RoSeS : A continuous query processor for large-scale RSS filtering and agregation*

Jordi Creus
Univ. Pierre et Marie Curie
Paris, France
jordi.creus@lip6.fr

Bernd Amann
Univ. Pierre et Marie Curie
Paris, France
bernd.amann@lip6.fr

Nicolas Travers
CNAM
Paris, France
nicolas.travers@cnam.fr

Dan Vodislav
ETIS, Univ. Cergy-Pontoise
Cergy, France
dan.vodislav@u-cergy.fr

ABSTRACT

We present *RoSeS*, a running system for large-scale content-based RSS feed filtering and aggregation. The implementation of *RoSeS* is based on 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. These queries are translated into continuous multi-query execution plans which are optimized using a new cost-based multi-query optimization strategy.

Categories and Subject Descriptors

H.3.3 [**Information Systems**]: Information Search and Retrieval—*Online Information Services, Information Filtering*

General Terms

Algorithms, Performance, Experimentation

Keywords

RSS, continuous query processing, multi-query optimization

1. INTRODUCTION

Global news sites such as Yahoo! News or AOL News attract millions of users each month. For example, during January 2011, 46.3 million unique U.S. people visited Yahoo! News using PCs and laptops from home and work locations[1]. Staying informed by consulting online journals and news aggregators has become a daily activity of millions of people and the previous numbers do not take account of the increasing success of mobile phones and tablets for accessing web contents. On the back-end side of this process, traditional commercial online media (journals) are more and more complemented by new content-producing social media (Facebook, Twitter) which generate an abundant number of text streams for promoting recently published contents.

The increasing number of news producers and the overwhelming amount of news articles published online each day calls for advanced filtering and aggregation techniques for the efficient personalized delivery of news. In this demonstration we present a prototype for large-scale content-based RSS feed querying and aggregation. This prototype is part of the Really Open Simple and Efficient Syndication (*RoSeS*) framework (ANR-07-MDCO-011) [5] implementing a range of services for crawling, filtering and aggregating of RSS feeds. A central goal of this framework is to enable large-scale RSS aggregation based on algorithms and data structures which are scalable in terms of the number of feeds, publications and subscription. The chosen approach is to revisit, redefine and reimplement standard online RSS aggregation services by applying and extending current data stream management and continuous query processing solutions. The implementation of *RoSeS* is based on traditional database concepts like declarative languages, views and multi-query optimization [8]. Users create personalized feeds by defining and composing content-based filtering and aggregation queries on collections of RSS feeds. These queries are translated into continuous multi-query execution plans which are optimized using a new cost-based multi-query optimization strategy.

The paper is organized as follows. Section 2 briefly describes the *RoSeS* language and model. An overview of the platform is explained in Section 3. Query processing and the optimization process are presented in Section 4. Finally Section 5 shortly describes the demonstration scenario.

2. ROSES LANGUAGE

The language we have implemented in the *RoSeS* system

*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"

CIKM '11 Glasgow UK

[1]http://blog.nielsen.com/nielsenwire/online_mobile/january-2011-top-u-s-web-brands-and-news-sites/ retrieved on June 15th 2011.

provides instructions for *registering feeds* (register), *defining new feeds* (create) and *creating subscriptions* (subscribe).

The most important component of the language concerns the creation of new feeds by aggregating existing feeds (**create feed**). We will call the used feeds *source feeds* and the newly created feeds *publications*. The underlying query language is expressive and simple to use, designed to facilitate most common operations and large scale multi-query optimization. The language favors the expression of large unions, combined with targeted filtering and joins through three clauses: (1) A mandatory **from** clause, which specifies the input feeds called *main feeds*, (2) zero or more **join** clauses for joining main feeds with other feeds called *annotation feeds* (see example below) where secondary feeds only produce annotations (no output) to main feed elements and (3) an optional **where** clause for defining filtering conditions on main feeds and annotation feeds. Source feeds are either external RSS feeds or publications. The second kind of sources allows to decompose the aggregation process similar to the usage of views in traditional database systems. External feeds can be referenced directly by their URL or by a local name defined by using the **register feed** instruction. Note that *RoSeS* does not allow item transformation but is only allowed through materialized and registered new sources by using XSLT stylesheets that produce new *RoSeS* items. Finally, subscriptions can be defined on registered source feeds or publications by using the instruction **subscribe to**. A subscription specifies for a given feed, a notification mode (RSS, mail, etc.), a periodicity and a optional transformation. Transformations are expressed by arbitrary XSLT stylesheets defined on the XML representation of RSS items. It is important to mention here the difference between a publication and subscription to a publication. As mentioned before, a publications are views which can be replaced by their definitions (queries) for enabling optimization through query rewriting. On the other hand, subscriptions transform publications into external output which impedes query rewriting.

Example:.

Suppose Bob regularly organizes with his friends outings to rock concerts. He therefore defines a publication *RockConcertStream*, including items about concerts from feed *FollowedTwitterStream*, and rock concert announces from feed *EventAnnounces*. For this, he first registers the corresponding source streams in the system and creates a new publication *RockConcertStream*:

```
register feed http://www.infoconcert.com/rss/news.xml as
      EventAnnounces;
register feed http://twitter.com/statuses/user_timeline
      /174451720.rss as FollowedTwitterStream;
create feed RockConcertStream
 from (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. 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'). Notice that a join specifies a window on a group of secondary feeds, a main feed (through a variable) and a join predicate. Moreover, the join annotations are materialized into the *MuseWithPhotos* publication by applying the *IncludePhotos.xsl* stylesheet.

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

The final 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. SYSTEM OVERVIEW

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 completed by two modules (catalog and system manager) provide meta-data management services for storing, adding, updating and deleting source feeds, publication queries and subscriptions :

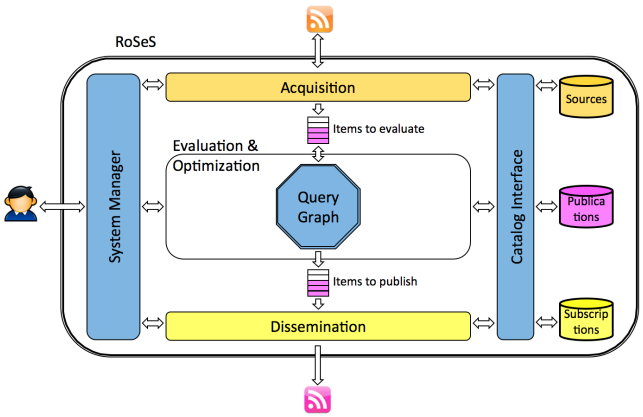

Figure 1: RoSeS System architecture

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 [6].

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...).

4. QUERY PROCESSING

Query processing consists in continuously evaluating a collection of publication queries. This collection is presented by a *multi-query plan* composed of physical operators reflecting the algebraic operators presented in section 2 (union, selection, join and window).

4.1 Query Evaluation and Cost-model

The query graph is observed by a *scheduler* that continuously decides which operators (tasks) must be executed. The scheduler has at his disposal a pool of threads for executing in parallel a fixed number of threads. The choice of an inactive operator to be evaluated is influenced by different factors depending on the input buffer of each operator (number/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 traditional cost estimation based on the input data size, 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 [2] and define the cost of each operator as a function of the publishing rate $R(b)$ of its input buffer b ($S(w)$ is the size of the input window 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	$d * R(b)$	$const$

Table 1: Cost model

As we can see in table 1, the cost of each operator mainly depends on the publishing rate of its input buffer(s). The selection operator assumes a constant execution cost for each item (items are in general small text fragments). The publishing rate of the selection operator reduces the rate of its input by the selectivity factor $sel(p) \in [0, 1]$ depending on the selection predicate p. Union generates an output with a publishing rate corresponding to the sum of its input rates. We assume zero memory and processing cost since each union can be implemented by a set of buffer iterators, one for each input. It is easy to see that the input rate of each operator strongly influences the global cost of the execution plan (sum of all operators cost). 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.

4.2 Query Graph Optimization

An important 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 rules for algebraic expressions (distributivity of selection over union, commutativity of selection with join and transforming selections into a cascade of selections). We describe the whole process in the following.

The optimization process can be decomposed into two phases: (1) a *normalization* phase which applies all rewriting rules for pushing selections towards their source feeds and for distributing joins over union, and (2) 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 these properties always allow us to obtain a normalized 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 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.

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 a best operator factorization, 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). Such a graph is shown in Figure 2. The

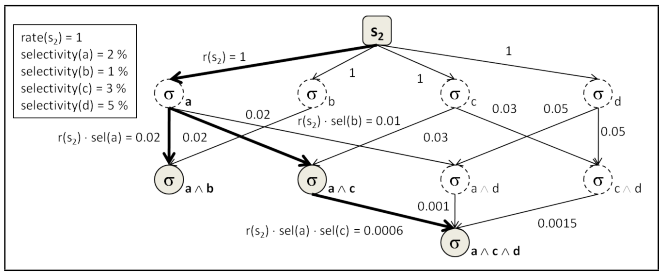

Figure 2: A Predicate Subsumption Graph and Minimal Steiner Tree

filtering predication applied to source s_2 are $a \wedge b$, $a \wedge c$ and $a \wedge c \wedge d$. The subsumption graph is composed of these three predicates and all their subsuming predicates: a, b, c, d, $a \wedge d$ and $c \wedge d$. Each subsumption arc from some predicate p to some predicate q is labeled by the estimated evaluation cost of evaluating q on the result of p. Corresponding to our cost model, this evaluation cost can be estimated by the product between the publishing rate $rate(s_2)$ of source s_2 and the estimated selectivity of predicate p.

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 which is equivalent to the original plan. Such a tree is called a *Steiner tree* [3] where the cost is obtained by the sum of all arc weights in the tree. Based

on this observation our optimization problem then consists in finding a Steiner tree of minimal cost. This is illustrated in Figure 2 showing a subsumption graph for the source s_2 and a minimal Steiner tree of this graph (in bold).

The *Steiner* tree problem [4, 7] 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 local heuristic for dynamically building the subsumption graph by choosing the most selective subsuming predicates. Whereas it will find only an approximate solution, our experiments on real data and synthetically generated queries show that the evaluation cost of the obtained approximate plan generally is near to the evaluation cost of the minimal Steiner tree plan with a much lower optimization cost.

5. DEMONSTRATION SCENARIO

We present a Java prototype integrating all *RoSeS* components described in Section 3. The prototype is based on a multi-threaded architecture and uses *Rome*, the Java's standard library for RSS management. The usage and functioning of the system will be explained using two graphical user interfaces. The first interface (fig. 3) is a Web interface powered by *GWT* (Google Web Toolkit) that allows users to register and share RSS feeds and to visually build publications through an easy-to-use visual programming interface. The second interface is a Java client using IBM's *SWT* (Standard Widget Toolkit) and the *JUNG* (Java Universal Network/Graph) framework for visualizing the multiquery evaluation (fig. 4) as well as predicate subsumption graphs and the resulting optimized filter plans. The prototype and the Java client can be downloaded from [1].

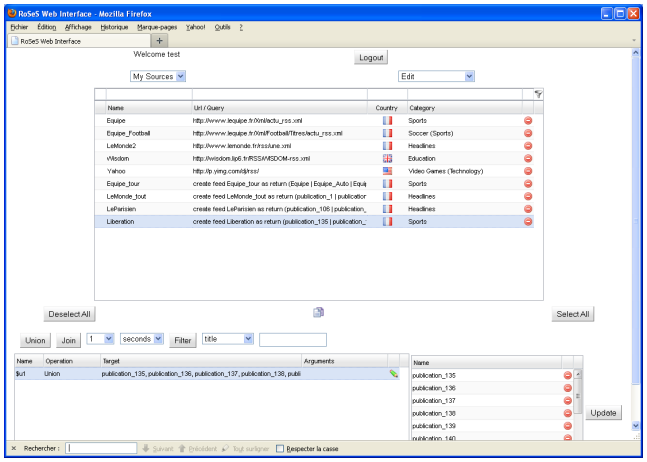

Figure 3: ROSES Web user Interface

Our demonstration is composed of two scenarios. In the first scenario we will present the Web user interface for creating publication queries. We will show how a user can create in some seconds a publication query aggregating hundreds of registered news feeds and personalize this publication by defininning some content-based filtering criteria. Secondly, we will illustrate the usage of join for annotating social network messages (Twitter, Faceobook) with crit-

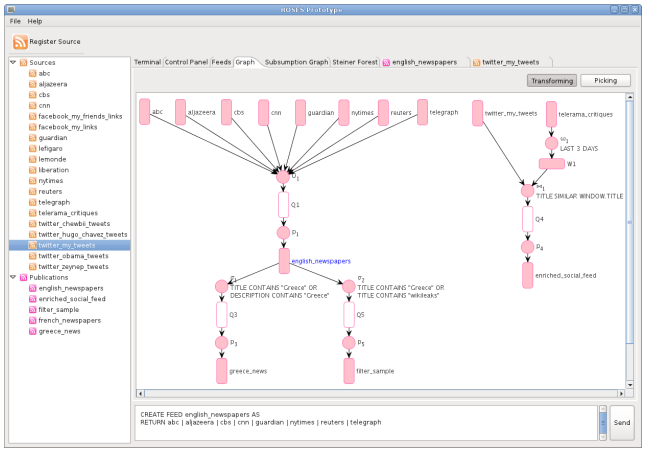

Figure 4: ROSES Prototype screenshot

ics about new movies published in form of RSS feeds (eg. www.filmcritic.com/atom.xml).

The second scenario will illustrate the query processing steps of our protopy using the Java interface. We will load a script of publication containing a workload of 1000 conjunctive filtering queries of the form $q = \sigma_{a_1 \wedge a_2 \wedge \ldots a_k}(src_1 \cup \cdots \cup src_n)$, where $k \in [1, 3]$) and $n \in [1, 10]$ defines the number of sources randomly picked from a fixed set of 25 feeds. Using the Java client, we will visualizethe multi-query plan generated by the system, as well as the predicate subsumption graphs and the optimized filtering trees. Finally, we will also compare query plan optimization and query execution costs for higher workload up to 10000 randonly generated publication queries.

6. REFERENCES

[1] Roses prototypes. http://www-bd.lip6.fr/roses/doku.php?id=prototypes.

[2] 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.

[3] 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.

[4] X. Cheng and D. Du. *Steiner Trees in Industry*, volume 11 of *Combinatorial Optimization*, pages 235–279. Kluwer Academic Publishers, 2002.

[5] J. Creus, B. Amann, N. Travers, and D. Vodislav. RoSeS : A continuous content-based query engine for RSS feeds. In *DEXA*, Toulouse, France, August 2011.

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

[7] F. Hwang, D. Richards, and P. Winter. The Steiner Tree Problem. *Annals of Discrete Mathematics*, (53), 1992.

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