input data, the estimation parameters of a continuous plan must reflect the streaming nature of the data. We adapt a simplified version of the model presented in [CKSV08] 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).

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: Cost model

As we can see in table 1, the cost of each operator strongly depends on the publishing rate of its input buffer(s). The selection operator assumes a constant execution cost for each item (a more precise model could take account of the size of each item, but since items are in general small text fragments we ignore this detail in our model). Memory cost of selection is also constant (the size of the operator plus the size of one item). 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 output buffer with a publishing rate corresponding to the sum of its input buffer rates. We assume zero memory and processing cost since each union can be 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 s 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 size of the window (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 input buffer rate of each operator strongly influences the global cost of the execution plan (the sum of the cost of all operators) and we will describe in the following section how we can reduce this rate by pushing selections and joins towards the source feeds of the query plan.

4.3 Query graph optimization

The main originality of our framework with respect to other multi-query optimization solutions lies in the explicit integration of a cost model. This makes it more expressive than other approaches without cost model. As mentioned before, our optimization strategy is based on the heuristic that selections and joins should be applied as early as possible in order to reduce the global cost of the plan. We use traditional rewriting rule for algebraic expressions (distributivity of selection over union, commutativity of selection with join and transforming selections into a cascade of selections). Observe that commutativity with join is possible because of the particular nature of our annotation joins which do not modify the input items and guarantee that all subsequent selections only apply to these items. We will describe the whole process in the following.

The optimization process can be decomposed into two phases: (a) a *normalization* phase which applies all rewriting rules for pushing selections towards their source feeds and for distributing joins over union and (b) a *factorization* phase of the selection predicates of each source based on a new cost-based factorization technique.

Query normalization: The goal of the first phase is to obtain a normalized query plan where all filtering selections are applied first to each source before applying joins and unions. This is possible by iteratively applying distributivity of selections on unions and commutativity between selection and join. It is easy to show that we can obtain an equivalent plan which is a four level tree where the leaves of the tree are the sources involved in the query (first level), the second level nodes are the selection operators which can be applied to each source (leaf), the third level are window/join operators operators applied to the results of the selections and the final (fourth) level are unions applied to the results of the selections/windowed-joins to build the final results. Normalization also flattens all cascading selection paths into a single conjunctive selection. This modification might increase the cost of the resulting normalized graph with respect to the original graph. However, as we will show at the end of this section this increase is only temporary and compensated by the following factorization phase.

We will explain our approach by a simple example without join operations. Suppose the three publication queries shown in Figure 5, 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 to push all the selection operations through the publication tree until the sources. This lets us obtain the normal form shown in Figure 6: $p_3 = \sigma_{a \wedge c \wedge d}(s_2) \cup \sigma_{a \wedge c \wedge d}(s_3) \cup \sigma_{a \wedge c \wedge d}(s_4) \cup \sigma_d(s_5)$

Query factorization: Normalization generates a global query plan where each source s is connected to a set of selection predicates $P(s)$. Factorization consists in building a minimal *filtering plan* for each source. To find to best operator factorization

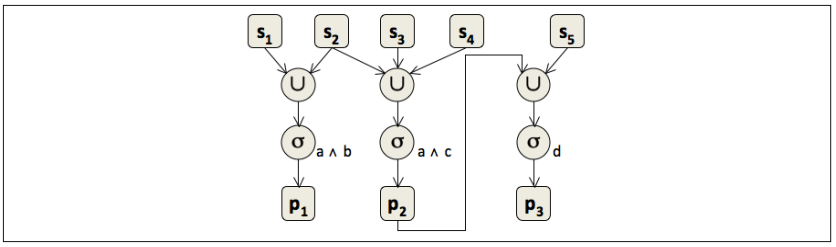

Fig. 5: A query plan for publications p_1, p_2 and p_3

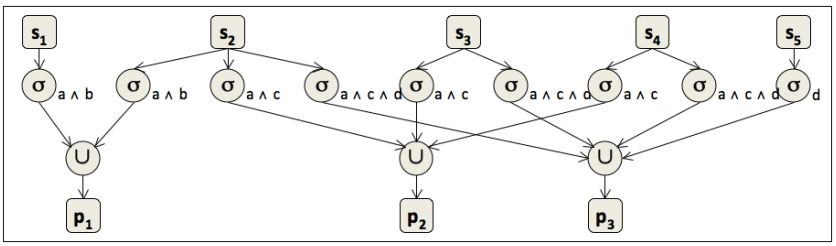

Fig. 6: Normalized query graph

we proceed in two steps: we first generate for each source a *predicate subsumption graph* which contains *all* predicates subsuming the set of predicates in $P(s)$. Each subsumption link in this graph is labeled by a weight corresponding to the output rate of the source node (source or filtering operation). It is easy to show that any sub-tree of this graph covering the source s (root) and all predicates in $P(s)$ corresponds to a filtering plan which is equivalent to the original plan where the cost is obtained by the sum of all arc weights in the tree. Based on this observation we then try to find a Steiner tree [CCC+98], which corresponds to a sub-tree of minimal cost covering all initial predicates. The idea of this procedure is illustrated in Figure 7 showing a subsumption graph for the source s_2 and a minimal *Steiner* tree of this graph (in bold). The selection operators corresponding to s_2 are (fig. 6) $\sigma_{a\wedge b}$, $\sigma_{a\wedge c}$, $\sigma_{a\wedge c\wedge d}$. The subsumption graph is composed of these three operators, the operators corresponding to the sub-expressions of the three initial operators: σ_a, σ_b, σ_c, σ_d, $\sigma_{a\wedge d}$ and $\sigma_{c\wedge d}$, and the subsumption arcs between all these operators ($\sigma_a \rightarrow \sigma_{a\wedge b}$, $\sigma_b \rightarrow \sigma_{a\wedge b}$, ...). Subsumption graph arcs are populated with weights, which represent the estimated evaluation cost of using that arc. They correspond to the product between the involved source's publishing rate and the estimated selectivity of the operator predicate at the tail of the arc (*e.g.*, $cost(\sigma_a \rightarrow \sigma_{a\wedge b}) = rate(s_2) \cdot selectivity(a)$). Then a *Steiner* tree algorithm can be run over the subsumption graph to find the best factorization tree. In our example with selection operators for source s_2, the resulting tree can be seen in Figure 7.

The *Steiner* tree problem [CCC+98] is known to be NP-complete and we have developed an approximate algorithm which exploits the particular properties of a filtering plan (all outgoing arcs of a node have the same weight and the weight is monotonically decreasing with the depth of the node in the graph). This algorithm is based on a

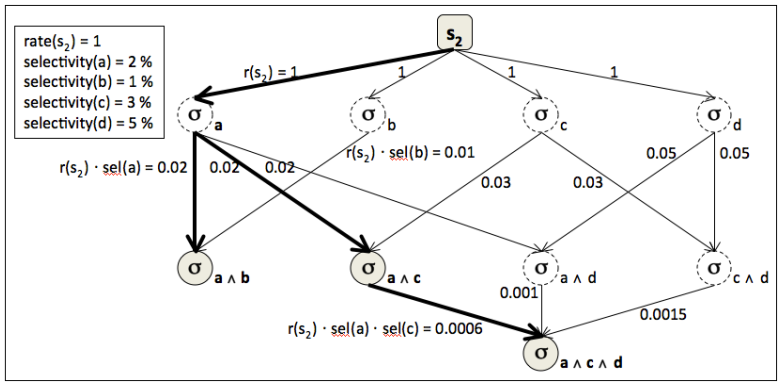

Fig. 7: Subsumption Graph for s_2 and Steiner Tree

local heuristic for dynamically building the subsumption graph by choosing the most selective subsuming predicates. Whereas it will find only an approximate solution, it can incrementally add new predicates generated by new publication queries. A detailed description of the whole process is outside the scope of this paper.

It is easy to show that the final filtering plan has less cost than the initial filtering plan. Normalization can increase the cost of the filtering plan by replacing cascading selection paths by a conjunction of all predicates on the path. However, it is easy to show that the subsumption graph regenerates all these paths and the original plan is 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 be the cost of the initial graph. For example, the cost for source s_2 in the original plan (Figure 5) roughly is twice the publishing rate of s_2 (both selectios $a \wedge b$ and $a \wedge c$ are applied to all items generted by s_2). This cost is reduced to its half in the final Steiner tree by introducing the additional filter a.

5 Related Work

There is a large amount of previous work on processing data streams and providing new continuous algebras and query languages [CDTW00,GÖ03,LMT+05,LTWZ05,ABW06] [WDR06,KLG07]. 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 one-shot query on a snapshot of its argument) and redesign relational operators 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 [GÖ03]. The semantic of our data stream model and algebra is strongly influenced by this work. The main originality of our algebra with respect to existing algebra concerns our definition of annotation join which facilitates the rewriting and optimization of the algebraic query plans.

Since RSS and Atom are encoded with XML, and we also explored existing approaches for querying XML streams. XML streaming systems are concerned with the continuous evaluation of XPath [BCG+03,GS03,PC03] and XQuery [KSSS04,BFF+07]

expressions. The rich semi-structured semantics of XML increases the complexity of the underlying query languages and their implementation. RSS feeds are simple streams of flat XML fragments which do not need highly expressive XML path expressions and we decided to follow a simple attribute/value approach which is more efficient and sufficiently expressive for encoding RSS data.

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 [Sel88]. 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 [DGH+06]. Solutions using this observations are based on predicate indexing [WGMB+09], sharing states in global NFA [DFFT02,DGH+06]), join graphs [HDG+07] and sub-query factorization [SG90,CDTW00,ABW06,CCD+03]. We followed the latter approach which appeared to be the most promising for a cost-based multi-query optimization solution.

Publish/subscribe systems aim to solve the problem of filtering an incoming stream of items (generally generated by a large collection of source streams) according to topic-based or content-based subscriptions. Different techniques propose efficient topic-clustering algorithms [LYD+07] [MZV07], appropriate subscription index structures [KCM09] [FJL+01] [CPY07], and distributed stream processing [RMP+07] [JA06]). These techniques mainly focus on the parallel processing and efficient indexing of conjunctive keyword queries, which are of less expressive power than our aggregation queries. However, they share certain issues and solutions which are conceptually similar to the MQO problem mentioned before.

Finally, we also must relate YahooPipes![Yah] which is very similar to our approach of feed aggregation. YahooPipes! is a web application for building so-called pipes generating *mashups* from several RSS feeds and other external web sources/services. A pipe is a visual representation of the *Yahoo! Query Language* (YQL [YQL][Kol09]) which is an SQL-like language for building tables aggregating information from external web sources (and in particular RSS feeds). To our knowledge, pipes are relational expressions querying a database of storing the items of each feeds. Whereas YQL is more expressive than our algebra, all queries are evaluated independently on demand which excludes MQO techniques as proposed by our solution.

6 Conclusion 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 [CATV10]. 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 in-

troduce 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 [ZSA09,YKPS07] 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.

References

[ABW06] A. Arasu, S. Babu, and J. Widom. The CQL continuous Query Language : Semantic Foundations and Query Execution. In *VLDB*, pages 121–142, 2006.

[BCG+03] C. Barton, P. Charles, D. Goyal, M. Raghavachari, M. Fontoura, and V. Josifovski. Streaming XPath Processing with Forward and Backward Axes. In *ICDE*, pages 455–466, 2003.

[BFF+07] I. Botan, P.M. Fischer, D. Florescu, D. Kossman, T. Kraska, and R. Tamosevicius. Extending XQuery with Window Functions. In *VLDB*, pages 75–86, 2007.

[CATV10] J. Creus, B. Amann, N. Travers, and D. Vodislav. Un agrégateur de flux rss avancé. In *26e Journées Bases de Données Avancées (demonstration)*, October 2010.

[CCC+98] M. Charikar, C. Chekuri, T. Cheung, Z. Dai, A. Goel, S. Guha, and M. Li. Approximation algorithms for directed steiner problems. In *Proceedings of the ninth annual ACM-SIAM symposium on Discrete algorithms*, SODA '98, pages 192–200. Society for Industrial and Applied Mathematics, 1998.

[CCD+03] S. Chandrasekaran, O. Cooper, A. Deshpande, M.J. Franklin, J.M. Hellerstein, W. Hong, S. Krishnamurthy, S. Madden, V. Raman, F. Reiss, and M. A. Shah. TelegraphCQ: Continuous Dataflow Processing for an Uncertain World. In *CIDR*, 2003.

[CDTW00] J. Chen, D. J. DeWitt, F. Tian, and Y. Wang. NiagaraCQ: A Scalable Continuous Query System for Internet Databases. In *SIGMOD Record*, pages 379–390, 2000.

[CKSV08] M. Cammert, J. Krämer, B. Seeger, and S. Vaupel. A cost-based approach to adaptive resource management in data stream systems. In *TKDE*, volume 20, pages 230–245, 2008.

[CPY07] B. Chandramouli, J. M. Phillips, and J. Yang. Value-based notification conditions in large-scale publish/subscribe systems? In *VLDB*, pages 878–889, 2007.

[DFFT02] Y. Diao, P. M. Fischer, M. J. Franklin, and R. To. YFilter: Efficient and Scalable Filtering of XML Documents. In *ICDE*, pages 341–344, 2002.

[DGH+06] A. Demers, J. Gehrke, M. Hong, M. Riedewald, and W. M. White. Towards expressive publish/subscribe systems. In *EDBT*, pages 627–644, 2006.

[FJL+01] F. Fabret, A. Jacobsen, F. Llirbat, J. Pereira, K. Ross, and D. Shasha. Filtering algorithms and implementation for very fast publish/subscribe systems. In *SIGMOD Record*, pages 115–126, 2001.

[GÖ03] L. Golab and M. T. Özsu. Issues in Data Stream Management. *SIGMOD Record*, 32(2):5–14, 2003.

[GS03] A. Kumar Gupta and D. Suciu. Stream Processing of XPath Queries with Predicates. In *SIGMOD Record*, pages 419–430, 2003.

[HAA10] R. Horincar, B. Amann, and T. Artières. Best-effort refresh strategies for content-based rss feed aggregation. In *WISE*, pages 262–270, 2010.

[HDG+07] M. Hong, A. J. Demers, J. Gehrke, C. Koch, M. Riedewald, and W. M. White. Massively Multi-Query Join Processing in Publish/Subscribe Systems. In *SIGMOD Record*, pages 761–772, 2007.

[JA06] S. Jun and M. Ahamad. FeedEx: Collaborative Exchange of News Feeds. In *WWW*, pages 113–122, 2006.

[KCM09] A. C. König, K. Ward Church, and M. Markov. A Data Structure for Sponsored Search. In *ICDE*, pages 90–101, 2009.

[KLG07] M. L. Kersten, E. Liarou, and R. Goncalves. A Query Language for a Data Refinery Cell. In *Int. W. on Event-driven Architecture, Processing and Systems*, 2007.

[Kol09] Nicholas Kolakowski. Yahoo Launches YQL Execute, Updates YSlow. *eWeek.com*, 2009.

[KSSS04] C. Koch, S. Scherzinger, N. Schweikardt, and B. Stegmaier. FluXQuery: An Optimizing XQuery Processor for Streaming XML Data. In *VLDB*, 2004.

[LMT+05] J. Li, D. Maier, K. Tufte, V. Papadimos, and P. A. Tucker. Semantics and Evaluation Techniques for Window Aggregates in Data Streams. In *SIGMOD Record*, pages 311–322, 2005.

[LTWZ05] C. Luo, H. Thakkar, H. Wang, and C. Zaniolo. A Native Extension of SQL for Mining Data Streams. In *SIGMOD Record*, pages 873–875, 2005.

[LYD+07] X. Li, J. Yan, Z. Deng, L. Ji, W. Fan, B. Zhang, and Z. Chen. A Novel Clustering-Based RSS Aggregator. In *WWW*, pages 1309–1310, 2007.

[MZV07] T. Milo, T. Zur, and E. Verbin. Boosting topic-based publish-subscribe systems with dynamic clustering. In *SIGMOD Record*, pages 749–760, 2007.

[PC03] F. Peng and S.S. Chawathe. XPath Queries on Streaming Data. In *SIGMOD Record*, pages 431–442, 2003.

[RMP+07] I. Rose, R. Murty, P. R. Pietzuch, J. Ledlie, M. Roussopoulos, and M. Welsh. Cobra: Content-based filtering and aggregation of blogs and rss feeds. In *NSDI*, 2007.

[Sel88] T. K. Sellis. Multiple-query optimization. *ACM Trans. Database Syst.*, 13:23–52, 1988.

[SG90] T. Sellis and S. Ghosh. On the multiple-query optimization problem. *TKDE*, 2:262–266, 1990.

[WDR06] E. Wu, Y. Diao, and S. Rizvi. High-Performance Complex Event Processing over Streams. In *SIGMOD Record*, pages 407–418, 2006.

[WGMB+09] S. E. Whang, H. Garcia-Molina, C. Brower, J. Shanmugasundaram, S. Vassilvitskii, E. Vee, and R. Yerneni. Indexing boolean expressions. *VLDB Endow.*, 2:37–48, 2009.

[Yah] The yahoo! pipes feed aggregator. http://pipes.yahoo.com.

[YKPS07] Y. Yang, J. Krämer, D. Papadias, and B. Seeger. Hybmig: A hybrid approach to dynamic plan migration for continuous queries. *TKDE*, 19(3):398–411, 2007.

[YQL] Yahoo! query language. http://developer.yahoo.com/yql.

[ZSA09] Y. Zhou, A. Salehi, and K. Aberer. Scalable delivery of stream query result. *VLDB Endow.*, 2:49–60, 2009.