UPMC
SORBONNE UNIVERSITÉS

THÈSE

Pour l'obtention du grade de

Docteur de l'Université Pierre & Marie CURIE

Faculté d'Ingénierie - UFR 919

Diplôme National - Arrêté du 7 août 2006

École doctorale Informatique, Télécommunications et Électronique (Paris)

DOMAINE DE RECHERCHE : BASES DE DONNÉES

Présentée par

Jordi CREUS TOMÀS

ROSES : Un moteur de requêtes continues pour l'agrégation de flux RSS à large échelle

ROSES: A Continuous Query Processor for Large-Scale Content-Based RSS Feed Aggregation

Directeurs de thèse : **M. Bernd AMANN**

et **M. Dan VODISLAV**

Soutenue le 7 décembre 2012
Devant la Commission d'Examen

JURY

Mme Ioana MANOLESCU	Directeur de Recherche au Inria	Rapporteur
M. Jean-Marc PETIT	Professeur des Universités à l'INSA Lyon	Rapporteur
Mme Anne DOUCET	Professeur des Universités à l'UPMC	Examinateur
Mme Béatrice FINANCE	Maître de Conférences à l'UVSQ (HDR)	Examinateur
M. Bernd AMANN	Professeur des Universités à l'UPMC	Directeur de thèse
M. Dan VODISLAV	Professeur des Universités à l'UCP	Directeur de thèse

Abstract

RSS and Atom are generally less known than the HTML web format, but they are omnipresent in many modern web applications for publishing highly dynamic web contents. Nowadays, news sites publish thousands of RSS/Atom feeds, often organized into general topics like politics, economy, sports, culture, etc. Weblog and microblogging systems like Twitter use the RSS publication format, and even more general social media like Facebook produce an RSS feed for every user and trending topic. This vast number of continuous data-sources can be accessed by using general-purpose feed aggregator applications like Google Reader, desktop clients like Firefox or Thunderbird and by RSS mash-up applications like Yahoo! pipes, Netvibes or Google News. Today, RSS and Atom feeds represent a huge stream of structured text data which potential is still not fully exploited. In this thesis, we first present ROSES –Really Open Simple and Efficient Syndication–, a data model and continuous query language for RSS/Atom feeds. ROSES allows users to create new personalized feeds from existing real-world feeds through a simple, yet complete, declarative query language and algebra. The ROSES algebra has been implemented in a complete scalable prototype system capable of handling and processing ROSES feed aggregation queries. The query engine has been designed in order to scale in terms of the number of queries. In particular, it implements a new cost-based multi-query optimization approach based on query normalization and shared filter factorization. We propose two different factorization algorithms: (i) STA, an adaption of an existing approximate algorithm for finding minimal directed Steiner trees [CCC+98], and (ii) VCA, a greedy approximation algorithm based on efficient heuristics outperforming the previous one with respect to optimization cost. Our optimization approach has been validated by extensive experimental evaluation on real world data collections.

Keywords: RSS, Atom, Data Stream Management Systems, publish/subscribe, continuous query processing, multi-query optimization, shared filter factorization, Steiner tree problem

Résumé

Les formats RSS et Atom sont moins connus du grand public que le format HTML pour la publication d'informations sur le Web. Néanmoins les flux RSS sont présents sur tous les sites qui veulent publier des flux d'informations évolutives et dynamiques. Ainsi, les sites d'actualités publient des milliers de fils RSS/Atom, souvent organisés dans différentes thématiques (politique, économie, sports, société...). Chaque blog possède son propre flux RSS, et des sites de micro-blogage comme Twitter ou de réseaux sociaux comme Facebook publient les messages d'utilisateurs sous forme de flux RSS. Ces immenses quantités de sources de données continues sont accessibles à travers des agrégateurs de flux comme Google Reader, des lecteurs de messages comme Firefox, Thunderbird, mais également à travers des applications *mash-up* comme Yahoo! pipes, Netvibes ou Google News.

Dans cette thèse, nous présentons ROSES –Really Open Simple and Efficient Syndication–, un modèle de données et un langage de requêtes continues pour des flux RSS/Atom. ROSES permet aux utilisateurs de créer des nouveaux flux personnalisés à partir des flux existants sur le web à travers un simple langage de requêtes déclaratif. ROSES est aussi un système capable de gérer et traiter des milliers de requêtes d'agrégation ROSES en parallèle et un défi principal traité dans cette thèse est le passage à l'échelle par rapport au nombre de requêtes. En particulier, on propose une nouvelle approche d'optimisation multi-requête fondée sur la factorisation des filtres similaires. Nous proposons deux algorithmes de factorisation: (i) STA, une adaptation d'un algorithme d'approximation pour calculer des arbres de Steiner minimaux [CCC+98], et (ii) VCA, un algorithme glouton qui améliore le coût CPU d'optimisation du précédant. Nous avons validé notre approche d'optimisation avec un important nombre de tests sur des données réelles.

Mots clés : RSS, Atom, Système de Gestion de Flux de Données, PubSub, traitement de requêtes continues, optimisation multi-requête, factorisation de filtres partagés, arbre de Steiner

Remerciement

Tout d'abord je tiens à remercier infiniment mes deux directeurs de thèse, Bernd AMANN et Dan VODISLAV. Bernd est un directeur de thèse exceptionnel, je pense qu'il est le directeur de thèse dont tous les doctorants, quoi qu'elle soit la discipline, rêvent. Il est très fort, il est toujours disponible, pour quoi que ce soit, jour, nuit, et même les week-ends. J'ai appris énormément avec lui, pas uniquement dans le domaine de l'informatique, mais aussi sur la recherche en général et la vie. Il a été, et est, un modèle pour moi, aussi bien comme chercheur et comme personne. Je le considère un peu comme mon petit père autrichien. Dan a été également une source inexorable d'apprentissage scientifique. Si bien moins présent, à cause de la distance géographique, il a été toujours là quand il fallait. J'apprécie énormément son effort à se déplacer toujours chez nous –surtout après mon année d'ATER à Cergy–. Ses remarques et sa capacité d'analyse ont toujours été très pertinentes et et m'ont permis d'avancer dans la bonne direction. Je dois dire que j'étais un privilégié pour pouvoir réaliser ma thèse avec ces deux monstres de l'informatique.

Je ne peux pas oublier trois personnes qui ont aussi contribué énormément à la réussite de ce projet: Michel SCHOLL, Nicolas TRAVERS et Vassilis CHRISTOPHIDES. Malheureusement, Michel n'est plus parmi nous mais il était un des pionniers dans les bases de données en France et je garde un très bon souvenir de lui et des réunions que nous faisions au début de ma thèse, c'était vraiment un honneur pour moi. Vassilis est aussi un grand dans ce domaine, et il m'a beaucoup aidé à avancer et dans mon apprentissage. Pareil pour Nicolas, qui en plus à beaucoup participé dans le développement du logiciel ROSES. J'avoue que je me sentais toujours gâté du fait d'être entouré par ces cinq magnifiques chercheurs.

Je ne peux pas oublier, non plus, mes petits anges (de charlie), Zeynep, Roxana et Myriam, avec lesquelles on a partagé pas mal de temps et qui ont contribué à rendre une expérience

dure comme c'est une thèse, en un voyage plus agréable et sympathique. Aussi Idrissa, avec ses blagues et bonne humeur et nos petits défis. Et les "nouveaux" thésards, la *sympa* Nelly, toujours souriante, et Andrés, avec lequel c'est toujours un plaisir de discuter. Un remerciement va aussi pour les permanents de notre équipe. A Stéphane, qui est à l'origine de mon réveil politique (et m'a appris à relier des manuscrits de thèse). A Anne, directrice de mon stage de master et la meilleure chef d'équipe, toujours prête à t'accueillir avec un grand sourire et à t'aider, un exemple de femme énergétique et radieuse. A Hubert, qui s'intéressait toujours par mes travaux et essayait de m'aider avec des conseils et solutions. A Camelia, à Cécile... et à toute l'équipe de BD du LIP6 en général, une équipe qui fait preuve d'une qualité humaine difficile de trouver ailleurs.

Je remercie aussi mes rapporteurs de thèse: Mme Ioana MANOLESCU et M. Jean-Marc PETIT, qui ont gentiment accepté de rapporter sur mon manuscrit de thèse, ainsi que tous les membres de mon jury: Mme Béatrice FINANCE, Mme Ioana MANOLESCU, Mme Anne DOUCET, M. Jean-Marc PETIT, M. Dan VODISLAV et M. Bernd AMANN.

Evidemment, je dois remercier très spécialement tous les amis que j'ai rencontré à Paris, qui ont fait de mon *aventure parisienne*, l'étape la plus belle de ma vie, qui ont marqué ma vie, et qui seront toujours très profond dans mon cœur: Fabrizio, José Luis, Jérôme, Tommaso, Dario, Donat, Philippe, Luis, David, María De, José *perú*, Sandro, Oktay, Cristina DiGi et Christina Papi, Olivier, Manal, Rouba, Bruno, Hugo et Leo, Joaquim, Kunal, Maria, Alejandro, Maya, Alexandra, Sara, Cherinne, Lara, Karyma et tants d'autres (la liste serait interminable). Mais aussi des amis qui sont plus à Paris comme Jordi Pedrola, Dimitris, Pascual, Gonzalo... et qui malgré ça sont toujours présents dans mes pensées. Je vous remercie tous, pour les bons moments passés ensemble, pour les *rigolades*, les voyages, les soirées, parce que vous m'avez apporté plein des choses, vous m'avez enrichi comme personne, j'ai appris avec vous et vous m'avez rendu une meilleure personne, et aussi parce que vous étiez là dans les mauvais moments.

Finalement, je dois remercier ma famille, mon frère, mon père et ma mère et la petite Dana. Merci pour votre soutien, pour votre aide, votre compréhension, pour votre amour parce que je sais que ce n'est pas facile qu'un fils soit loin de chez soi, mais vous avez été toujours très compréhensifs et vous m'avez toujours donné la liberté de pouvoir choisir. Merci.

Contents

List of Figures

List of Tables

List of Algorithms

Introduction

In its origins the World Wide Web was in essence an evolving collection of *semi-structured* (HTML) documents interconnected by hypertext links. Users could access and observe information using a web browser for navigating from page to page or by querying a keyword-based search engine. This vision has been valid for many years and the main effort for facilitating access to and for 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 built on modern software (AJAX, Web Services) and hardware technologies (mobile handheld user devices) appeared on the scene. These *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. The contents published by Web 2.0 applications is generally evolving very rapidly in time and can best be characterized by a stream of information entities. *Google News*, *Facebook*, *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. This proliferation of content generating 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 daily in Web 2.0, this is unprecedented and creates a vital need for efficient continuous information processing methods which allow users to effectively follow personal-interesting information.

The Web Syndication formats (RSS and Atom) have emerged as a popular mean for timely delivery of frequently updated Web content. *RSS* and *Atom* are standard formats for publishing and aggregating 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 and follow the design principles of Web standards for generating advanced Web applications (openness, simplicity, extensibility and genericity). In RSS/Atom Syndication, 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 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, Flickr) also employ RSS for notifying users about the newly available posts of their preferred friends (or followers).

Among the many different kinds of RSS applications, we can distinguish between three main classes according to the producers [1]: (1) professional newspapers and news agencies, (2) personal blogs and discussion forums, and (3) social media platforms. First, RSS is widely used by professional information publishers like journals and magazines for broadcasting news on the Web. The particular nature of news with respect to other kinds of RSS streams, has lead to the development of specialized news aggregators which offer an efficient way for the personalized delivery of news. They automatically organize news into stories by appropriate tagging, clustering and ranking techniques (*e.g.*, Google News, Yahoo! News, AOL News or MSN News). For example, the *Google News* service covers news articles appearing within the past 30 days on various news websites. In total, *Google News* aggregates content from more than 25,000 publishers in different languages (4,500 English language sites) and provides a number of services (archiving, search, sort...) and is based on a *PageRank*-like score function for promoting news stories (collections of articles related to some topic). Personalization consists in creating personalized news categories containing all news satisfying a user defined full-text search predicate.

The second important kind of RSS applications covers personal blogs, discussion forums and user groups. While this kind of RSS sources exhibits usually a moderate publishing rate compared to professional news sources, the myriad of feeds they produce unquestionably goes above the number of newspapers producers. It is estimated that more than 60M blogs (without considering discussion forums and user groups) actually exist.

The last type of RSS stream producers are social media platforms like *Facebook*, *Twitter*, *Flickr* or *YouTube*. All these sites propose services for exchanging messages and sharing web contents (web pages, images, videos) between users of different groups. Modest estimations

1. In the following we will use the term *RSS* for both formats, RSS and Atom.

place the total number of user accounts on various social media around 230M. As in the news scenario, RSS aggregators can be used to subscribe to RSS feeds for observing and aggregating the activity of a particular user or a user group. On the other hand, news websites and blogs are more and more complemented by social media platforms, which generate an abundant number of text streams for promoting recently published contents.

Therefore, any user who wants to build a personalized information space by subscribing to news feeds, blog feeds, etc. rapidly faces the problem of efficiently exploiting an increasing number of feeds and news items. 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. RSS/Atom intermediaries such as *Google Reader*[2], *Yahoo! Pipes*[3] 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 Alerts*[4] and *Yahoo! Alerts*[5] services enable to filter out syndicated content that is not of interest to users and notify them by email.

Both of these two types of systems, RSS 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. The main difference between them is that existing RSS aggregators are essentially search engines for RSS feeds and they build huge indexes over all RSS feeds, whereas RSS alerters *periodically* evaluate content-based queries over these feeds to generate a stream of results. 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). Furthermore, 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. For these two reasons, the huge number of feeds and the copyright issues, the approach based on *continuously* processing the user-defined queries becomes an interesting solution.

2. www.google.com/reader
3. pipes.yahoo.com
4. www.google.com/alerts
5. alerts.yahoo.com

In this thesis we present *ROSES*, acronym for *Really Open Simple and Efficient Syndication*. *ROSES* is a generic framework for large-scale content-based RSS feed *continuous* querying and aggregation. Our framework relies on a data-centric approach, based on the combination of standard database concepts like declarative query languages, view composition and multi-query optimization techniques. *ROSES* enables the personalization by defining and publishing RSS views over collections of news feeds. Each such view is defined by a declarative query capable of merging and filtering RSS information items originating from a potentially large number of source feeds (for example from all major French journals). *ROSES* supports expressive continuous queries (selection, join, union, window) over textual and factual information streams. 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 involved feeds. These query graphs generally grow very rapidly and call for efficient multi-query optimization strategies to reduce their evaluation cost. This is one of the major issues tackled in this thesis.

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. For instance, a user interested in a precise topic may create a new RSS feed through his *Google Reader* account aggregating thousands of different sources (from economics feeds of general-purpose newspapers websites to specialized blogs). He may then define a *ROSES* publication filtering the feed, that he has created through *Google Reader*, in a specific topic (*e.g.*, the Spanish risk premium and/or the capital stocks of a given company) and aggregating it with other user-defined *ROSES* publications of his interest. Finally, he may use the feed produced by his customized *ROSES* publication on *Yahoo! Pipes*, creating a new feed (*pipe*) using any of the services furnished by *Yahoo!* (for instance, translating the item contents of the *ROSES* feed). Figure 1 depicts such a scenario. This example clearly illustrates the interoperability possibilities of web syndication due to the simple nature intrinsic to RSS and Atom data models.

On the other hand, *ROSES* proposes a join operation which opens some interesting perspectives in the context of feed aggregation. A first usage of join is to define flexible filtering and annotation of news feeds. Joining two feeds consists in filtering all items of a primary feed according to some join predicate (text similarity on the title, description or the whole item) with items present in a sliding window on a second feed. There is no restriction of how this second feed is generated and in particular, it can be defined by the user itself to annotate the items in the first feed. For example, a user might define a primary feed p aggregating a collection of feeds produced by his friends on many different social platforms (*twitter*, *facebook*,

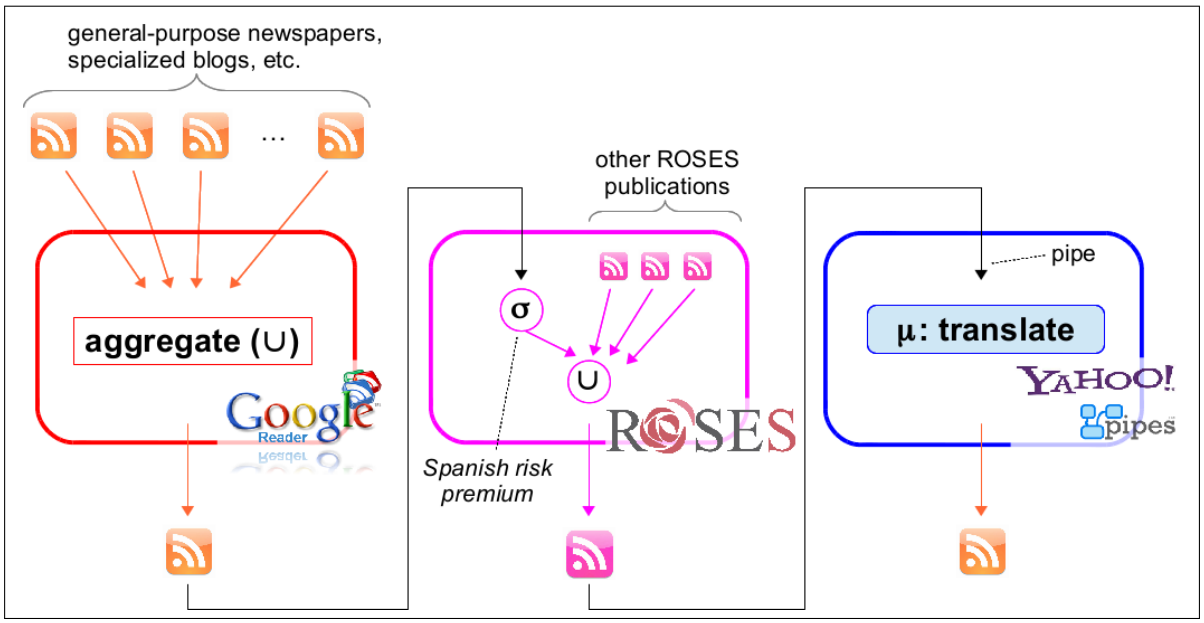


Figure 1 – ROSES as a stream middle-ware

youtube, *flickr*), then he creates a secondary feed s aggregating another collection of feeds generated by different French newspapers and/or weblogs. He defines a sliding window w on the secondary feed s (*e.g.*, a time-based window of a few days) and a join between the primary feed p and this window w (based on the keyword distance between the items from the primary feed and the items of the window, for instance). The results produced by this query will contain the items produced by the main feed p (the social platform items) enriched with the related items previously published on the newspapers/blogs. Another interesting way of using joins, is to join the items of a stream with all items published in the same stream during some time period (*autojoin*). The result is a new stream where all items are regrouped with similar items published before in the same stream according to the text similarity function used by the join predicate.

A central goal of this framework is to enable RSS aggregation at *large-scale*, so we propose algorithms and data structures which are scalable in terms of the number of feeds and aggregation queries (publications). Our approach is to revisit standard online RSS aggregation services by applying traditional database concepts like declarative languages and views and by extending current data stream management and continuous multi-query processing solutions. Multi-query optimization [Sel88] is an important aspect of the *ROSES* System to achieve scalability. Indeed, *ROSES* queries are translated into continuous multi-query execution plans which are optimized using a new cost-based multi-query optimization strategy.

In particular, our work on multi-query optimization focus on content filter factorization on individual sources. The problem can be stated as follows: given a large set of filters which have to be applied on a given source, we want to find the best factorization tree covering all filters and according to a well-defined evaluation cost model. The best factorization tree corresponds to the filtering tree with the lower evaluation cost *w.r.t.* the cost model. We show that the filter factorization problem can be reduced to the (minimum) *Steiner* tree problem [HRW92] applied on a filter *subsumption graph*. The general *Steiner* tree problem is defined as follows: given a weighted graph $G = (V, A, w(\cdot))$ and a set of nodes T appearing in the graph ($T \subseteq V$) and called *terminal* nodes, the *Steiner* tree corresponds to the minimal spanning tree that covers at least all nodes in T, *i.e.* it may also contain other nodes, called Steiner nodes ($S = V - T$). The Steiner tree problem is a known NP-complete problem. Moreover, the *subsumption graph* data-structure grows exponentially *w.r.t.* the number of queries and the number of atomic filtering predicates.

Thus, we propose two algorithms (*STA* and *VCA*) and a data-structure (*VCB*) in order to improve the filtering factorization process. First, we have adapted a general-purpose *approximation* algorithm due to Charikar *et al.* [CCC+98] in order to fit into our *subsumption graphs* characteristics. We have then conceived the *VCA* (Very Clever Algorithm), a *greedy* algorithm that decreases drastically the computation cost of *STA*. Finally, the *VCB* data-structure (Very Clever Border) allows cutting down the space memory required by the *subsumption graphs*. Thus, the VCA algorithm coupled with the VCB data-structure provides a clear amelioration over the STA algorithm.

At a glance, the main contributions we present in this thesis are the following ones:

- A declarative RSS feed aggregation language for publishing large collections of structured queries/views aggregating RSS feeds (Section 2.1),
- An extensible data model and stream algebra for building efficient continuous multi-query execution plans for RSS streams (Section 2.2),
- A join operator called *annotation-join* that opens new interesting possibilities in the context of news aggregation (Sections 2.1.2 and 2.2.2.1),
- A flexible and efficient cost-based multi-query optimization strategy for optimizing large collections of publication queries (Section 3.4),
- A dynamic multi-query optimization technique for reoptimizing query evaluation graphs at runtime (Section 3.5),
- A running prototype based on a multi-threaded asynchronous query plan execution model (Chapter 4).

This manuscript is organized in four chapters. In the first Chapter, the State-of-the-Art, we conduct a survey on different types of systems related to *ROSES*, from existing real world RSS aggregators, to scientific publish/subscribe systems and datastream management systems; then we describe different existing multi-query optimization techniques for static and continuous queries which influenced our work. In Chapter 2, we present the *ROSES* Query Language, the data model and logical algebra, which represent first important contribution of this work. Chapter 3 is the main chapter of this work and is devoted to our query processing engine and our cost-based multi-query optimization approach. In this Chapter, we introduce our filtering factorization technique based on searching minimal-cost Steiner trees in weighted filtering subsumption graphs, we present the *STA* and *VCA* algorithms and the *VCB* data structure for efficiently computing such trees and we evaluate them experimentally. In Chapter 4, we describe the overall *ROSES* System architecture. Then we detail some interesting technical issues encountered during the implementation of the *ROSES* Prototype, and we present the prototype's functionalities. Finally, a last Chapter concludes our work and discusses future work.

Chapter 1

State of the Art

In this thesis we are particularly interested in the problem of evaluating large sets of continuous queries on syndication feeds. *ROSES* queries are complex compositions of continuous operators including selection, join, union and windowing. In this context user-defined queries exhibit *filter sharing*, that is similar selection operators occurring in multiple queries. The aim of the optimization is to minimize the total processing cost by factorizing selection predicate evaluations across multiple queries. Although ROSES queries implement the publish/subscribe principle, our problem is not the same as that addressed by publish/subscribe systems [YGM99], which focus on special techniques for indexing a large number of queries (*subscriptions*) to identify which user subscriptions match incoming data as fast as possible.

This state-of-the-art is organized as follows. First we discuss existing commercial RSS aggregation applications in Section 1.1. Then we make a survey on publish/subscribe systems in Section 1.2. Section 1.3 presents and compares some of the most relevant Datastream Management Systems proposed in the literature (STREAM, Aurora, TelegraphCQ, etc.). Finally, Section 1.4 analyses some related works on *Multi-query Optimization Techniques* (MQOT) close to the one we propose in this thesis.

1.1 Existing RSS aggregation tools

A *Feed Aggregator* enables to register and follow many syndication feeds simultaneously. It may be a local client software or a large-scale web application. We can find a large range of dedicated RSS clients or desktop applications (most current web browsers and messaging clients already incorporate RSS readers). Desktop RSS aggregators periodically crawl the feeds subscribed by the user and store them locally. The advantage of web-based RSS aggregators is that crawling and processing of feed sources is centralized for many different users, which reduces bandwidth usage by efficient crawling strategies [HAA10] and opens many opportunities

for sharing processing effort between different user subscriptions.

There exists a large number of tools and applications enabling RSS/Atom feed aggregation, which can be classified according to many criteria. We underline the following ones: i) the classification model of subscribed feeds, ii) feed discovery and feed search capabilities, iii) social network integration and iv) other mash-up capabilities.

Google Reader Google Reader [URLa] is an RSS feed registry and feed reader which allows Google users to create a personalized hierarchy of RSS feed collections. Feeds can be discovered by keyword search and new feeds can be added manually. Feed collections can be searched for keywords and be published as new RSS feeds representing the union of all feeds in the collection. Google Reader also integrates the Google social network features for sharing and recommending feeds.

Figure 1.1 shows a screenshot of Google Reader interface. Users can select a feed from the left panel and visualize its contents on the main panel, or a collection of feeds (a folder) to visualize all their items together. They can sort the items by date or by relevance. They can add new feeds through the button at top left or search for other feeds via the search bar.

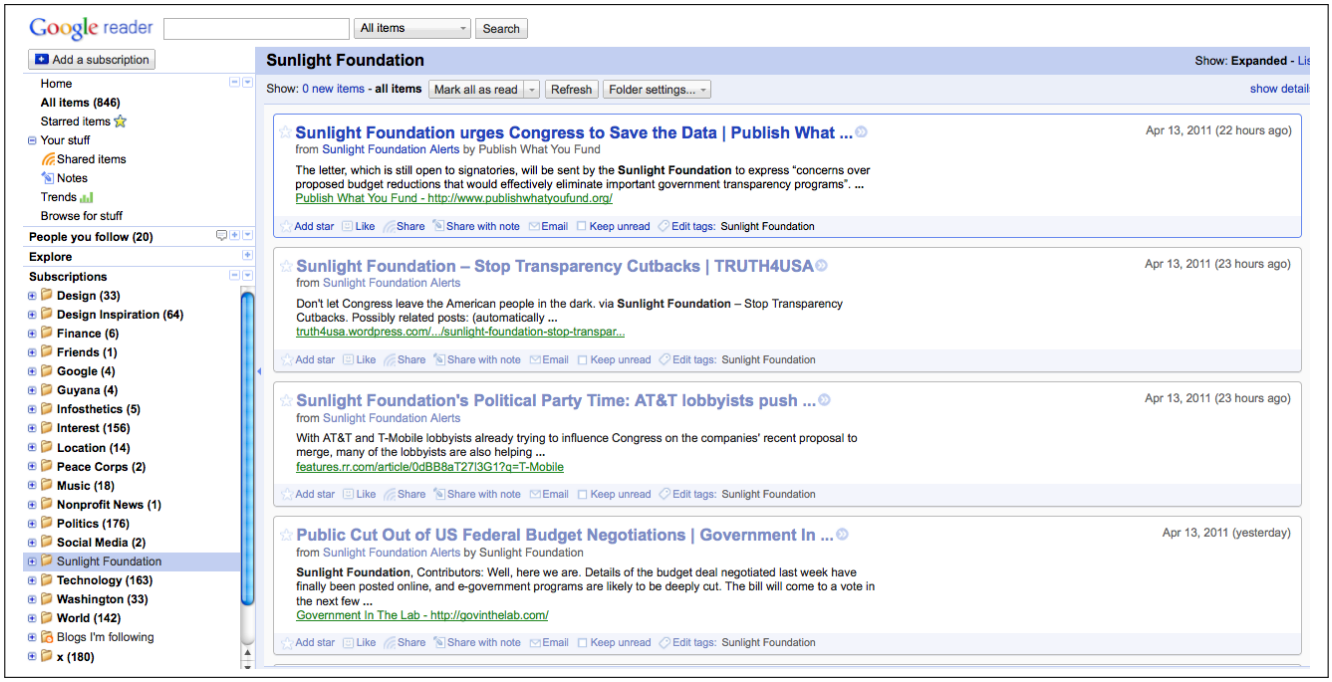


Figure 1.1 – *Google Reader*'s screen dump

Yahoo! Pipes Yahoo! Pipes [URLb], like ROSES, goes beyond the traditional concept of feed aggregators. Users may build their own feeds (*pipes*) through a user-friendly visual programming interface. A *pipe* is a sort of algebraic tree composed of customizable operators, including user inputs, union, filtering, count, rename, URL builders, translate, tokenizer, replace, regex, etc. *Pipes*' sources may also be of very different nature: RSS/Atom feeds, WebServices, HTML, static text, Yahoo Search results, other users *pipes*, etc.

Figure 1.2 illustrates a pipe that searches the word "RSS" (parameter box on the right) in the feed *http://example.com/feeds* (URL-builder). The URL-builder box defines which feed to collect and how to fetch it. Then, the feed is fetched from the given URL of the URL-builder. Finally the feed is sorted by the pubDate attribute of the item and outputted. The output can be seen interactively by selecting the *Pipe Output* tab.

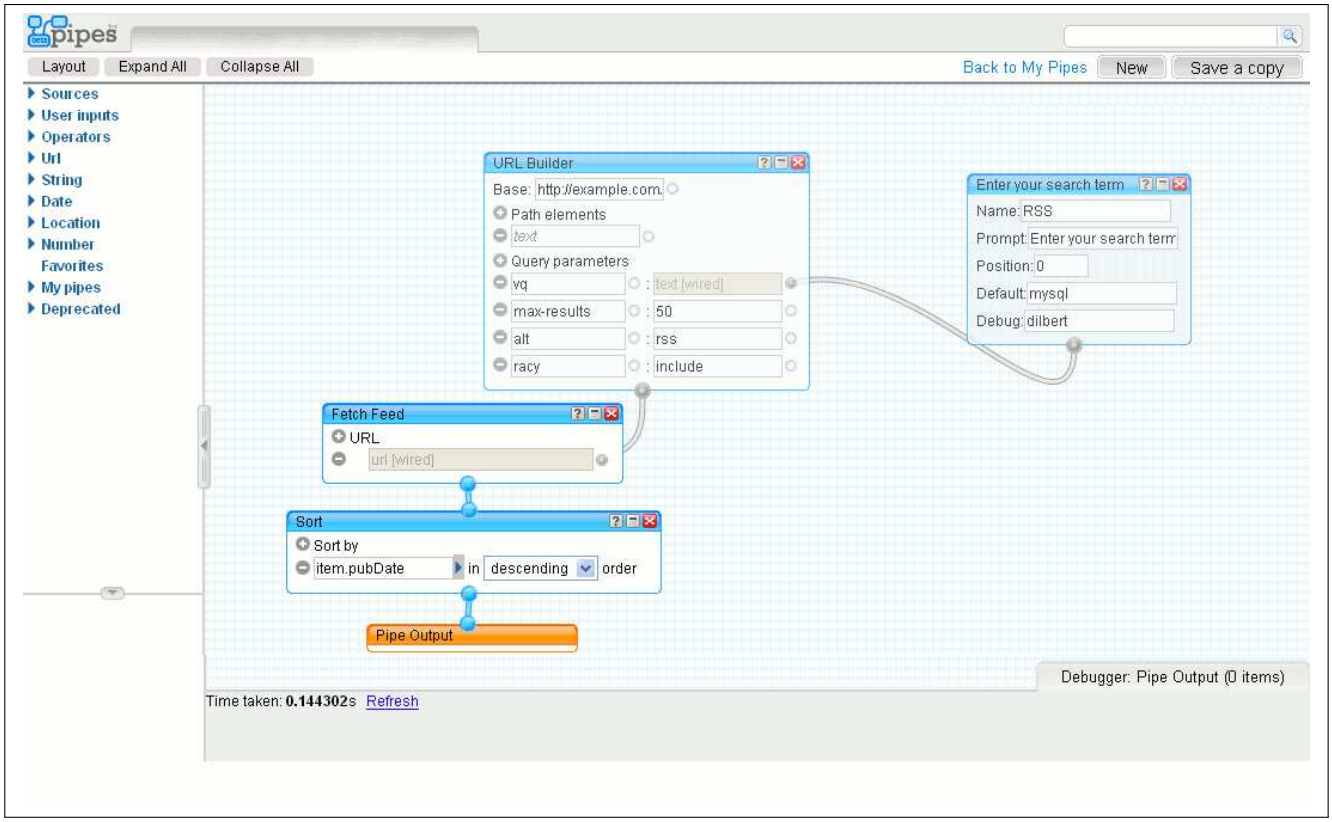


Figure 1.2 – *Yahoo! Pipes*'s screen dump

Yahoo! Pipes is an example of a more powerful mash-up system for the structural extraction and composition of RSS feeds and other external web services. While Yahoo! Pipes offers a similar functionality as the *ROSES* system, as far as we know (few information is publicly available), they use a static execution approach, *i.e.* pipe queries are periodically evaluated or

on user-demand. To our knowledge, each Yahoo! pipe is executed separately and Yahoo! Pipes does not implement any multi-query optimization technique for reducing execution cost.

Digg Digg [URLc] is a social news website, its success lies in its simple yet original recommendation system. Users may submit web content references for general consideration. Then the other members of the community can vote news items up or down (digging or burying respectively). Thus, items with more positive votes appear first in the main page of the user.

NetVibes NetVibes [URLd] is a popular feed aggregation portal providing a collection of graphical widgets to compose a personalized information space. NetVibes is an example of general mash-up system.

Feedzilla Feedzilla [URLe] is another RSS registry which collects and categorizes the content of thousands of RSS feeds. Feedzilla supports the building of user interface widgets publishing a chosen category of news personalized by some filter conditions.

NewsIsFree NewsIsFree [URLf] is a feed registry and feed reader where users can browse feed contents by feed category, name, date or language. Feeds can be searched by keyword search on the name and the description. Once the keyword entered, a list appears on the screen. The user needs to copy and paste into the aggregator each web address for each web site he/she wants to sign up to.

Table 1.1 shows a non exhaustive list of RSS registries classified according to four main different classes of aggregation services. All these tools are generally based on a simple keyword search on the item contents and a non-personalizable set of hierarchically organized categories for describing and retrieving feeds. Aggregation is limited to the visual aggregation of widgets in a web page (NetVibes, Feedzilla) or the creation of collections (Google Reader). Opposed to graphical and procedural feed aggregation techniques, *ROSES* thoroughly exploits declarative data and knowledge modeling for organizing, filtering and aggregating continuous RSS data in a complex application domain.

1.2 Pub/sub systems

Publish/subscribe is a messaging paradigm where message senders (*publishers*) do not send messages to specific receivers (*subscribers*) beforehand defined. Instead of that, published

Aggregator	Classification	Feed Search	Others
Google Reader	topic hierarchy	category, description	social network, recommendation
Digg	category list	content	social network, recommendation
Netvibes	category list	keyword	widgets, mash-up
Feedzilla	topic hierarchy	category, keyword	widget generator
NewsIsFree	category	keyword	
Feedsee	topic	topic, keyword	
Search4rss	-	keyword	feed discovery
Syndic8	topic hierarchy	keyword	
Yahoo! Pipes	-	content, structured	extensible mash-up algebra

Tableau 1.1 – RSS Feed Registries and Aggregators

messages are classified into categories to which receivers have subscribed. In our context we can define the process of publish/subscribe as follows: a system publishes a stream of textual information items (collected from one up to thousands of sources). Then users can subscribe to this stream by defining different kinds of filtering queries.

The main problem we find in publish/subscribe systems is that we have to process thousands of keyword-queries over a possibly high rate of message streams. In the context of DSMSs, this problem might be seen as the problem of processing a unique very high rate datastream with a huge number of filtering operators applied to this datastream.

There is a major line of research in the context of *publish/subscribe* systems, that is on subscription indexing. The main goal of these indexes it to efficiently detect all subscriptions which are relevant to an incoming published item. In most cases relevance is defined using a broad matching semantics where the incoming item must contain all keywords in the subscription. Some systems allow users to define more complex filtering predicates using conjunction and disjunction. The literature mainly distinguishes between two indexing schemes for counting explicitly vs implicitly the number of contained keywords: *Count-based* (CI) and *Tree-based* (TI), respectively [HKC+12].

The Count-based indexes are usually implemented through inverted lists. Two models are commonly adopted for subscriptions and items with inverted lists: a boolean model and a vector space model. Inverted lists maintain a directory D of all words extracted from subscriptions placed in a different cell $D[i]$. Each word $D[i]$ is associated to an inverted list L_i that contains all subscriptions s_{ij} that contain this keyword. When a new item is received, we first extract the different words q_k $(0 \leq k < N)$ from the item. For each q_k we retrieve the corresponding

inverted list L_k, and the subscriptions that should be notified are subscriptions in $\bigcup_{0 \leq k < N} L_k$. The whole technique is depicted in Figure 1.3 [CCdM+].

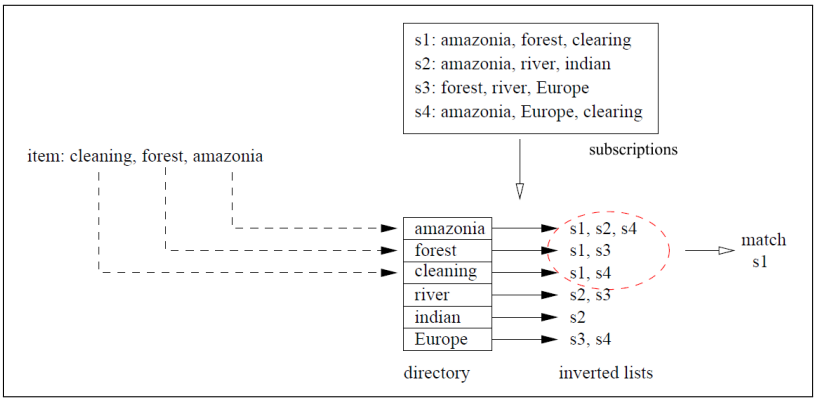


Figure 1.3 – Basic use of inverted lists in pub/sub

The Tree-based approaches (TI) propose tree structures based on a partial order defined by containment relationships. Figure 1.4 illustrates an example of a prefix tree built on a set of subscriptions. The root of the tree corresponds to the empty prefix. Each node n is the root of a subtree that indexes all the subscriptions that start with the sequence corresponding to the tree traversal until n. When a new item is received, a tree traversal allows determining the set of matching subscriptions. We start by the root and read one event after another. Then for each node n reached, the item is forwarded to its children if the input event match the node predicate. At each node, if there are many subscriptions that correspond, a matching notification is produced.

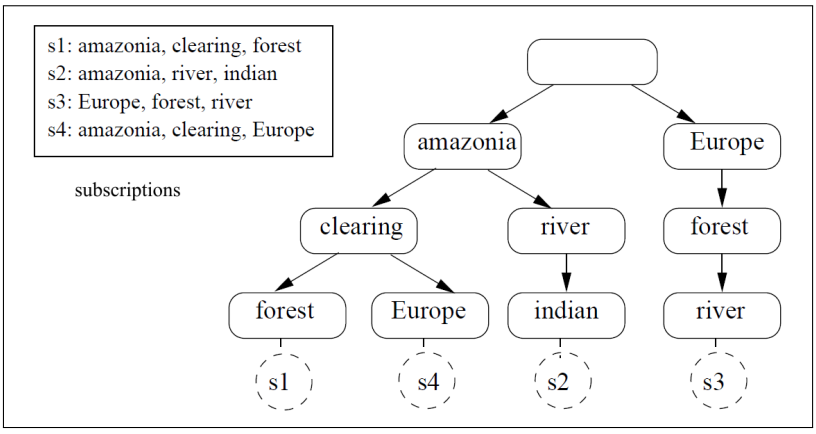


Figure 1.4 – Basic use of prefix tree in pub/sub

Since memory requirement is larger for *TI* than for *CI* systems, related works mostly rely on CI schemes rather than on TI structures. For instance, Le Subscribe [PFL+00] system uses

a CI index on predicates where subscriptions are matched by counting the item keywords they contain. In a naive solution all partially matching subscriptions are visited, so a goal is to reduce the search space by grouping subscriptions according to their size. Aguilera *et al.* propose in [ASS+99] a TI approach based a two phases matching technique for conjunctive subscriptions assuming a fixed total ordering among subscription predicates. The pre-processing phase creates a matching tree over the subscription predicates, in which nodes are predicates. This TI matching time complexity is sub-linear, with respect to the number of indexed subscriptions.

Few Pub/Sub systems have been proposed for keyword-based subscriptions. The most remarkable is the SIFT selective dissemination of text documents [YGM99]. Garcia-Molina *et al.* propose alternative indexes based on disk implementations of the ranked key *counter-less inverted list*, the *regular trie* and the *regular ordered trie*.

Special mention should be made to [HKC+12] in the framework of the *ROSES* project. Hmedeh *et al.* suggest three new indexing techniques based on inverted lists and on an ordered trie called POT (a *Patricia Trie*). These indexes implement various counting techniques in order to prune as early as possible non matching subscriptions. For small vocabularies POT's matching time is one order of magnitude faster than the best inverted-list index. While for large vocabularies, both exhibit the same matching time order.

The ROSES system is based on the *publish/subscribe* mechanism, *i.e.* users can subscribe to publications generated by ROSES queries. Nevertheless we go beyond the *publish/subscribe* paradigm by allowing users to create their own publications through complex declarative queries. This is why the ROSES fits better in the context of datastream management systems, where we deal with the problem of generating optimal execution plans capable of processing thousands of complex publication queries in parallel.

1.3 Datastream Management Systems

A *DataStream Management System* (DSMS) is a set of programs that provides the maintenance and querying of continuous datastreams. A conventional database query executes once and returns a set of results at a given point in time. In contrast, a continuous query is continuously executed as new data enters the stream, and generate itself a continuous stream of items as its result. In a more general sense, continuous queries can also be characterized as views which continuously propagate the insertion of updates into a database.

Over the last fifteen years, many prototypical DSMSs have been developed by database research groups, including MIT/Brown's *Aurora/Borealis* [ACc+03], *CAPE* [RDS+04], *Gigas-*

cope [CJSS03], *PIPES* [KS04], Stanford's *STREAM* [MWA+02] and Berkeley's *TelegraphCQ* [CCD+03b] (based on the earlier CACQ System). Stream processing engines have also gained recognition as commercial products. *StreamBase Systems Inc.* is a university spin-off from M.I.T., Brown University and Brandeis University based on findings from the Aurora and Borealis project, Gigascope is used at AT&T.

In this Section we give a survey of some DSMSs. We can classify them according to five axes:

1. the kind of data they handle,
2. the type of language (visual/declarative) and operators they support (type of joins, windowing...),
3. their theoretical foundations and formal semantics,
4. the execution model used to process queries, and
5. the kind of multi-query optimization they propose (at which execution level).

A key element of multi-query optimization is the semantics foundation of the concerned continuous query language. Indeed, a continuous query language whose operators support the *snapshot-reducible* property offers a larger level of optimization possibilities. Snapshot-reducibility is a well-known concept from temporal databases that ensures that the properties of the operators of the relational model are preserved in their continuous counterparts if we can *reduce* the continuous operator to the corresponding relational operator. This makes possible to export many rewriting rules from the relational model to a continuous model. Snapshot-reducibility will be later defined in the logical algebra Section (2.2.2.2).

1.3.1 STREAM

In recent years various SQL-like query languages have been proposed to formulate continuous queries. CQL, used in the STREAM system, enriches native SQL with window constructs [ABW06]. One particularity of CQL is that streams are turned into time-varying relations in order to apply traditional operators over these relations. Then after applying the relational operators, the relations are transformed again into streams. For this, the authors introduce two new operators: *stream-to-relation* and *relation-to-stream* operator.

We illustrate the use of these operators through a query example extracted from [ABW06]:

```
SELECT Istream(*)
FROM PosSpeedStr [RANGE Unbounded]
WHERE speed > 65;
```

This query is defined on a stream called *PosSpeedStr* that contains different vehicle speed-position measurements and whose schema is: $(vehicleId, speed, xPos, dir, hwy)$, the vehicle id and its speed, position and direction in a highway. The query is composed of three operators: an unbounded window on the aforesaid stream (stream-to-relation), a relational filter operator (relation-to-relation) and an *Istream* operator defined on the result of the filter operator (relation-to-stream). The semantics of the unbounded window is that at every time instant τ, the window (the resulting relation) contains all speed-position measurements up to τ. The semantics of the *Istream* operator is that every new tuple in the relation produced by the filter operator is streamed out as the result of the query. Thus, this query defines a simple filtering operator over *PosSpeedStr*.

While *ROSES* and *STREAM* differ little on the implementation and *query execution model*, an important difference between them is the semantics of their query languages. STREAM relies on a stream to relation logical model, while ROSES is based on a fully-streamed query execution model. Moreover, the kind of data and operators are quite different. We consider structured textual data and STREAM is conceived for relational structured data. On the other hand, both systems converge on the execution model: a graph of physical operators, which are connected by inter-operator queues and a scheduling process is in charge of ensuring the processing of each operator. STREAM is implemented by a single thread for scheduling and the order of the operator evaluation is determined by a scheduling strategy [BBMD03]. In contrast, ROSES uses a multi-threaded architecture. Finally, the main difference between both systems is that *ROSES* is conceived to define queries over large collections of text streams (big unions), while *STREAM* is more data stream oriented.

1.3.2 Aurora

Aurora proposes an algebra called SQuAl. Instead of using textual query expressions, users construct query plans through a graphical interface, where boxes represent operators, which are connected via arrows indicating the dataflow [ACc+03]. An important difference with *ROSES* is that stream elements are not uniform, *i.e.* they are tuples consisting of application-specific data fields. SQuAl offers seven different operators: order-agnostic operators (filter, map and union) and order-sensitive operators (join, aggregation, BSort and resample).

It is not easy to compare Aurora's query semantics with ROSES, given that some operators (such as BSort) would not fit into our algebra as its output stream might violate our ordering requirement. Moreover, the majority of the SQuAl operators are not snapshot-reducible. As a consequence, query optimization through query rewriting is almost not possible. Aurora's

main target is to satisfy user-defined QoS specifications. The *Borealis* project extends Aurora towards distributed stream query processing.

1.3.3 PIPES

PIPES[1] is a library developed by Krämer *et al.* [Krä09] at the University of Marburg. PIPES is a flexible infrastructure providing the fundamental building blocks to implement a general-purpose query engine for datastreams. PIPES offers two ways to its users to create new datastreams: via a *GUI* that enables users to specify data flows by combining operators from its stream algebra in a procedural manner, or via a declarative query language close to *SQL*. This query language enhances the basic SQL syntax with the *window* constructs defined in SQL:2003 for OLAP functions. The PIPES algebra provides a stream-counterpart for every operator in the extended relational algebra (except for the sorting operator). Besides, this stream algebra furnishes a *sliding* window operator, which is the most frequently used type of window in real world applications. So the window functionality is separated from the other operators.

The authors are the first to apply the notion of *snapshot-reducibility* [SJS01] to the *datastream* context. They show that each of their algebra operators is snapshot-reducible. This property is the fundamental foundation of their query plan generation and query optimization techniques. Query optimization in PIPES is based on the concept of plan equivalence, *i.e.*, two query plans or subplans are *equivalent* when they generate *snapshot-equivalent* results. So they perform query plan optimization by applying traditional rewriting rules (*e.g.*, join reordering, selection push-down) to snapshot-reducible subplans.

Another important problem tackled by Krämer *et al.* [CKSV08] concerns the adaptive resource management. The stream characteristics and query workload of a DSMS vary over time, thus DSMSs have to manage as well their resources *adaptively*. The approach they propose for the adaptive resource management lies in adjusting the window sizes and time granularities to keep the use of system resources within predefined bounds. Thus, the query language proposes some extensions to define a *Quality-of-Service* range on the size of the windows and the time granularity. The system then exploits the statistics extracted from runtime queries to maximize the overall QoS while keeping window and granule sizes within these bounds.

1. for Public Infrastructure for Processing and Exploring Streams

1.3.4 TelegraphCQ

TelegraphCQ is a continuous query processing system developed at the University of Berkeley [CCD+03a]. Its predecessors are *Telegraph*, *CACQ* and *PSoup*. TelegraphCQ primarily addresses shared queries scheduling and resource management, adaptive query execution and Quality of Service (QoS) support. Operators are called *dataflow modules* in TelegraphCQ. The system proposes a *module* for every operator from the extended relational algebra, without any *windowing* operator. Communication between the dataflow modules may be individually configured to be synchronous (pull-based) or asynchronous (push-based).

The Telegraph's query processing is not based on traditional query execution plans. The query engine of Telegraph, called *Eddies* [AH00], employs adaptive routing modules to control dataflow tuples evaluation. The Eddies optimization strategy lies in continuously routing dataflow tuples to query operators.

1.3.5 XML-stream query engines

Even if RSS/Atom feeds may be represented using the XML data format, they exhibit a nearly flat structure, and it is more convenient to consider RSS/Atom items as textual relational tuples. Below we describe several significative *XML-stream* DSMSs proposed in the literature.

NiagaraCQ *Niagara* system endeavours to answer queries over distributed XML documents crawled across a large network [NDM+01]. It proposes an XML-based query language that facilitates join specification over XML documents and the construction of complex results. The *NiagaraCQ* subsystem [CDTW00] proposes some continuous queries sharing techniques based on subquery grouping. In this project, Viglas and Naughton [VN02] introduce a cost model based upon stream rates to enable query optimization over streaming data.

Niagara is conceived to process XML-streams of *non-textual* data, such as datastreams generated by RFID sensors or stock market datastreams. We work with datastreams that have a strong textual component. This difference in the type of handled data entails a substantial difference on the type of operators and, consequently, the costs models for each system diverge considerably.

OptimAX OptimAX is a project of the Gemo group at INRIA. The OptimAX system deals with *Active XML* (AXML) query optimization in a distributed setting. An Active XML document is a document including several WebService calls. These service calls are mainly XQuery queries returning an asynchronous stream of XML results. Abiteboul *et al.* [AMZ07, AMZ08]

have extended the AXML language with three new operators (*newNode*, *send* and *receive*). AXML Web Services may include recursive Web Service calls, thus many call activation strategies may be carried out. The authors propose a cost-based optimization model for AXML documents computation. Their approach consists in building an initial AXML execution plan and iteratively improving this plan through the use of different heuristics.

In [AM07] Abiteboul and Marinoiu propose a monitoring overlay for P2P networks. Their system called *P2P Monitor* allows to handle streamed monitoring data across the P2P network. *Peers* produce monitoring data that may be filtered and/or reused by other peers. In this article the authors mainly propose an efficient filtering technique and a stream reuse mechanism. The filtering technique is based on the *Atomic Event Set* (AES) [NACP01] and *YFilter* [DFFT02] algorithms. AES is used in a first step to evaluate simple filtering conditions, afterwards YFilter evaluates the complex queries if still needed.

While using a different approach in *ROSES* to optimize queries on continuous feeds, *i.e.* we optimize large sets of queries by filtering factorization, OptimAX proposes interesting clues to extend our optimization model to distributed settings.

ViP2P Another related project carried out at Gemo is the ViP2P project (Views in Peer-to-Peer). In this project Manolescu *et al.* [MZ09, KKMZ12] present a platform, called ViP2P, for the distributed and parallel dissemination of XML data in a peer-to-peer setting. They consider the problem in the context of structured P2P networks indexed through a distributed hash table (DHT). In such networks, the peers own XML documents which they share with each other. The peers can define XPath-based views (subscriptions) on the top of the global collection of documents, they can as well request *ad-hoc* queries on both the XML documents and the views. The approach they propose lies in the use of a DHT to index the view definitions, which allows to improve the processing of both subscriptions and *ad-hoc* queries. Moreover, after a view lookup the queries are locally optimized by each peer using traditional optimization techniques (push selections, join reordering...).

Both works, OptimAX and ViP2P, are strongly related to Pitoura's works on XML data management/dissemination in P2P networks [KP05, PAP+03]. For instance, in [LP08] Pitoura and Lillis propose a cooperative caching system for XPath query results in DHT-based structured P2P networks. The authors introduce a new distributed index based on the prefixes of the XPath queries. They consider two different approaches, a loosely-coupled index and a tightly-coupled index. Documents that are frequently queried are indexed and can be easily located by looking up the cache entries, while less popular documents are not indexed. The semantics of user queries evolves with time, thus cached results must be periodically updated/replaced.

Therefore they also propose a proactive cache replacement policy, such replacement policy is based on a *utilization value* attached to the trie branches of the index and which is updated as new queries arrive.

Xyleme *Xyleme* [NACP01] is a content-based management system for XML documents. Xyleme uses a trigger-based evaluation of continuous queries in order to monitor changes in HTML and XML documents. Similar to *Tapestry* [TGNO92], its continuous query semantics derives from the periodic execution of one-shot queries.

1.4 Multi-query optimization problem

Multi-query optimization (MQO) has first been studied in the context of DBMSs where different users could request *one-shot* queries simultaneously to a system. For instance, in a system where various users can send different queries within the same time interval, one possible execution strategy is *batching* the queries. However, a major problem with this approach is the effect on the response time. It is unacceptable to delay a user's query due to other more expensive queries. A better approach should consider reusing shared intermediate results between the queries [Sel88, SG90].

In the context of DSMS, MQO consists in exploiting the fact that a set of continuous queries can be evaluated more efficiently together than independently, because queries often share state and computation. Solutions using these observations are based on different mutualizing methods:

- *Predicate indexing*: used for indexing subscriptions in publish/subscribe systems, *e.g.* indexing by ranked posting lists [WGMB+09];
- *Join interval indexing*: used for sharing join computations, they index the intervals of the predicate values on join queries (*e.g.*, [AXYY09]);
- *Sharing states in global NFA*: used for sequence detection, *e.g.*, with *YFilter* [DFFT02] or with the pub/sub system *Cayuga* [DGH+06];
- *Join graphs*, *e.g.* [HDG+07], or *Similarity joins hashing*, *e.g.* [TRFZ07]; and finally,
- *Sub-query factorization*: with the *HA* algorithm [SG90], *NiagaraCQ* [CDTW00], *TelegraphCQ* [CCD+03a], *STREAM* [ABW06] and *RUMOR* [HRK+09]. [2]

In ROSES, we have followed the factorization approach which appeared to be the most promising for a cost-based multi-query optimization solution.

2. Table 1.2 summarizes these techniques and the systems using these techniques.

Techniques	Systems
Predicate indexing	[WGMB+09]
Join interval indexing	[AXYY09]
Sharing states in NFA	YFilter [DFFT02], Cayuga
Join graphs/ Similarity joins hashing	[HDG+07] [TRFZ07]
Sub-query factorization	STREAM, NiagaraCQ, TelegraphCQ, RUMOR

Tableau 1.2 – Multi-Query Optimization Techniques summary

The multi-query optimization through *predicate factorization* brings difficulties in identifying similar operations with similar parameters. It might be seen like a "matching" problem within a set of separate query plans where the goal consists in finding a good rewriting containing a set of factorized sub-queries which are shared in the processing of the original queries. This problem is known to be *NP-hard* [SG90] and has to be simplified to scale up those systems.

1.4.1 RUMOR

An interesting approach is that of the system *RUMOR* (RUle-based Multi-query Optimization fRamework). Hong *et al.* [HRK+09] propose the use of *multi-operators*, grouping the processing of similar operators and producing *multi-streams*, where the items are annotated according to the operations that the item has validated. Figure 1.5 illustrates the use of multi-operators and multi-streams. We consider two queries Q_1 and Q_2 defined on the same input stream S with two selection operators σ_1 and σ_2 and one *aggregation* operator α_1 (Figure (a)). Both selection operators are grouped into a multi-operator $\sigma_{1,2}$ producing two different streams (Figure (b)). Then, both aggregation operators α_1 are also merged into another multi-operator $\alpha_{1,1}$, but now the selection multi-operator $\sigma_{1,2}$ must produce a multi-stream instead of two streams (Figure (c)). This multi-stream contains all items that have evaluated to true at least one of the selection operators (σ_1 or σ_2), and they are tagged according to the predicates they have validated. This allows removing one of the aggregation operators (α_1). But new multi-operator $\alpha_{1,1}$ must consider the tags of the items to perform the aggregation function.

The main difference with our approach is that by combining several operators in a single operator, the processing of the subgraph corresponding to that set of operators requires to

be synchronous, whereas with our approach the processing of the execution graph remains completely asynchronous.

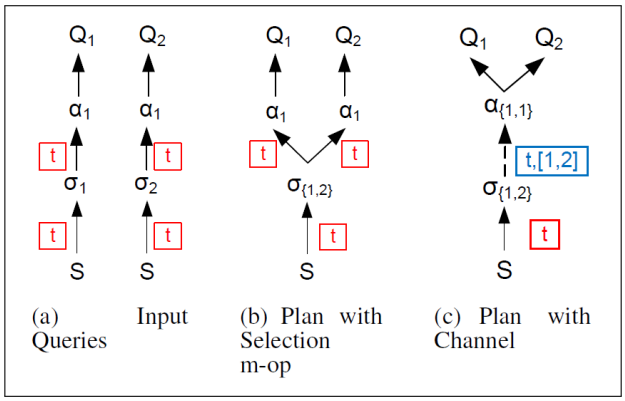


Figure 1.5 – Query plans in *RUMOR* [HRK+09]

1.4.2 Widom *et al.*

Another interesting approach is the one proposed by Widom *et al.* in [MSW07]. They propose a greedy optimization algorithm for continuous queries with expensive filters (*e.g.*, filters on image or video datastreams). They restrict to queries with *conjunctive* filters, thus a query is resolved once one of its filters evaluates to false or all its filters are true. Their query execution model consists in evaluating the query filters one-by-one in a given ordering such that all queries may be resolved as soon as possible. This ordering can be determined by a *fixed* strategy or an *adaptive* strategy. In the fixed strategy, the evaluation ordering is determined before the evaluation starts, whereas in an adaptive strategy the next evaluated filter is dynamically chosen in function of results of preceding filters on the evaluated item. The adaptive strategy is based on a decision tree composed of all the filtering predicates. So the best evaluation ordering corresponds to a path from the root of this tree to one of its leaves. The size of such decision tree is exponential in the number of queries and the number of filters, so they have proposed an algorithm to *partially* precompute it. An adaptive strategy involves an *execution overhead* on the evaluation process for finding at each step the appropriate filter to be evaluated next. In spite of this execution overhead, the adaptive strategy allows reducing the overall cost of resolving all queries for an item. Widom *et al.* have proved theoretically and experimentally that despite this overhead their adaptive strategy based on the precomputed decision tree is twice as fast as the fixed strategy.

The cost model proposed by the authors is based on three factors: i) filter cost, ii) selectivity and iii) *participation* (*i.e.*, the number of queries that contain a given filter). While in *ROSES*

we consider all filtering predicates to have a uniform cost (we only consider textual predicates), our optimization approach is general enough to integrate filtering predicates with variable costs. That is, our factorization technique can also incorporate this factor without problems, it keeps correct and optimal. On the other hand, in the same way as Widom *et al.* ([MSW07]) we consider the selectivity and participation factors in the optimal filtering tree building. This corresponds to our *benefit* function, detailed in Section 3.4.2.2. Finally, while their multi-query optimization technique is limited to simple conjunctive predicates, *ROSES* can handle any kind of boolean expressions as query predicates and take into account disjunctive and negation predicates on the optimization process. Another difference *w.r.t.* our approach is that their execution model is mono-threaded, while we take advantage of parallel predicate evaluation thanks to our multi-threaded architecture. In conclusion, the approach proposed by Widom *et al.* ([MSW07]) is similar to our approach except that they consider a setting with very *expensive* filters and we handle less expensive textual predicates.

1.4.3 Liu *et al.*

Yet another recent work similar to previous one is that of Liu *et al.* in [LPRY08]. The authors present two new algorithms, a *greedy* and a randomized *harmonic* algorithm, based on the well-known *edge cover* problem, with improved performance compared to the previous work [MSW07]. They view the *shared filter ordering* problem as an edge-coverage problem over a bipartite graph whose partitions are i) the set of queries and ii) the set of all filters appearing in the queries. Edges represent the containment of a filter in a query and all edges must be covered. However the difference with the classic edge-coverage problem is that once an edge is covered (*i.e.*, a filter is evaluated), a subset of queries might be resolved, thus a subset of edges is removed dynamically during the execution of the algorithm. The goal is to define an ordering of filters such that at each step (evaluation of a filter) we remove as much edges as possible. Like in previous work, the authors consider filter costs, filter selectivities and filter *popularities* (participation of a filter in the set of queries) in a *unit-price* function, used to determine the next filter to evaluate. They also prove that both algorithms they propose are near-optimal approximation algorithms with provably-good performance guarantees. Finally, they have experimentally evaluated these algorithms and show that the edge-coverage based greedy algorithm performs a 17% better than the greedy algorithm proposed by Widom *et al.* [MSW07].

Although focusing only on expensive conjunctive queries like [MSW07], an interesting point of this work is that the *harmonic* algorithm decides which filter will be evaluated next *randomly*,

thus filters associated with disjoint sets of queries could be executed in parallel. This opens the door to concurrent filtering evaluation.

1.4.4 Conclusion

Our approach addresses an original *multi-query optimization technique* (MQOT) 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 similar predicates, this solution brings an efficient technique to decrease CPU and memory use with a finer factorization than previous techniques.

The Steiner Minimal-Cost Tree is an NP-hard problem [CD02, HRW92] for which it is known that unless $P = NP$, it is impossible to produce by a polynomial-time algorithm an approximation factor better than $(1 + \epsilon)$ for some constant ϵ (called ϵ-approximation algorithms [Ihl91b]). Several approximation algorithms have been proposed in the literature for directed and undirected graphs [KMB81, Win87, Ihl91a, CCC+98, LCVA02, ZK02, DXYW+07, FKM+07] for various application contexts (*e.g.* computer networks, VLSI design, keyword-based database queries). These algorithms come with different memory requirements and approximation guarantees depending on the way they prune the search space for SMT solutions (*e.g.* distance network heuristics, spanning and cleanup techniques, dynamic programming, linear programming reductions, local search heuristics) and the general characteristics of the graph they are applied (*e.g.* directed or undirected, cyclic or acyclic, weighted edges/nodes). The VCA algorithm we propose 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. See Section 3.4 for more details.

Chapter 2

ROSES Query Language and Logical Algebra

In this Chapter we introduce the *ROSES* Query Language and the Logical Algebra that holds up this language. This is the first contribution of our work, since to the best of our knowledge, there is no proposal of a declarative query language for RSS/Atom feeds. The closest language was that of *Yahoo! Pipes*, which is in fact not a declarative language but a visual programming language.

The Chapter is organized as follows: the next Section (2.1) presents the Query Language and each one of its statements, after that Section 2.2 discusses first the *ROSES* Data Model and then the Logical Algebra and the properties of its different operators.

2.1 ROSES query language

The *ROSES Language* provides three kinds of statements: two DML [1] instructions (to register new data sources and to subscribe user-defined publications) and one query-like instruction (to create a new publication through a continuous query). Namely, we have:

1. **`REGISTER`** statements, to register new information sources in the system (in order to be used later by the *publication language* and by the *subscription language*),
2. **`CREATE`** statements, to define new publications, and
3. **`SUBSCRIBE`** statements, to define subscriptions to registered sources and/or to created publications.

In this Section, we describe all three languages (registration, publication and subscription), we focus however on the *publication language* since we are interested in the optimization possibilities of the publication queries.

1. Data Manipulation Language

2.1.1 Registering sources

ROSES allows users to register any kind of XML data sources which are continuously producing new information items. These sources may be (1) *external* sources: existing RSS/Atom feeds on the Web, Web Service request calls (executed periodically), or database queries; and (2) *internal* sources, *i.e.* internally user-defined publications.

The registered sources are transformed by a crawler (see Section 4.1) into continuous streams of items (see Section 2.2.1 for the *ROSES* Data Model). In the case of RSS and Atom sources this transformation is straightforward since the *ROSES* Data Model largely follows the RSS standard. For other kinds of data sources, transformations have to be defined by the user through the use of XSLT style-sheets.

We illustrate **register** instructions with three examples. The first example illustrates the most frequent case where a user registers an RSS syndication feed (The New York Times) by defining the URL of the feed and associating a name which can be used instead of the the URL for defining publications:

```
REGISTER FEED "http://feeds.nytimes.com/nyt/rss/HomePage" AS nytimes;
```

The second one allows to define a weather WebService as a data source. In this example we call a *Yahoo! Developer* WebService providing weather information for a given location $w = 615702$ (Paris) and in a given format $u = c$ (Celsius):

```
REGISTER FEED "http://weather.yahooapis.com/forecastrss?w=615702&u=c" EVERY 4 HOURS
  APPLY "yahooweather2roses.xsl" AS WeatherInParis;
```

The XML document returned by this call contains many data in a custom format (see below). Thus, we transform (wrapping) the data of this XML document into the *ROSES* format through the style-sheet *yahooweather2roses.xsl*. Furthermore, the user can explicitly specify the refresh frequency of the WebService (*every 4 hours*).

```
<item>
  ...
  <geo:lat>48.86</geo:lat>
  <geo:long>2.34</geo:long>
  <yweather:condition text="Fair" code="34" temp="25" date="Mon, 23 Jul 2012 7:30 pm CEST" />
  <yweather:forecast day="Mon" date="23 Jul 2012" low="16" high="25" text="Clear" code="31" />
  <yweather:forecast day="Tue" date="24 Jul 2012" low="17" high="28" text="Sunny" code="32" />
</item>
```

The last example shows how to materialize an existing ROSES publication featuring a transformation. This transformation may extract the items referenced by an annotation join and insert them into the description field of the item:

```
REGISTER FEED QueryWithJoinOperation APPLY "annotation2description.xsl"
  AS FinalPublication;
```

2.1.2 Publication language

The *ROSES publication language* has been designed to fulfill the following three desiderata:

- to be expressive but simple to use,
- to facilitate most common aggregation and filtering operations, and
- to support real scale web syndication systems.

In RSS/Atom syndication, the most frequently used form of aggregation queries is to collect and filter items originating from a large number of RSS/Atom feeds. A unique feature of the ROSES publication language is its ability to (semi-)join items of different feeds by keeping track of the matching items under the form of annotations.

Users can formulate queries using four types of operations: **selection**, **union**, **window** and **join**. We have excluded *transform* operations from *publication query language* in order to simplify the query optimization process. In fact, transform operators imply a strong constraint when trying to reformulate query execution plans. Transform operators rarely exhibit commutativity with the other operators, which makes it difficult to move them within the query plan. So, transform operations may only be performed at *registration* or *subscription* level.

The publication queries are composed by three parts for (i) naming the new feed, (ii) defining the input feeds and (iii) applying some filtering conditions. Let's consider the following two query examples for illustration:

```
CREATE FEED NewsOfSyria
FROM nytimes | cnn | telegraph
WHERE title CONTAINS "syria" OR title CONTAINS "assad";
```

This query defines a new publication *NewsOfSyria* as a feed containing all items of the registered feeds *nytimes*, *cnn* and *telegraph*, whose title contains either the word *syria* or the word *assad*. This is an example of queries corresponding to the large class of queries handled by publish/subscribe systems. The main difference between *ROSES* and pub/sub systems like *Le Subscribe* [PFL+00] is that in *ROSES* users subscribe to feeds generated by publication queries,

whereas in pub/sub systems users define complex filtering subscriptions on a unique stream of items generated by a predefined feed collection.

```
CREATE FEED MessiFeed
FROM (eurosport AS $e | fcbarcelonaBlog) AS $u | facebookMessi
WHERE $e[author <> "diego"] AND
      $u[title CONTAINS "messi"];
```

This second example shows the nesting possibilities of the union operation, as well as the possibility of using variables to refer to individual feeds or *groups* of feeds. It defines a new feed (*MessiFeed*) that contains all items of Messi's facebook newsfeed, and all items talking about *Messi* on *eurosport* and *fcbarcelonaBlog* sources, except those published by *Diego* on *eurosport*.

Users can define *publication queries* on registered sources (as in the two previous examples) or directly on URLs (sources are implicitly registered with a default name). Users also can use existing publications to define new publications, such a mechanism of feed composition is illustrated on the following example:

```
CREATE FEED FrenchSports
FROM FrenchFootball | FrenchRugby | FrenchBasket;
```

The following example finally illustrates an original way of generating new items combining the information of several different items (through a join-window query):

```
CREATE FEED MyMovies
FROM allocine AS $a
JOIN LAST 3 WEEKS ON MyFriendsTweets
     WITH $a[title SIMILAR WINDOW.title]
WHERE $a[description NOT CONTAINS "julia roberts"];
```

This query defines a join operation between two feeds, *allocine* and *MyFriendsTweets*. In fact, the latter one (*MyFriendsTweets*) is already a *ROSES* publication aggregating a collection of feeds: the RSS feeds produced by several friends of a given user on *Twitter*. The query defines: (1) a window of three weeks on *MyFriendsTweets* and then a join between *allocine* and this window (on *MyFriendsTweets*) based on the similarity between the titles of their corresponding items; and (2) a filter on the *allocine* feed (description does not contain "julia roberts"). *MyFriendsTweets* items are stored in the window during three weeks. We note I the set of items stored in the window at some moment. When the feed *allocine* publishes a new item i^{new}, we evaluate the join predicate on this item and every item i appearing in the window

($i \in I$). When the join predicate evaluates to true, the item of *allocine* (i^{new}) is annotated with the corresponding item i. This way this query allows to enrich a given feed (*allocine*) with items coming from a social network (*Twitter*). We show in Table 2.1 an example of the items that may produce this query.

Another example of join query is the next one:

```
CREATE FEED MyFriends
FROM (TwitterFriend1 | FlickrFriend1 | TwitterFriend2 | BlogFriend2 | ...) AS $friends
JOIN LAST 2 WEEKS ON (MyTwitter | MyFlickr | ...)
     WITH $friends[item CONTAINS WINDOW.keywords];
```

In this query the user defines a window on a collection of feeds produced by himself in many different social services (*Twitter*, *Flickr*...). Then he joins the contents of the window with another collection of feeds corresponding to different feeds produced by his friends. Thus, this user may be automatically notified when one of his friends publishes anything related to one of his recent posts.

In summary, *publication queries* contain three clauses:

- A mandatory **FROM** clause, where users specify the input feeds (called *primary feeds*) that produce the items of the output feed. Feeds are composed by union and it is possible to define named unions of feeds which can be referenced in the **JOIN** and **WHERE** clauses of the expression.
- Zero, one or more **JOIN** clauses, each one specifying a join with a *secondary feed*, defined in the same way as in the **FROM** clause. Joins are always defined between a single feed or a feed union and a window over secondary feeds. That is, in a **JOIN** clause users define a *secondary feed* and they apply a windowing operation on this feed, then they define a join predicate between one feed (or feed group) appearing in the **FROM** clause and the aforementioned window. We define two kinds of windows, time-based windows (*e.g.*, **LAST 3 WEEKS**) and count-based windows (*e.g.*, **LAST 50 ITEMS**). Finally, join predicates are binary atomic predicates, which relate one attribute of *primary feed*'s items to one attribute of window's items (*e.g.*, `title SIMILAR WINDOW.title`). In this way, *secondary feeds* only produce annotations (no output) to *primary feeds*' items. Each item of a *primary feed* is annotated by all items in the *secondary feed* window which satisfy the join condition. The *annotation join* behaves as an *inner-join*: if no such *secondary* items exist, the item is removed from the result.
- An optional **WHERE** clause, that enables users to define filtering conditions on the *primary* and *secondary feeds*, defined in the **FROM** and **JOIN** clause respectively. Filtering predicates

AlloCiné feed

```
...
<item>
  <roses:id>1000</roses:id>
  <title>The Dark Knight Rises</title>
  <link>http://www.allocine.fr/film/fichefilm_gen_cfilm=132874.html</link>
  <description>Il y a huit ans, Batman a disparu dans la nuit : lui qui était un
    héros est alors devenu un fugitif. S'accusant de la mort du procureur-adjoint
    Harvey Dent, le Chevalier Noir...</description>
  <pubDate>Tue, 24 Jul 2012 22:00:00 GMT</pubDate>
</item>
```

MyFriendsTweets feed

```
...
<item>
  <roses:id>885</roses:id>
  <title>carnage à la premiere de the dark knight</title>
  <description>carnage à la premiere de the dark knight</description>
  <pubDate>Sat, 21 Jul 2012 16:07:11 +0000</pubDate>
  <link>http://twitter.com/andresiniesta8/statuses/223448426682138624</link>
</item>
...
<item>
  <roses:id>860</roses:id>
  <title>the dark knight la semaine prochaine !!</title>
  <description>the dark knight la semaine prochaine !!</description>
  <pubDate>Tue, 17 Jul 2012 22:11:29 +0000</pubDate>
  <link>http://twitter.com/chavezcandanga/statuses/226439212038242304</link>
</item>
...
```

MyMovies feed

```
...
<item>
  <roses:id>1000</roses:id>
  <title>The Dark Knight Rises</title>
  <link>http://www.allocine.fr/film/fichefilm_gen_cfilm=132874.html</link>
  <description>Il y a huit ans, Batman a disparu dans la nuit : lui qui était un
    héros est alors devenu un fugitif. S'accusant de la mort du procureur-adjoint
    Harvey Dent, le Chevalier Noir...</description>
  <pubDate>Tue, 24 Jul 2012 22:00:00 GMT</pubDate>
  <roses:annotations>
    <roses:annotation join-id="530" item-id="885" />
    <roses:annotation join-id="530" item-id="860" />
  </roses:annotations>
</item>
```

Tableau 2.1 – An example of *MyMovies*'s input feeds and result

are boolean expressions formed by atomic predicates on the item values, see Data Model in Section 2.2.1, (=, <>, <, **CONTAINS**...) using the logical operators **NOT**, **AND** and **OR**.

We give in Table 2.2 a simplified version of *ROSES Query Language*'s grammar in EBNF [2] notation. The complete language grammar is presented in Appendix A.1.

```
  <publication-query> ::= "CREATE" "FEED" <publication-name>
                          "FROM" <union>
                          ( <join-clause> )*
                          [ <where-clause> ]
              <union> ::= <flow> ( "|" <flow> )*
               <flow> ::= ( <url> | <source-name> | <publication-name> | "(" <union> ")" )
                          ( "[" <selection-predicate> "]" )*
                          [ "AS" <variable> ]
        <join-clause> ::= "JOIN" <window-predicate> "ON" <union>
                          "WITH" <join-operation>
     <join-operation> ::= <variable> "[" <join-predicate> "]"
       <where-clause> ::= "WHERE" <selection-operation> ( "AND" <selection-operation> )*
<selection-operation> ::= <variable> "[" <selection-predicate> "]"
```

Tableau 2.2 – Simplified grammar of the *ROSES* publication language

2.1.3 Subscription language

The *ROSES subscription language* is used for declaring subscriptions to existing *source* or *publication* feeds. A subscription essentially specifies a feed, a notification mode (RSS, mail, SMS, etc.), a frequency (depending on the notification mode) and optional item transformations expressed also by XSLT style-sheets. Unlike transformations during feeds' registration, the output format on subscription transformations is free.

The following examples depict two subscriptions. The first one is an e-mail subscription to the *NewsOfSyria* publication, the second one is a subscription to the *MyMovies* publication using a transform operation:

```
SUBSCRIBE TO NewsOfSyria OUTPUT MAIL "Jordi.Creus@lip6.fr" EVERY 12 HOURS;
```

```
SUBSCRIBE TO MyMovies APPLY "annotation2description.xsl";
```

2. Extended Backus–Naur Form

2.2 Data model and logical algebra

The *ROSES Data Model* and *Operator Algebra* borrow from state-of-the-art datastream models and in particular from that proposed by Krämer in [Krä07], with specific modeling choices adapted to RSS/Atom syndication and aggregation. In this Section we first introduce the *ROSES* Data Model. Afterwards we describe the Logical Algebra accompanied by its *operator properties*, which will be used later in the *normalization* and *optimization* processes.

2.2.1 Data model

The *ROSES* Algebra $\mathscr{A}^{ROSES}$ is defined by a domain of data $\mathbb{D}$ and a set of operations $\mathbb{O}$: $\mathscr{A}^{ROSES} = (\mathbb{D}, \mathbb{O})$. $\mathbb{D}$ contains two types of data: *streams* ($\mathbb{S}$) and *windows* ($\mathbb{W}$): $\mathbb{D} = (\mathbb{S}, \mathbb{W})$. We define a *ROSES stream* as a full-fledged datastream of annotated *ROSES items*. More precisely, a *ROSES stream* $S \in \mathbb{S}$ is a (possibly infinite) set of *ROSES* elements $e = (t, i, A)$, where:

- t is a *timestamp*: t belongs to a **discrete** time domain $\mathbb{T}$ (as proposed in [BDE+97]) and represents the System acquisition timestamp of item i. For simplicity and without loss of generality, we map this timestamp discrete domain to the set of natural numbers $\mathbb{N}$ ($\mathbb{T} = \mathbb{N}$), *i.e.* to successive system acquisition moments we associate successive timestamps values in $\mathbb{N}$ [SW04].
- i is a *ROSES item*, which will be described in more detail below.
- A is an annotation set, referring to all items that have been joined with item i. More formally, a *ROSES annotation* is a set of couples (j, I), where j is a *join identifier* and I is a set of items. Annotations are produced by join operations, thus streams not resulting from joins have an empty annotation set. The semantics of annotation join is further detailed in the next Section.

Moreover, a *ROSES stream* presents the following two properties:

- the set of elements e for a given timestamp t is finite, and
- i behaves as a key in the set of elements $e = (t, i, A)$ for a given stream S, *i.e.*

$$\forall (t, i, A) \in S : \nexists (t', i, A') \in S, t \neq t'$$

Before defining the *ROSES item*, we define the *ROSES feed*. A *ROSES feed* corresponds to either an existing RSS/Atom (*source feed*) published on the Web, or to a *virtual feed* published through *ROSES* publication queries. Formally, a registered feed F is a couple $F = (d, S)$, where d is a *feed descriptor* and S is a stream of *ROSES items* as previously defined. Feed

descriptor d is a tuple, representing usual RSS/Atom feed properties: title, description, URL, etc. Thus, *ROSES feeds* may be placed at a "meta-model" level.

The *ROSES items* represent the information content conveyed by RSS/Atom items. 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 window* captures subsets of a stream's items that are valid over various time periods. More precisely, a window $W \in \mathbb{W}$ on a stream S is a set of couples (t, I), where t is a timestamp and I is a (finite) set of items of S. We say that I is the set of items valid at time instant t in window W. Note that, in this representation, (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. We may yet use an "unnested" representation where window W is defined as a (possibly infinite) set of couples (t, i). In this case i is not a key and each item in W appears exactly once during a continuous time period (all couples (t, i) for a given i must be consecutive in time). So:

1. $W \subset \mathbb{T} \times \mathbb{I}$, and
2. $\forall (t, i), (t', i) \in W, t \leq t' : \exists (t, i), (t + 1, i), ..., (t', i) \in W$

Finally, windows are used in *ROSES* only for computing joins between streams. *ROSES* uses time-based (last n units of time) and count-based (last n items) sliding windows.

The unnested definition of *ROSES streams* and *windows* facilitates the definition of operators with snapshot-reducible semantics, as we will see in the next Section.

2.2.2 Logical algebra

The *ROSES* query language is based on *five operators* for composing streams of *ROSES* items. We distinguish between *conservative* operators and *altering* operators:

- four *conservative* operators including selection, union, window and join: we call them conservative because they do not produce new items or change the contents of input items (join operators do not modify the items, they only add annotation references to the item), and

- one *altering* operator: *transform* which transforms the item contents and structure (similar to the relational projection).

A central design choice for the *ROSES language* is to rely only on conservative operators in the *publication queries* and to allow information transformation only during the registration of new sources and the subscriptions to publications. This choice is justified by the fact that conservative operators dispose of 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.

The publication queries (for which subscriptions exist) are translated into algebraic expressions as shown in Figures 2.1 and 2.2 for *MessiFeed* and *MyMovies* publications:

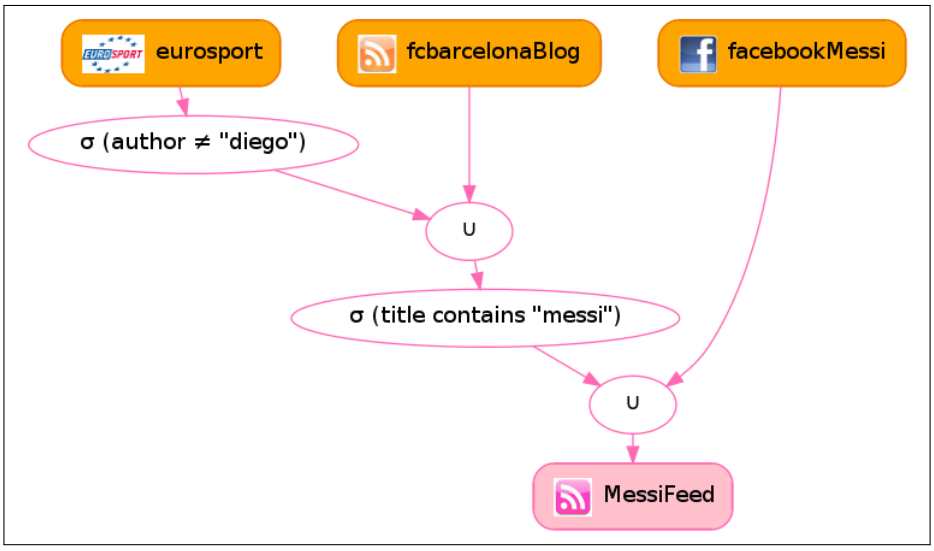


Figure 2.1 – A possible query execution plan for *MessiFeed* publication

The *logical operators* are defined in the next Subsection, finally their algebraic properties are described in Subsection 2.2.2.2.

2.2.2.1 Logical operators

The set of operations $\mathbb{O}$ of the *ROSES Algebra* is a set of functions defined over streams and windows. For instance, selection operator is defined as a function that has one stream as input and another as output ($\sigma : \mathbb{S} \rightarrow \mathbb{S}$), while windowing operator has a stream as input

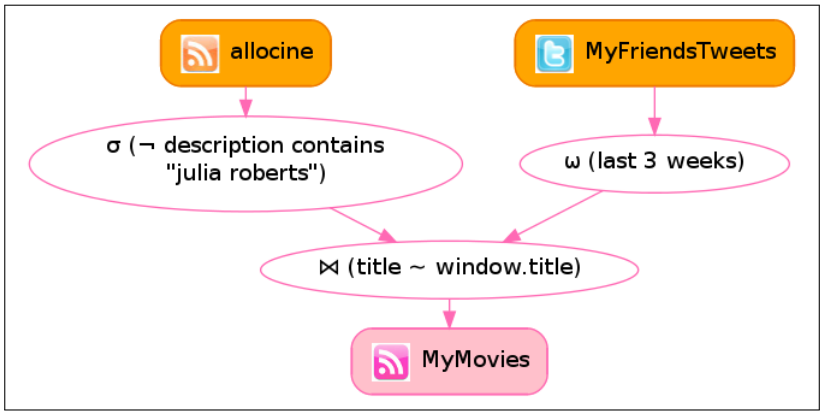


Figure 2.2 – A *MyMovies*' query plan

and produces one window as output ($\omega : \mathbb{S} \rightarrow \mathbb{W}$). We describe next all *ROSES* operators, for each one we give an informal and a formal definition, and a small example. For simplicity, annotation field A is omitted from the examples when its presence is irrelevant, this means all cases except join operator.

Selection

The filtering operation $\sigma : \mathbb{S} \rightarrow \mathbb{S}$ outputs only the elements of its input stream whose item satisfies a given selection predicate *pred*:

$$\sigma_{pred}(S) := \{(t, i, A) | (t, i, A) \in S \wedge pred(i)\} \quad \text{where:} \quad pred : \mathbb{I} \rightarrow \mathbb{B} \tag{2.1}$$

The *selection predicates* are boolean expressions (using conjunction, disjunction and negation) of *atomic selection predicates* that express a condition on an item attribute. Thus, depending on the attribute type, these atomic predicates may be:

- for simple types: comparison with a value (equality, inequality),
- for date/time: comparison with date/time values (year, month, day, etc.),
- for text: operations *contains* (word(s) contained into a text attribute) and *similar* (text similar to another text),
- for links: operations *references*/*extends* (link references/extends an URL or host) and *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 links in the item's attributes. Observe also that it is not possible to filter stream elements by their timestamp t or annotation attributes A, but

only on the item content.

The next example shows the contents of a stream S at different timestamps t (annotations are omitted). The right column shows the contents of a stream S' produced by a selection operator with a given predicate *pred* on S, *i.e.*, $S' = \sigma_{pred}(S)$. We can see that the contents of S' at every timestamp t represent a subset of the set of items corresponding to S ($\forall t \in \mathbb{T} : S'(t) \subseteq S(t)$):

t	$S(t)$	$\sigma_{pred}(S)$
1	i_1, i_2, i_3	i_1, i_3
2	i_4, i_5	i_5
3	i_6, i_7, i_8, i_9	i_6, i_7

Union

The *union* operator $\cup : \mathbb{S} \times \cdots \times \mathbb{S} \to \mathbb{S}$ returns all elements in its input streams preserving the timestamp ordering:

$$S_1 \cup \cdots \cup S_n := \{(t, i, A) | (t, i, A) \in S_1 \vee \cdots \vee (t, i, A) \in S_n\} \tag{2.2}$$

t	$S_1(t)$	$S_2(t)$	$S_1 \cup S_2$
1	i_1, i_2, i_3	j_1, j_2, j_3, j_4	$i_1, i_2, i_3, j_1, j_2, j_3, j_4$
2	i_4, i_5	j_5, j_6	i_4, i_5, j_5, j_6
3	i_6, i_7, i_8, i_9	j_7, j_8, j_9	$i_6, i_7, i_8, i_9, j_7, j_8, j_9$

Transformation

The *transformation* operator $\mu : \mathbb{S} \to \mathbb{S}$ modifies each input element following a given transformation function *map*:

$$\mu_{map}(S) := \{(t, i', A) | (t, i, A) \in S \wedge i' = map(i)\} \quad \text{where:} \quad map : \mathbb{I} \to \mathbb{I} \tag{2.3}$$

This *map* function may, for instance, translate the content of every item's field. μ is the only ***altering*** operator, whose use is limited to produce subscription results or new source feeds, as explained in previous Section (2.1).

t	$S(t)$	$\mu_{map}(S)$
1	i_1, i_2, i_3	i'_1, i'_2, i'_3
2	i_4, i_5	i'_4, i'_5
3	i_6, i_7, i_8, i_9	i'_6, i'_7, i'_8, i'_9

Windowing

The *windowing* operator is a function that takes a stream as input and produces a window in output ($\omega : \mathbb{S} \rightarrow \mathbb{W}$). Windowing produces a time-based or a count-based sliding window on the input stream according to the window specification.

Time-based windows Time-based windowing is defined as follows:

$$\omega_w^{time}(S) := \{(t', i) | (t, i, A) \in S \wedge t \leq t' < t + w\} \quad \text{where:} \quad w \text{ is window's size} \tag{2.4}$$

w is a time interval expressing an upper bound on the items publication date (3 months, 2 weeks, etc).

t	$S(t)$	$\omega_3^{time}(S)$
1	i_1, i_2, i_3	i_1, i_2, i_3
2	i_4, i_5	i_1, i_2, i_3, i_4, i_5
3	i_6, i_7, i_8, i_9	$i_1, i_2, i_3, i_4, i_5, i_6, i_7, i_8, i_9$
4	i_{10}, i_{11}	$i_4, i_5, i_6, i_7, i_8, i_9, i_{10}, i_{11}$
5		$i_6, i_7, i_8, i_9, i_{10}, i_{11}$

Count-based windows Our stream model is time oriented, thus time-based windows ω^{time} can be defined naturally. However, it is impossible to well define count-based windows ω^{count}. As stated by Arasu in [Ara06], count-based windows may cause ambiguous semantics: they may produce a nondeterministic output because the window size has to be exactly N elements, but multiple elements in the input stream may have the same timestamps.

In fact the problem is that in these models items are ordered by timestamp, but many items may share the same timestamp, thus we ignore the ordering of the items within the timestamp. That is why it may be impossible to determine which is the N^{th} last item. The solution that we propose is to consider *all* last timestamps until we reach the value of N items. Thus, we

avoid to do undeterministic crops within timestamps and this makes our solution deterministic. More formally, we define our count-based windowing operator as follows:

$$\omega_N^{count}(S) := \{(t', i) | (t, i, A) \in S \wedge t \leq t' \leq t_e \wedge \\ t_e = max\{t'_e \in \mathbb{T} || \{(t'', i'', A'') \in S | t \leq t'' \leq t'_e\}| \leq N\} + 1\} \tag{2.5}$$

t	$S(t)$	$\omega_5^{count}(S)$
1	i_1, i_2, i_3	i_1, i_2, i_3
2	i_4, i_5	i_1, i_2, i_3, i_4, i_5
3	i_6, i_7, i_8, i_9	$i_4, i_5, i_6, i_7, i_8, i_9$
4	i_{10}, i_{11}	$i_6, i_7, i_8, i_9, i_{10}, i_{11}$
5		$i_6, i_7, i_8, i_9, i_{10}, i_{11}$

We can see in previous example that for $t = 2$ there is no problem, the window contains all 5 items appearing in $t = 1$ and $t = 2$. In contrast, for $t = 3$ there is no way to know which is last 5^{th} item (i_4 or i_5), thus we take both of them, as a result the window contains 6 elements, and so on.

Join

The *join* operator takes a primary stream S and a window on a (secondary) stream W as input and generates a stream S' as output ($\bowtie$: $\mathbb{S} \times \mathbb{W} \rightarrow \mathbb{S}$). *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 $S \bowtie_{pred^J} W$ of identifier jid returns the elements in S for which the join predicate $pred^J$ is satisfied by a non-empty set I of items in window W, enriched by the annotation entry (jid, I). More precisely,

$$S \bowtie_{pred^J} W := \{(t, i, A') | (t, i, A) \in S \wedge A' = A \cup \{(jid, I)\} \wedge \\ I = \{i' | (t, i') \in W \wedge pred^J(i, i')\}\} \tag{2.6}$$
$$\text{where:} \quad pred^J : \mathbb{I} \times \mathbb{I} \rightarrow \mathbb{B}$$
$$jid \text{ is the join identifier}$$

t	$S(t)$	$W(t)$	$S \bowtie_{pred^J} W$
1	$(i_1, \varnothing), (i_2, \varnothing), (i_3, \varnothing)$	j_1, j_2, j_3, j_4	$(i_2, \{(jid, \{j_2, j_3\})\}), (i_3, \{(jid, \{j_4\})\})$
2	$(i_4, \varnothing), (i_5, \varnothing)$	$j_1, j_2, j_3, j_4, j_5, j_6$	$(i_4, \{(jid, \{j_1, j_2, j_5\})\})$
3	$(i_6, \varnothing), (i_7, \varnothing), (i_8, \varnothing), (i_9, \varnothing)$	$j_1, j_2, j_3, j_4, j_5, j_6, j_7, j_8, j_9$	$(i_7, \{(jid, \{j_1\})\}), (i_8, \{(jid, \{j_6, j_7, j_9\})\}),$ $(i_9, \{(jid, \{j_4, j_9\})\})$
4	$(i_{10}, \varnothing), (i_{11}, \varnothing)$	$j_5, j_6, j_7, j_8, j_9, j_{10}, j_{11}$	$(i_{10}, \{(jid, \{j_5, j_7\})\})$
5		$j_7, j_8, j_9, j_{10}, j_{11}$	

2.2.2.2 Snapshot-reducibility and rewriting rules

The **snapshot-reducibility** is a well-known concept in the temporal database community [SJS00]. It guarantees that the semantics of a relational, non-temporal operator is preserved in its more complex, temporal counterpart. In other words, a temporal operator may be emulated by applying its relational counterpart at every time instant. More precisely, for all $t \in \mathbb{T}$ the set of items associated to time t and produced by a temporal operator is equal to the set of items produced by the analogous relational operator on the set of the input items associated to the same t [BBJ98]:

$$\text{if } op^T(S) = S' \text{ and } \forall t \in \mathbb{T} : S'(t) = op^R(S(t)) \Leftrightarrow op^T \text{ is snapshot-reducible} \tag{2.7}$$

Our temporal operators of selection, union and join are snapshot-reducibles, thus the algebraic properties they have in the relational model may be exported to the temporal one. Recall that our annotation join operator do not alter the content of its input items, but it only add the references to the matching items. Further, the predicates of the selection operators do not concern the items annotated by joins, this makes it possible to commute joins and selections. Table 2.3 lists some of the algebraic equivalences derived from the *extended relational model*. We will see later their interest in order to normalize the publication queries (in Section 3.3), and then to reorganize the query execution plans and find the optimal ones (in Section 3.4).

Selection cascading	$\sigma_{pred_1}(\sigma_{pred_2}(S)) = \sigma_{pred_1 \wedge pred_2}(S)$
Union commutativity	$S_1 \cup S_2 = S_2 \cup S_1$
Union associativity	$(S_1 \cup S_2) \cup S_3 = S_1 \cup (S_2 \cup S_3) = S_1 \cup S_2 \cup S_3$
Selection distributivity over union	$\sigma_{pred}(S_1 \cup S_2) = \sigma_{pred}(S_1) \cup \sigma_{pred}(S_2)$
Selection-join commutativity	$\sigma_{pred}(S \bowtie_{pred^J} W) = \sigma_{pred}(S) \bowtie_{pred^J} W$
Join distributivity over union	$(S_1 \cup S_2) \bowtie_{pred^J} W = (S_1 \bowtie_{pred^J} W) \cup (S_2 \bowtie_{pred^J} W)$

Tableau 2.3 – Rewriting rules of algebraic trees

Chapter 3

Multi-query Processing and Optimization

There are two possible approaches we may consider in order to process the kind of queries we have seen in previous Chapter: by *periodic* processing or by *continuous* processing. The first one consists in crawling and storing the streams in regular databases, which afterwards are queried by periodically triggered (traditional one-shot) queries. In the continuous approach, a long-running global query execution plan is built, using the appropriate algebraic operators, connected by inter-operator queues. This query plan is able to process/evaluate the new items continuously arriving into the system.

This second approach enables to improve the system performances, since we do not store the data and thereby we do not need to access the disk. Moreover, the algebraic approach we have adopted offers important factorization opportunities. However, the processing of continuous queries entails many different challenges when we consider a large volume of queries. In particular, we underline two of these challenges: (1) the optimization of the operator scheduling techniques, and (2) the optimization of the overall query execution plan itself. In this work we mainly focus on the second problem which yields in turn many other problems: (i) the static optimization of a large number of continuous queries, or (ii) the dynamic optimization of a multi-query graph, *i.e.*, the reoptimization of the physical query graph at runtime.

This Chapter corresponds to the main contribution of this work. In the first Section (3.1), we introduce the concept of multi-query graphs, we describe the query engine of the *ROSES* system and we present our cost model. Section 3.2 describes the multi-query optimization problem and sketches our optimization approach which is based on *filtering factorization*. Section 3.3 is devoted to the query normalization, we introduce a formal definition of a *ROSES* query and of a normal form query, finally we present the concept of the global normal query graph. In Section 3.4, we describe our multi-query factorization technique for a static set of queries. We present our *simplified* cost model for the factorization of filtering predicates, and we introduce the concept of the *predicate subsumption graph*. Then, we present our different

factorization algorithms and data-structures (Section 3.4.2). In Section 3.5, we introduce our *runtime* optimization strategy. Finally, Section 3.6 presents our Generator of *ROSES* queries and the results obtained by the experimental evaluation we have conducted on the different algorithms we propose.

3.1 Query processing and cost model

In this Section we describe the ROSES evaluation model, in particular Section 3.1.1 introduces the notion of *Multi-Query Graph* through a graphic example, in Section 3.1.2 we describe the query execution engine, then we introduce the notion of flow rate in ROSES and present our execution cost model.

3.1.1 Multi-query graphs

In *ROSES*, 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 2.2.2 (union, selection, join, window and transformation). *ROSES* adopts a standard execution model for continuous queries [ABW06, CKSV08]. This query execution model 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 (in the case of join operations).

A query plan for a set of queries Q can then be represented as a directed acyclic graph $G(Q)$ as shown in Figure 3.1. Graph in Figure 3.1 illustrates one out of the several possible physical query plans for the following three publication queries p_1, p_2 and p_3 over six sources s_1, s_2, s_3, s_4, s_5 and s_6:[1]

- $p_1 = \sigma_1(s_1 \cup s_2)$
- $p_2 = (s_3 \cup s_4) \bowtie_1 \omega_1(s_5)$
- $p_3 = \sigma_2(p_1 \cup s_6)$

As we can see, the root of the graph is a dispatcher which generates for each source a read/write buffer which can be used by the different operators. Window operators produce a different kind of output, namely *window buffers*, which are consumed by join operators. Publication (*view*) composition by publish/subscribe pattern 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 transform operations are applied after the publication query evaluation.

1. Observe that p_3 queries the stream generated by query p_1.

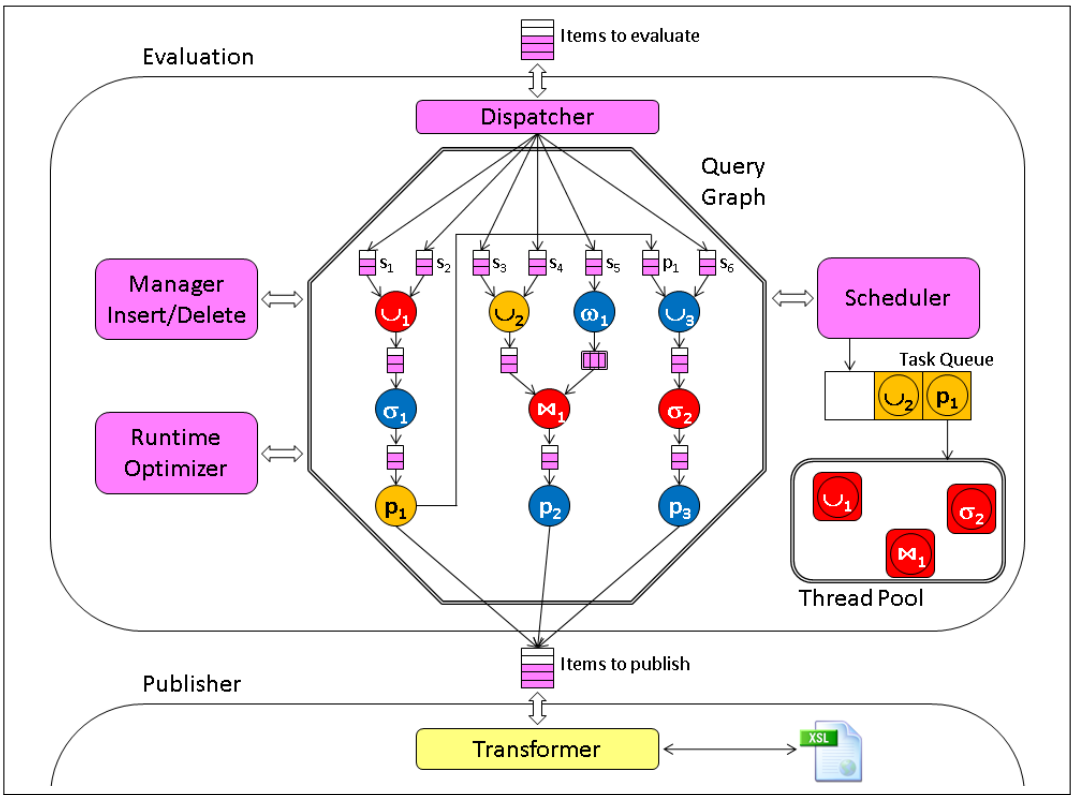


Figure 3.1 – Architecture of the ROSES evaluation engine

3.1.2 Query processing and cost model

This Section presents the query graph processing and the underlying cost model.

As explained in the previous Section, a set of publication queries is translated into a physical multi-query plan. Such a multi-query plan is a graph composed of physical operators connected by read/write queues (we call them *inter-operator queues*) and by window buffers (used by join operators). New items are continuously arriving to this graph (in a asynchronous way) and have to be consumed by the different operators. We have adopted a multi-threaded pipelining execution model which is a standard approach in continuous query processing architectures. In an ideal frame, each operator could be allocated to its own execution thread. However, with a big number of operators to be evaluated, the naive solution of attaching one thread to each operator rapidly becomes inefficient/impossible due to thread management overhead or system limitations [2]. Therefore only a subset of all the operators can be executed concurrently. We have adopted a *Thread Pool* approach. *Thread pools*, widely used in the context of web server applications, allow a number of tasks to be concurrently executed by a *fixed* number of threads. Task allocation consists in dynamically maintaining a waiting list defining an order over the tasks to be executed by the Thread Pool.

The query graph is observed by a *scheduler* that continuously decides which operators (tasks) must be executed (see Figure 3.1). This scheduler has at its disposal a pool of threads for executing in parallel a fixed (or barely varying) number of tasks. 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). These factors correspond to different strategies that may be applied by the *scheduler*. A good calibration of these parameters is essential to achieve the best evaluation performance. We have implemented two different scheduling strategies in the *ROSES* Prototype (a randomized strategy and a *round-robin* strategy). Both strategies avoid starving, a detailed description is given in Section 4.1.2. It is not difficult to adopt other allocation strategies, but that issue is outside the scope of this thesis.

In this setting, we rely on a cost model for estimating the resources (memory and/or 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. Thus our cost model leans on the rate of the flows produced by the different type of operators that compose the query graph. We define $rate(v)$ as follows:

2. Besides, current programming languages restrict the number of threads that can be created by process.

Definition 3.1. *We denote by* $rate(v)$*, the number of items produced per time unit (*publication rate*) by the vertex* v *of the graph.*

Table 3.1 sums up the different rates associated to each vertex type. This cost model describes the cost associated to the *physical* operators, which present a few differences *w.r.t* the logical operators introduced in Section 2.2.2.1. Namely, our physical union operator do not remove duplicate items from its input streams. Another difference with the logical operators concerns *count-based* windows. The physical operator implementing the *count-based* window produces a window buffer with *exactly* last N items, as opposed to its logical counterpart that was defined as returning a bounded window of at least N items.

v	$rate(v)$
s	$rate(s)$
$\sigma_{pred}(v')$	$selectivity(pred) \cdot rate(v')$
$v_1 \cup \cdots \cup v_n$	$\sum_{i=1}^{n} rate(v_i)$
$\omega^*(v')$	Null
$v_1 \bowtie_{pred^J} v_2$	$selectivity(pred^J) \cdot rate(v_1)$
$p(v')$	$rate(v')$

Tableau 3.1 – Production rate for each kind of operator

We have adapted the model presented in [CKSV08] and we define the cost of each operator v as a function of the publishing rate $rate(v')$ of its direct predecessor/s v' in the query graph. Each operator has three kinds of cost associated, *read cost*, *evaluation cost* and *write cost*. As we can see in Table 3.2, the cost of each operator mainly depends on the publishing rate of the input buffer/s of the operator. Thus, we define the total cost of a query graph $cost^{Total}(G)$ as the sum of read, evaluation and write costs of all operators composing the graph [3]:

$$cost^{Total}(G) = \sum_{v \in V(G)} \Big(cost^{Read}(v) + cost^{Eval}(v) + cost^{Write}(v) \Big)$$

Selection. The output rate of the selection operator corresponds to the rate of its input buffer reduced by the selectivity factor $selectivity(pred) \in [0, 1]$, depending on the selection predicate *pred*. Selection operator processes only one item at a time. Function $cost^{Eval}(pred)$ represents the average cost of evaluating a given predicate *pred* on a single item. This cost function depends solely on the complexity of the predicate and we assume that it is independent

3. We will see later that we search for globally minimizing this cost function in order to optimize the global query plan.

Vertex v	Read cost $cost^{Read}(v)$	Evaluation cost $cost^{Eval}(v)$	Write cost $cost^{Write}(v)$
s	0	0	0
$\sigma_{pred}(v')$	$rate(v')$	$cost^{Eval}(pred) \cdot rate(v')$	$selectivity(pred) \cdot rate(v')$
$v_1 \cup \cdots \cup v_n$	$\sum_{i=1}^{n} rate(v_i)$	0	$\sum_{i=1}^{n} rate(v_i)$
$\omega_w^{time}(v')$	$rate(v')$	$rate(v')$	$rate(v')$
$\omega_N^{count}(v')$	$rate(v')$	0	$rate(v')$
$v_1 \bowtie_{pred^J} \omega_w^{time}(v_2)$	$rate(v_1) \cdot rate(v_2) \cdot w$	$cost^{Eval}(pred^J) \cdot cost^{Read}(v)$	$selectivity(pred^J) \cdot cost^{Read}(v)$
$v_1 \bowtie_{pred^J} \omega_N^{count}(v_2)$	$rate(v_1) \cdot N$	$cost^{Eval}(pred^J) \cdot cost^{Read}(v)$	$selectivity(pred^J) \cdot cost^{Read}(v)$
$p(v')$	$rate(v')$	0	0

Tableau 3.2 – Read, evaluation and write costs of the different operator types

of the item contents/size. A more precise model could take account of the size of each item, but since items are, in general, small text fragments of similar size we ignore this detail in our model.

Union. The union operator generates an output buffer with an output rate corresponding to the sum of its input buffer rates. Our union does not remove duplicates from flows. A union operator implementing a *distinct* operation (*i.e.* removing duplicates) should be implemented using a sliding window of a fixed size. Then the union operator should check for each incoming item if it already appears in that window. So, without duplicate removal we assume that union has zero evaluation cost since each union operator is simply implemented by a set of buffer iterators, one for each input buffer. Reader may refer to Section 4.2 for a detailed explanation on these consumption iterators.

Window. The window operator transforms its input stream into a *window buffer*, where the size depends on (i) the time-interval w and the input buffer rate $rate(v')$, for *time-based* windows, or on (ii) the specified number of items N, for *count-based* windows. The only difference between the two types of windows is that when a time-based window is executed it must additionally detect and remove all out-of-date items from the corresponding *window buffer*. We consider that this buffer is sorted by timestamp and that the rate of items entering the buffer is roughly the same as the rate of those leaving the buffer. This means that the number of out-to-date items to check in the buffer is the same as those entering the buffer, *i.e.* a cost of $rate(v')$. Count-based windows avoids this problem by using a fixed sized FIFO queue, for instance, a *circular* buffer in which each new item is written into the next cell, thus with a zero evaluation cost.

Join. The join operator generates an output buffer with a publishing rate of $selectivity(pred^J) \cdot rate(v_1)$, where $rate(v_1)$ is the publishing rate of the primary input stream and $selectivity(pred^J) \in$

$[0, 1]$ corresponds to the probability that an item produced by v_1 joins with an item in window v_2 using join predicate $pred^J$. This is due to the behavior of the annotation join: an item is produced by the join operator when a new item i^{new} arrives on the main stream and matches at least one item in the window. The resulting item is generated by annotating the main item i^{new} with all matching window items. The processing cost of the join operator (read + evaluation + write costs) depends on the kind of window it is attached to. It also depends on the implementation of the physical operator. We assume a naive implementation of this operator (using a nested loop), however more sophisticated join implementations might reduce this processing cost like in [OU05], where inverted files are used in order to reduce similarity join's cost.

It is easy to see that the global cost of the execution plan (the sum of the costs of all operators) is strongly influenced by the operator ordering and the input buffer rate of each operator. We will describe in the following how we can reduce this cost function by pushing selection and join operators towards the source feeds of the global query plan.

3.2 Multi-query optimization problem

One of the main goals of our system is to scale in terms of number of publications. This means the system to be able to manage and process thousands of publication queries simultaneously. Query logs show that users frequently ask *similar* publications, *i.e.*, user-defined queries often share some computation and merging these computations enables the system to reduce computing costs substantially and thus increase the number of handled publication queries.

We say that two queries are *similar* if they share some of their sources and/or they have similar filtering predicates. Formally:

Definition 3.2. ***Similarity***. *Two publication queries q_1, q_2 are similar if:*

- the intersection between their source sets is not empty $(sources(q_1) \cap sources(q_2) \neq \varnothing)$ [4],
  and/or
- at least one selection predicate of one query *subsumes* a selection predicate of the other one [5] $(\exists\, pred_i \in q_1, pred_j \in q_2 \mid pred_i \models pred_j \vee pred_j \models pred_i)$ [6].

For instance, let pub_1, pub_2 be two publication queries with the following algebraic expressions:

4. We denote by $sources(q)$ the set of sources used to define the query q.
5. The *Predicate Subsumption* algorithm is fully detailed in Listing 3.3.
6. We say that $pred \in q$ if query q contains a selection operator with predicate $pred$.

- $pub_1 = \sigma_{pred_3}(\sigma_{pred_1}(src_1 \cup src_2) \cup \sigma_{pred_2}(src_3 \cup src_4))$
- $pub_2 = \sigma_{pred_2 \wedge pred_4}(src_1 \cup src_3 \cup src_5)$

As we can see in Figure 3.2, both publications pub_1 and pub_2 are defined on sources src_1 and src_3, and both of them have similar filtering predicates defined on these sources ($pred_2$ and $pred_2 \wedge pred_4$). So, we can "merge" them somehow in order to reduce computation costs and memory usage. The problem really arises when we have a large query set and we want to merge all similar publications. On the other hand, filtering conditions can appear anywhere in the query plans, which makes the matching problem still harder. Thus, we have defined a *normal form* for queries in order to facilitate matching research between queries.

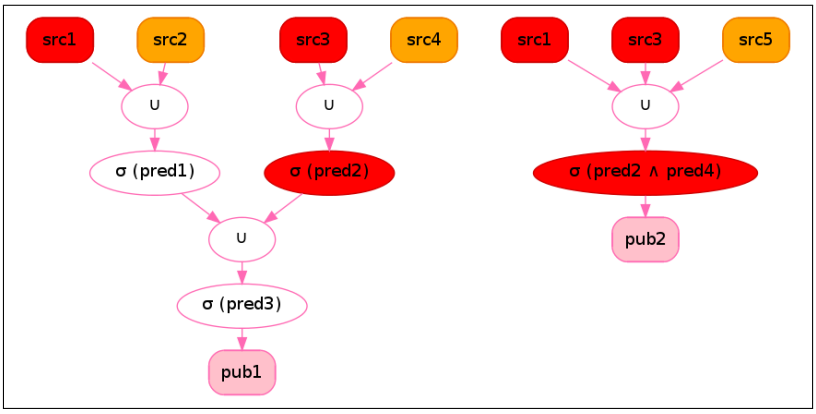


Figure 3.2 – Example of two *similar* publications

We separate the optimization problem into two subproblems: i) optimization of selection operators and ii) optimization of join and window operators. As mentioned before, our optimization strategy is based on the classical heuristics that selections and joins should be applied as early as possible in order to reduce the global cost of the query graph. Nevertheless, this work does not address the problem of the join optimization.

The main novelty of our framework with respect to other multi-query optimization solutions lies in the explicit integration of a cost model for continuous queries. This makes it more expressive than other approaches without cost model.

The *ROSES* cost model (Table 3.2) 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 conditions, and (ii) factorize common operators among publications.

The optimization process can be decomposed into two main phases:

1. a *normalization* phase which applies rewriting rules for pushing all filtering operators towards their source feeds and for distributing joins over union, and

2. a *factorization* phase where we find for each source an optimal filter plan based on a new cost-based factorization technique.

As we can see in Figure 3.2, the items of a source can be filtered by several similar selection operators. We search to factorize these operators in order to get a selection operator tree of minimal cost. This must be done for every source appearing in the *query graph*, therefore the result produced by this factorization operation will be a forest of trees containing exactly one tree for each source of the graph and where the root of each tree is the source itself and its children are selection operators.

Listing 3.1 gives a general overview of the whole optimization process. Given a query set Q, we first normalize all queries and construct a *Global Normal Query Graph* from the normalized queries. Then for each source feed s_i, we get all selection predicates $Pred_i$ depending on s_i and generate its *subsumption graph* $SG(s_i)$. This *subsumption graph* contains all filtering predicates in $Pred_i$ as well as all their subsuming predicates and the subsumption arcs among them. All arcs in the graph $SG(s_i)$ are annotated according to our Cost Model (each arc from a vertex v_1 to a vertex v_2 is labeled by the cost of vertex v_2) and finally we look for a minimal *Steiner tree* on the subsumption graph. A minimal Steiner tree is a *minimum tree* spanning at least a given subset of vertices called *terminal vertices* that correspond to the initial set of predicates $Pred_i$.

Algorithm 3.1 Optimization Algorithm Overview

Input: a set of queries Q
Output: an optimal query graph G^*

```
1: Q^N ← normalize all publication queries
2: G^N ← build a Global Normal Query Graph
3: for all source s_i ∈ G^N do
4:     SG_i ← generate a subsumption graph for the selection predicates of s_i
5:     populate SG_i with the corresponding processing costs
6:     T_i ← seek a minimal Steiner tree on the subsumption graph
7:     insert T_i in G* for source s_i
8: end for
```

We will describe the *normalization* process in the following Section (3.3), then the *factorization* process and its different algorithms in Section 3.4. Experimental results are presented in Section 3.6. Finally, Section 3.5 concludes this chapter with a description of our *dynamic* multi-query optimization strategy performed at runtime.

3.3 Query normalization

The first problem that we have when we try to find a good matching among several publication queries is that the number of possible execution plans per query is too large to try to explore all the matching possibilities among all the possible query plans for each query. This problem can be simplified through the use of normal forms, actually transforming the queries to a unique normal form enables to reduce the search space for the matching process. Thus, query normalization addresses the problem of the complexity of factorizing two or more arbitrary query plans. Normalization corresponds to the first step of the optimization process allowing to compare and match the filtering conditions of two and more queries.

Our normalization process is built upon the operator properties from the *extended* relational algebra [DGK82]. We have seen in Section 2.2.2 that these properties hold in the operators of the *ROSES* Algebra due to the snapshot-reducibility of our operators. We use the following *Rewriting Rules* for algebraic expressions:

- distributivity of selection operators over unions,
- flattening cascading selections into a single selection,
- join distributivity over union,
- commutativity between join and selection, and
- publication (*view*) decomposition.

We will give a formal definition of these five rules later in Section 3.3.1. The aim of the normalization process is to use these rules in order to (a) push selection and join operators towards the query sources, and (b) decompose the publications. This is possible by iteratively applying aforementioned rewriting rules.

It is possible to show that under *snapshot semantics* and by applying these rewriting rules we can obtain an equivalent query plan which is a four level graph, where:

- the first level (leaves) of the graph are the source feeds involved in the query,
- the second level nodes involve the filtering operators which have to be applied to each source feed (leaf),
- 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 and/or windowed-joins to build the final results.

Normalization also flattens all cascading filtering paths into elementary filtering predicates under a *Conjunctive Normal Form* (CNF).

In the following sections we will formally define the normalization process: we first introduce

a formal logical model for general queries (*i.e.*, in any form), as well as a logical model for normalized queries. These models are presented in the next Section (3.3.1). Then, Section 3.3.2 introduces the concept of *Global Normalized Query Graph*.

3.3.1 Query logical model and query normalization

Let $\mathbb{N}^S$ be a set of *Source Names* and $\mathbb{N}^P$ a set of *Publication Names* ($\mathbb{N} = \mathbb{N}^S \cup \mathbb{N}^P$). We define a *ROSES* query q as a triple (U, S, J), where:

1. U (as *Union*) is a set of *streams* $\{s_i\}$, and a *stream* s_i is:
   - either a source name $n^S \in \mathbb{N}^S$,
   - or a publication name $n^P \in \mathbb{N}^P$,
   - or a *subquery* $q' = (U', S', J')$,
2. S (as *Selection*) is a set of *predicates* $\{pred_i\}$, and $pred_i$ is a selection predicate, and
3. J (as *Join*) is a set of *join* operations $\{j_i\}$, and a join operation j_i is a 4-tuple $(\tilde{U}, \tilde{S}, pred^W, pred^J)$, where:
   1. $\tilde{U}$ is a set of *joinless* streams $\{\tilde{s}_i\}$, and a *joinless* stream $\tilde{s}_i$ is:
      - either a source name $n^S \in \mathbb{N}^S$,
      - or a publication name $n^P \in \mathbb{N}^P$,
      - or a *joinless subquery* $\tilde{q} = (\tilde{U}', S')$,
   2. $\tilde{S}$ is again a set of selection operations $\{\tilde{pred}_i\}$,
   3. $pred^W$ is a windowing predicate applied on $\tilde{U}$, and
   4. $pred^J$ is a join predicate between fields U and $\tilde{U}$.

Observe that with this model, window operators may only be applied on *secondary* streams ($\tilde{U}$) and never on a *primary* stream (U) of a query. This constraint is due to the fact that *ROSES* joins are always defined between a stream and a window (see Section 2.2.2). Note also that this logical query model ensures that all constraints enforced by the syntax of the *ROSES* Query Language in Section 2.1 are satisfied.

Example: Suppose two publication queries, one containing only union and selection operators, and a second one composed only by union and join/window operators:

```
CREATE FEED Publication1
FROM ((source1 | source2) AS $var1 | (source3 | source4) AS $var2) AS $var3
WHERE $var1[pred1] AND $var2[pred2] AND $var3[pred3];
```

```
CREATE FEED Publication2
FROM (source1 | source2) AS $var1
JOIN pred_1^W ON (source3 | source4)
     WITH $var1[pred_1^J]
JOIN pred_2^W ON (source5 | source6)
     WITH $var1[pred_2^J];
```

Figure 3.3 shows a graphical representation of a '*syntactic*' query plan for both publication queries.

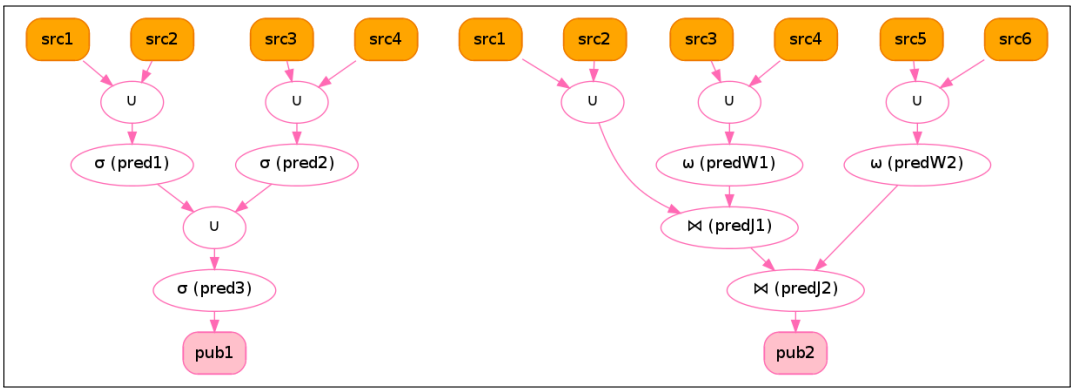


Figure 3.3 – '*Syntactic*' query plans for publications pub_1 and pub_2

The logical representation of these queries according to our query model shall be the following:

$$q_1 = \left(\left\{ \left(\{src_1, src_2\}, \{pred_1\}, \varnothing \right), \left(\{src_3, src_4\}, \{pred_2\}, \varnothing \right) \right\}, \{pred_3\}, \varnothing \right)$$

$$q_2 = \left(\{src_1, src_2\}, \varnothing, \left\{ \left(\{src_3, src_4\}, \varnothing, pred_1^W, pred_1^J \right), \left(\{src_5, src_6\}, \varnothing, pred_2^W, pred_2^J \right) \right\} \right)$$

The query normal form

A *ROSES* query $q = (U, S, J)$ is in normal form *iff*:

1. U is a set of *subqueries* $\{q'_i\}$, and for each *subquery* $q'_i = (U', S', J')$:
    1. U' is a singleton set containing exactly one element, either a source name $n^S \in \mathbb{N}^S$, or a publication name $n^P \in \mathbb{N}^P$, *i.e.* $U' = \{n\}$ where $n \in \mathbb{N} = \mathbb{N}^S \cup \mathbb{N}^P$,

2. S' is also a singleton set containing a selection predicate in *CNF*, *i.e.* $S' = \{pred_{CNF}\}$ [7], and
3. J' is a set of *join* operations $\{j_i\}$, and for each join operation $j_i = (\tilde{U}, S, pred^W, pred^J)$:
    1. $\tilde{U}$ is a set of *joinless subqueries* $\{\tilde{q}_j\}$, and for each *joinless subquery* $\tilde{q}_j = (\tilde{U}', S')$:
        1. $\tilde{U}'$ is a singleton set containing either a source name or a publication name: $\tilde{U}' = \{n'\}$, and
        2. S' is also a singleton set containing a selection predicate in *CNF*, $S' = \{pred'_{CNF}\}$;
    2. S is an empty set ($S = \varnothing$), since selection operations are not allowed after unions,
    3. $pred^W$ the windowing predicate, and
    4. $pred^J$ the join predicate;
2. S is an empty set ($S = \varnothing$), since selection operations are not allowed after unions, and
3. J is also an empty set ($J = \varnothing$), since join operations are not allowed after unions.

Nevertheless, the properties of the normal form allow a simpler modeling of *ROSES* normalized queries. Thus we can also describe a normalized publication query as a set of triples $\{(n, pred_{CNF}, J)\}$, where:

1. n is either a *Source Name*, or a *Publication Name*: $n \in \mathbb{N} = \mathbb{N}^S \cup \mathbb{N}^P$,
2. $pred_{CNF}$ is a selection predicate in *Conjunctive Normal Form*, and
3. J is a set of joins $\{j_i\}$, where j_i is now a triple $(U, pred^W, pred^J)$ and:
    1. U is a set of pairs $\{(n', pred'_{CNF})\}$,
    2. $pred^W$ is a windowing predicate and
    3. $pred^J$ is the join predicate between n and U

The following five *rewriting rules* are sufficient for transforming any publication query into the normalized form. The first four rules (3.1, 3.2, 3.3, 3.4) allow to normalize the query itself (*i.e.* the composition of its operators), while the last rule (3.5) enables normalization of the selection predicates appearing in the query:

7. We may have no selection operator associated to that source/publication, in this case we consider $pred_{CNF} = true$

Selection distributivity over union

$$\sigma_{pred}(S_1 \cup S_2) \mapsto \sigma_{pred}(S_1) \cup \sigma_{pred}(S_2) \tag{3.1}$$

This rule enables to push selection operators towards the sources through union operators in both cases, in *main* and *secondary* streams.

Selection-join commutativity

$$\sigma_{pred}(S \bowtie_{pred^J} W) \mapsto \sigma_{pred}(S) \bowtie_{pred^J} W \tag{3.2}$$

Observe that commutativity of selection 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 (*i.e.*, filtering operations cannot be applied on the annotations generated by join operators).

Join distributivity over union

$$(S_1 \cup S_2) \bowtie_{pred^J} W \mapsto (S_1 \bowtie_{pred^J} W) \cup (S_2 \bowtie_{pred^J} W) \tag{3.3}$$

With this rule we push join operators towards the sources through union operators splitting the join operator over each stream.

Selection cascading

$$\sigma_{pred_1}(\sigma_{pred_2}(S)) \mapsto \sigma_{pred_1 \wedge pred_2}(S) \tag{3.4}$$

This rule enables to aggregate all consecutive selection operators into a single selection.

Normalization of selection predicates

$$\sigma_{\neg\neg A}(S) \mapsto \sigma_A(S) \qquad \text{double negative law} \tag{3.5a}$$

$$\sigma_{\neg(A \wedge B)}(S) \mapsto \sigma_{\neg A \vee \neg B}(S) \qquad \textit{De Morgan}\text{'s laws} \tag{3.5b}$$

$$\sigma_{\neg(A \vee B)}(S) \mapsto \sigma_{\neg A \wedge \neg B}(S) \tag{3.5c}$$

$$\sigma_{A \vee (B \wedge C)}(S) \mapsto \sigma_{(A \vee B) \wedge (A \vee C)}(S) \qquad \text{distributive laws} \tag{3.5d}$$

$$\sigma_{(A \wedge B) \vee (A \wedge C)}(S) \mapsto \sigma_{A \wedge (B \vee C)}(S) \tag{3.5e}$$

Finally, the equations 3.5a-3.5e enable to normalize all selection predicates appearing in the query into the *Conjunctive Normal Form*.

Iteratively applying these rewriting rules (3.1-3.5) till fixed point allows normalizing all *ROSES* queries. So, if we apply query normalization to the two preceding queries, we get these *Normal Forms* for pub_1 and pub_2:

$$NF(q_1) = \Big\{\Big(n_1^S, pred_1 \wedge pred_3, \varnothing\Big), \Big(n_2^S, pred_1 \wedge pred_3, \varnothing\Big), \Big(n_3^S, pred_2 \wedge pred_3, \varnothing\Big), \Big(n_4^S, pred_2 \wedge pred_3, \varnothing\Big)\Big\}$$

$$NF(q_2) = \Big\{\Big(n_1^S, true, \{j_1, j_2\}\Big), \Big(n_2^S, true, \{j_1, j_2\}\Big)\Big\}$$
$$\text{where: } j_1 = \Big(\big\{(n_3^S, true), (n_4^S, true)\big\}, pred_1^W, pred_1^J\Big)$$
$$j_2 = \Big(\big\{(n_5^S, true), (n_6^S, true)\big\}, pred_2^W, pred_2^J\Big)$$

A graphical representation of these normal forms is shown in Figure 3.4.

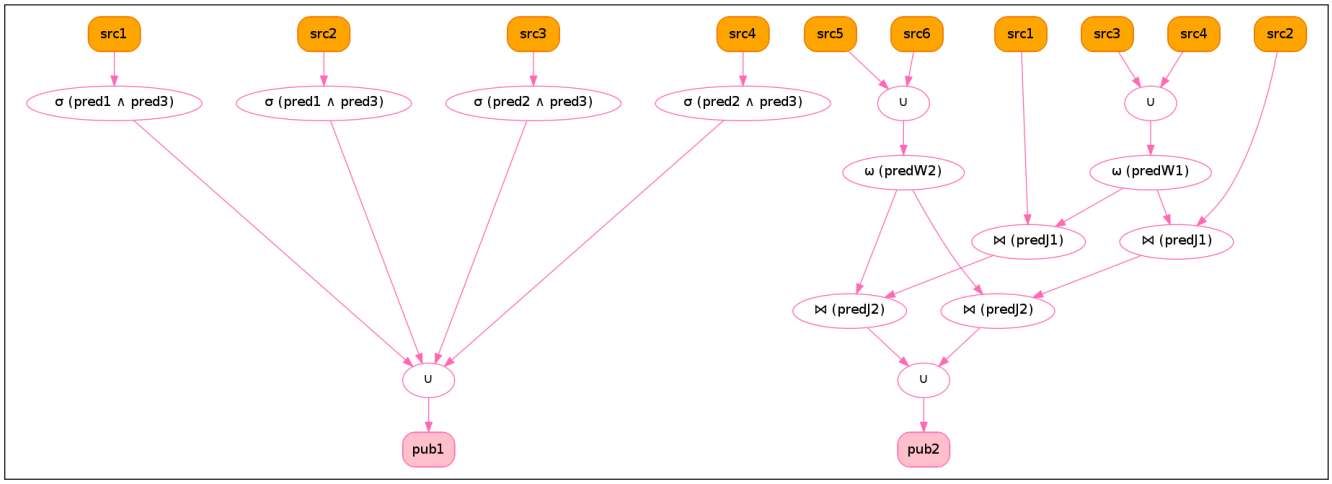


Figure 3.4 – Graphical representation of the *Normal Form* of queries pub_1 and pub_2

3.3.2 Global normal query graph

We create the *Normal Graph of Queries* G^N by merging all normalized queries. If we define a *ROSES* publication p as a pair (n^P, q) (*i.e.*, a named query), given a set of publication queries P we can define a normal query graph G^N as a set of quadruplets $\{(n^S, pred_{CNF}, J, n^P)\}$, where

n^P is the name of the publication. More formally:

$$G^N(P) = \{(n^S, pred_{CNF}, J, n^P) \mid p \in P \wedge p = (n^P, q) \wedge (n^S, pred_{CNF}, J) \in NF(q)\} \quad (3.6)$$

We illustrate our approach by a simple example without join operations. Assume we have the following three publications:

- $pub_1 = \sigma_{a \wedge b}(src_1 \cup src_2)$
- $pub_2 = \sigma_c(pub_1 \cup src_3)$
- $pub_3 = \sigma_d(\sigma_{b \wedge c}(src_2 \cup src_3) \cup src_4)$

A graphical representation of the corresponding query plans is given in Figure 3.5. We can see that src_2 is used by all three publications: src_2 is used directly by queries pub_1 and pub_3 and indirectly by publication pub_2 (through query composition). In this case (without join), the normalization process consists only in pushing all filtering operations through the publication tree to the sources for obtaining a normalized query plan as shown in Figure 3.6.

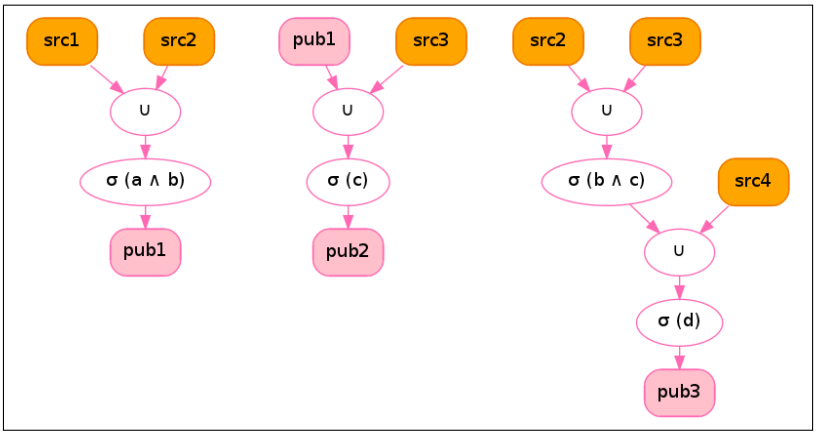


Figure 3.5 – Three publications using all of them source src_2

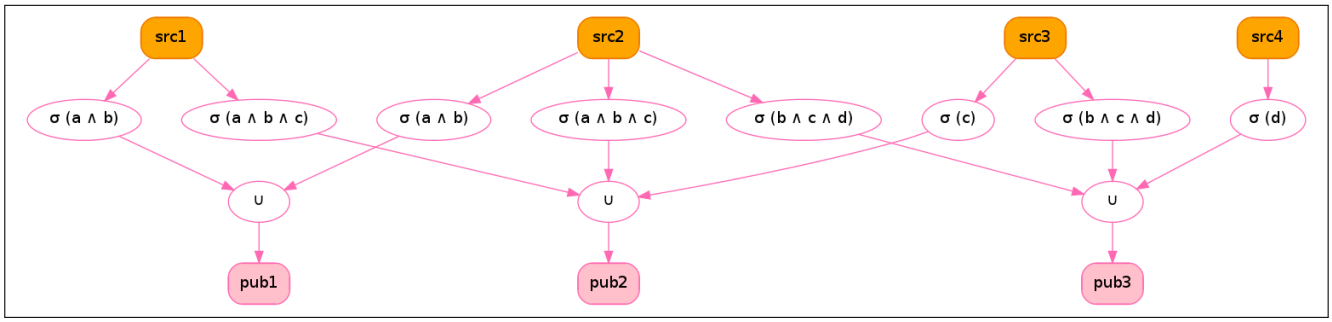


Figure 3.6 – Global Normal Query Graph for the query set $\{pub_1, pub_2, pub_3\}$

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 later, this cost increase is temporary and will be amortized by the subsequent factorization phase. Our algorithm guarantees to produce optimized plans with equal or lower cost than the original plan.

3.4 Factorization algorithms

In this Section we tackle the last step of the optimization process, *i.e.*, the query factorization. First we introduce a few preliminary concepts used during the factorization phase: the subsumption graphs and a simplified version of the cost model (Section 3.4.1). Afterwards we present in Section 3.4.2 our three factorization algorithms: *STA*, *VCA* with subsumption graph and *VCA* with *VCB*.

3.4.1 Query factorization

In this thesis we focus on *filtering factorization*. The most common type of queries defined by the users in our context involve filtering a large collection of sources, thus filtering factorization becomes a really effective optimization technique. As in oneshot queries, first filtering leads to optimal solutions, and in our context the probability to find similar user-defined filters is higher than the one for similar join operations. [8]

The normalization process generates a global query plan G^N (obtained through Formula 3.6), where each source src_i is filtered by a set of selection predicates $Pred(src_i)$. The factorization considers each source separately and consists in building for each such set of predicates $Pred(src_i)$ an optimized *filtering plan* with respect to our cost model. To find the best operator factorization we proceed in two steps: we first generate for each source feed src_i a *Predicate Subsumption Graph* $G^S(src_i)$ which contains *all* predicates subsuming the set of predicates in $Pred(src_i)$. 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 src_i (root) and *all predicates* in $Pred(src_i)$ corresponds to a filtering plan that is equivalent to the original one. The cost of this plan is the sum of the costs of the tree edges.

The second step consists in finding a *Steiner Minimal-Cost Tree* (SMT) rooted at src_i which covers $Pred(src_i)$, given a directed edge-weighted graph $G = (V, E, w)$, a root $src_i \in V$, and a subset $T \subseteq V$ of required vertices (corresponding to $Pred(src_i)$). A *Steiner Tree* is a tree t in G

8. We do not tackle the join optimization issue in this work.

with src_i as the root and that spans all vertices of T. The optimization problem associated with Steiner trees is to find a minimal-cost Steiner tree. Observe that if all nodes in the graph are required (*i.e.* $T = V$), a minimal-cost Steiner tree corresponds to a minimum-weight spanning tree of G.

The Predicate Subsumption Graph construction may be done using the following approach:

1. In a first step, we recursively generate all *sub-predicates* for each selection predicate appearing in $Pred(src_i)$. Given a predicate $pred_i^j$ in $Pred(src_i)$, the set of *sub-predicates* of $pred_i^j$ might be defined as the *power set* on the clauses that $pred_i^j$ consists of. For instance, if $pred_i^j = (a \vee b) \wedge c \wedge (d \vee e)$, the sub-predicates would be: $(a \vee b) \wedge c$, $(a \vee b) \wedge (d \vee e)$, $(a \vee b)$, $c \wedge (d \vee e)$, etc. This decomposition must be done recursively till the literal/disjunction level. Given that, after the normalization process, all predicates are in conjunctive normal form, sub-predicate generation may be just done by computing a power set on the *and* operation. We give in Listing 3.2 an efficient algorithm to compute all possible sub-predicates from a *conjunctive selection predicate*. This algorithm offers the best possible order of complexity ($O(2^n)$), since we need to generate 2^n possible combinations of predicates (where n is the number of clauses in the conjunctive predicate). We compute all possible *sub-predicates* for each predicate $pred_i^j$ appearing in $Pred(src_i)$ and a new node is created for every sub-predicate. So, the total cost of this step corresponds to $O(2^n \cdot m)$, where m is the number of predicates in $Pred(src_i)$.

2. Once all nodes created, the second step consists in computing and adding all the *subsumption relationships* among all these nodes. Listing 3.3 gives the *Subsumption* algorithm for all different predicates in CNF. It provides a complete list of syntactic rules for producing all subsumption relationships between any pair of predicates in CNF (if it exists): atoms, negation of atoms, disjunction and conjunction.

Note that in Section 3.4.2.3, we will present a data structure which avoids the computation of the whole Predicate Subsumption Graph.

Finally, the Subsumption Graph [9] is weighted according to our cost model. However, as long as our optimization approach consists in reducing the input/output rate between the operators and we do not optimize the operators themselves but the structure of the operator plan, that cost model may be simplified. Indeed, we are interested in optimizing a tree of selection operators. In this case, read and write costs may be neglected if we compare them to the evaluation cost of the filtering operators, thus the *processing* (total) *cost* of a filtering tree is proportional to its *evaluation cost*. This leads us to the *Simplified Cost Model* we summarize

9. We use *Subsumption Graph* as a short name for *Predicate Subsumption Graph* in the rest of the document.

Algorithm 3.2 PowerSet: Sub-predicate set computation

Input: a conjunctive selection predicate $originalPredicate$
Output: the set of all sub-predicates of $originalPredicate$

```
1: sets ← ∅
2: if originalPredicate = ∅ then
3:     return sets
4: end if
5: list ← listOfClauses(originalPredicate)
6: head ← list[0]
7: rest ← toSet(subList(list, 1, size(list)))
8: for all pred ∈ PowerSet(rest) do
9:     newPred ← head ∧ pred
10:    sets ← sets ∪ newPred ∪ pred
11: end for
12: return sets
```

in Table 3.3. Thus, Subsumption Graph's arcs are weighted according to the corresponding source rate and the predicate selectivity of the destination node of the arc.

Operator op	Output rate $rate(op)$	Memory cost $cost^{Mem}(op)$	Processing cost $cost^{Proc}(op)$
$\sigma_{pred}(ioq)$	$selectivity(pred) \cdot rate(ioq)$	$const$	$cost^{Eval}(pred) \cdot rate(ioq)$
$ioq_1 \cup \cdots \cup ioq_n$	$\sum_{i=1}^{n} rate(ioq_i)$	0	0
$\omega_w^{time}(ioq)$	0	$w \cdot rate(ioq)$	$rate(ioq)$
$\omega_N^{count}(ioq)$	0	N	0
$ioq \bowtie_{pred^J} win$	$selectivity(pred^J) \cdot rate(ioq)$	$const$	$rate(ioq) \cdot size(win)$

Tableau 3.3 – Simplified Cost Model: memory and processing costs

Example. Figure 3.7 illustrates the subsumption graph for the source feed src_2 of the example used in previous Section (3.3.2). The filtering operators involving src_2 were: $\sigma_{a \wedge b}$, $\sigma_{a \wedge b \wedge c}$ and $\sigma_{b \wedge c \wedge d}$ (see Figure 3.6). We include one more operator $\sigma_{d \wedge e}$ in order to improve the example. Thus, the subsumption graph is composed of these four *required* predicates (shown in blue), as well as of all their sub-predicates: a, b, c, d, e, $a \wedge c$, $b \wedge c$, $b \wedge d$ and $c \wedge d$ (in yellow), while the edges express subsumption between predicates: $a \rightarrow a \wedge b$, $b \rightarrow a \wedge b$, $b \rightarrow b \wedge c \wedge d$, etc. Note that *direct* subsumption arcs are represented in Figure 3.7 by straight black lines, in fact the graph also includes arcs from the transitive closure of the *subsumption relation*, *e.g.*, $b \rightarrow b \wedge c \wedge d$. These last arcs are depicted by dashed gray lines. The arc weight represents

Algorithm 3.3 Subsumption of Predicates

All predicates are in CNF, i.e., either an atom, or the negation of an atom, or a disjunction/conjunction of literals, or yet a conjunction of disjunctions of literals.

In the following A and B denote atomic predicates, L_i denotes a literal (an atom or a negated atom), and D_i denotes a disjunction of literals.

Atoms:

A subsumes $B \Leftrightarrow A = B$
A subsumes $\neg B \Leftrightarrow$ **false**
A subsumes $(L_1 \vee \cdots \vee L_n) \Leftrightarrow \forall L_i : A$ subsumes L_i
A subsumes $(D_1 \wedge \cdots \wedge D_n) \Leftrightarrow \exists D_i : A$ subsumes D_i

Negation:

$\neg A$ subsumes $B \Leftrightarrow$ **false**
$\neg A$ subsumes $\neg B \Leftrightarrow A = B$
$\neg A$ subsumes $(L_1 \vee \cdots \vee L_n) \Leftrightarrow \forall L_i : \neg A$ subsumes L_i
$\neg A$ subsumes $(D_1 \wedge \cdots \wedge D_n) \Leftrightarrow \exists D_i : \neg A$ subsumes D_i

Disjunction:

$(L_1 \vee \cdots \vee L_n)$ subsumes $A \Leftrightarrow \exists L_i : L_i$ subsumes A
$(L_1 \vee \cdots \vee L_n)$ subsumes $\neg A \Leftrightarrow \exists L_i : L_i$ subsumes $\neg A$
$(L_1 \vee \cdots \vee L_n)$ subsumes $(L'_1 \vee \cdots \vee L'_m) \Leftrightarrow \forall L_i : L_i$ subsumes $(L'_1 \vee \cdots \vee L'_m)$
or $\forall L_i : \forall L'_j : L_i$ subsumes L'_j
$(L_1 \vee \cdots \vee L_n)$ subsumes $(D_1 \wedge \cdots \wedge D_m) \Leftrightarrow \forall L_i : L_i$ subsumes $(D_1 \wedge \cdots \wedge D_m)$
or $\forall L_i : \exists D_j : L_i$ subsumes D_j

Conjunction:

$(D_1 \wedge \cdots \wedge D_n)$ subsumes $A \Leftrightarrow \forall D_i : D_i$ subsumes A
$(D_1 \wedge \cdots \wedge D_n)$ subsumes $\neg A \Leftrightarrow \forall D_i : D_i$ subsumes $\neg A$
$(D_1 \wedge \cdots \wedge D_n)$ subsumes $(L_1 \vee \cdots \vee L_m) \Leftrightarrow \forall D_i : D_i$ subsumes $(L_1 \vee \cdots \vee L_m)$
or $\forall D_i : \forall L_j : D_i$ subsumes L_j
$(D_1 \wedge \cdots \wedge D_n)$ subsumes $(D'_1 \wedge \cdots \wedge D'_m) \Leftrightarrow \forall D_i : D_i$ subsumes $(D'_1 \wedge \cdots \wedge D'_m)$
or $\forall D_i : \exists D'_j : D_i$ subsumes D'_j

the *selectivity* of the originating predicate, proportional to the rate of items coming from the source feed ($\top$).

In our example, the rate of items produced by src_2 is equal to 1 and the selectivity for each atomic predicate is given in Table 3.4. Thus the weight of arc $(a, a \wedge b)$ is $selectivity(a) \cdot rate(src_2) = 0.2$ and the weight of $(a \wedge b, a \wedge b \wedge c)$ is equal to $selectivity(a) \cdot selectivity(b) \cdot rate(src_2) = 0.2 \cdot 0.1 \cdot 1$.

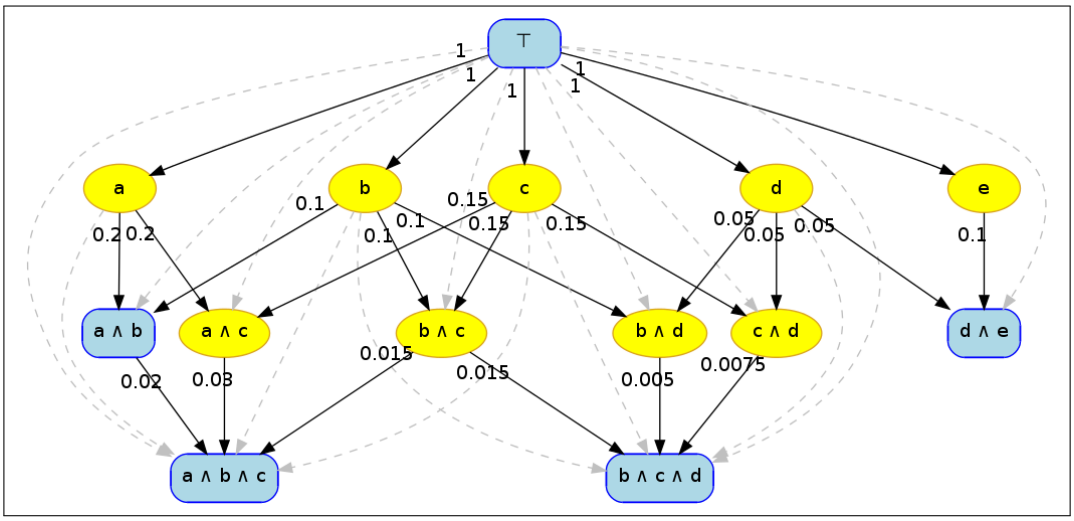


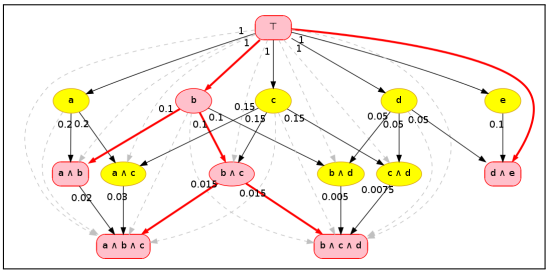
Figure 3.7 – *Predicate Subsumption Graph* for src_2 with its weights

Predicate	a	b	c	d	e
Selectivity	0.2	0.1	0.15	0.05	0.1

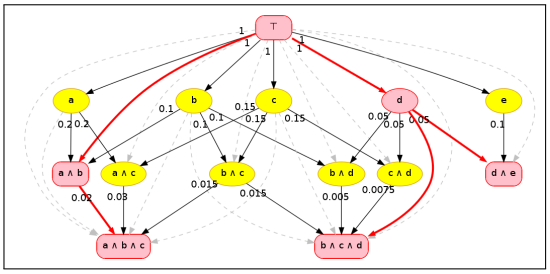
Tableau 3.4 – Atomic predicate selectivities for source feed src_2

As we can see in Figure 3.8, several Steiner trees (in red) might be computed for each subsumption graph, each one with a different cost. In our example, the second tree is the minimal one with an estimated cost of 2.12.

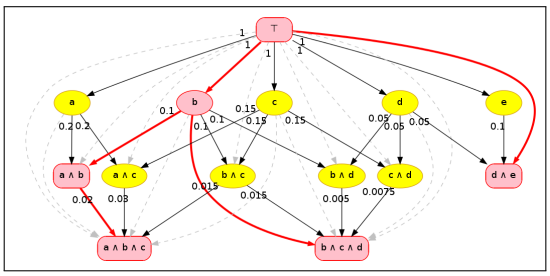
We can easily observe that the resulting filtering plan is less costly than the initial one. And this is true despite the fact that normalization can increase the cost of the filtering plan since it replaces cascading selection paths by a conjunction of all predicates on the path. As a matter of fact, 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 src_2 in the original plan (Figure 3.5) roughly is three times the publishing rate of src_2 (if we suppose that selection $\sigma_{d \wedge e}$ is directly defined over src_2,



(a) Tree cost = 2.23



(b) Tree cost = 2.12



(c) Tree cost = 2.22

Figure 3.8 – Different *Steiner* trees with different tree costs

all three selections $\sigma_{a \wedge b}$, $\sigma_{b \wedge c}$ and $\sigma_{d \wedge e}$ are applied to all items produced by src_2). This cost is reduced by a 2/3 factor in the final Steiner tree (Figure 3.8b) by introducing the additional filter σ_d.

3.4.2 The factorization algorithms

This Section describes our *factorization algorithms*. First we present *STA*, an adaption of an existing approximate algorithm due to Charikar *et al.* [CCC+98]. STA performs a near exhaustive search on the subsumption graphs, which makes this algorithm to provide optimal solutions, but at the expense of a lengthy computation time (see Section 3.4.2.1). Then we present in Section 3.4.2.2 the *VCA* predicate factorization algorithm. VCA is a greedy algorithm that takes advantage of some properties inherent to our subsumption graphs. VCA outperforms STA's computation cost, nevertheless we still require an expensive memory usage to store the subsumption graph, this is why we have developed a second version of VCA. This second version uses an optimized dynamic data structure called *VCB*, which is described in Section 3.4.2.3.

3.4.2.1 The STA algorithm

The *Steiner Minimal-cost Tree* (SMT) problem is known to be *NP-complete* [HRW92, CD02] and many *approximation* algorithms have been proposed in literature. We have initially implemented in *ROSES* a modified version of the algorithm introduced in [CCC+98] for edge-weighted directed graphs. The algorithm proposed by Charikar *et al.* in [CCC+98] reaches an approximation ratio of $i \cdot (i-1) \cdot k^{1/i}$ in time $O(n^i \cdot k^{2i})$ for any fixed $i > 1$, where k is the number of terminal nodes. Thus setting $i = \log k$, they obtain an $O(\log^2 k)$ approximation ratio in quasi-polynomial time.

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). In fact, the approximation factor of the algorithm proposed by Charikar depends on the search depth of the algorithm. If we consider *cyclic* graphs, the search depth is unbounded. However our subsumption graphs are *acyclic*, thus our variant of the algorithm can apply infinite search depth and produce an optimal result.

Despite this simplification, our Algorithm 3.4 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 our *experimental evaluation* in Section 3.6).

In Algorithm 3.4, $Graph$ is a directed labeled graph where each node $n \in Nodes(Graph)$ is labeled by a filtering predicate $\lambda(n)$ and there exists an arc $(a, b) \in Arcs(Graph)$ 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 $pred$ subsuming $\lambda(a)$, either $pred \equiv \lambda(a)$ or there exists an ancestor b of a in $Graph$ such that $\lambda(b) \equiv pred$ (for the root $root$ of $Graph$: $\lambda(root) \equiv true$).

The labeling function $\omega_S : Arcs(Graph) \rightarrow \Re$ returns for each edge $(a, b) \in Arcs(Graph)$ 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 3.4 is to recursively iterate over all successors for each vertex in the graph in order to find an 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 covered by T (*tree density*). Remind that the set of terminal nodes $Term$ corresponds to the set of initial selection predicates applied on the source S.

Algorithm 3.4 STA

Input: a directed acyclic labeled weighted graph $Graph$, a current tree root $root$, a set of $terminal$ nodes $Term \subseteq Nodes(Graph)$, and the minimum number of terminals k that must be satisfied

Output: A tree $Tree$ rooted at $root$ that spans at least k terminals in $Terminals$

1: $Tree \leftarrow \varnothing$
2: $k \leftarrow |Term|$ // size of $Term$
3: **while** $(k > 0)$ **do**
4: // $Tree$ does not cover all terminals
5: $Tree_{best} \leftarrow \varnothing$
6: $density(Tree_{best}) \leftarrow \infty$
7: **for all** $vertex \in Nodes(Graph)$ **do**
8: **for all** $k', 1 \leq k' \leq k$ **do**
9: $Tree' \leftarrow STA(Graph, vertex, Term, k')$
10: $Tree' \leftarrow Tree' \cup \{(root, vertex)\}$ // add root $root$
11: $density(Tree') \leftarrow \omega_S(Tree')/|Nodes(T') \cap Term|$
12: **if** $(density(Tree_{best}) > density(Tree'))$ **then**
13: $Tree_{best} \leftarrow Tree'$
14: **end if**
15: **end for**
16: **end for**
17: $Tree \leftarrow Tree \cup Tree_{best}$
18: $Term \leftarrow Term - Nodes(Tree_{best})$
19: $k \leftarrow |Term|$
20: **end while**
21: **return** $Tree$

3.4.2.2 The VCA predicate factorization algorithm

The need to exhaustively search for minimal Steiner subtrees makes previous algorithm unviable on large subsumption graphs. For this reason we have devised a new approximate algorithm for minimal-cost Steiner Trees, called VCA (*Very Clever Algorithm*), see Algorithms 3.5 and 3.6. This algorithm takes into account the peculiarities of predicate subsumption graphs and the corresponding cost-model in *ROSES*. VCA is a *greedy* algorithm, iteratively improving an existing filter plan according to a local heuristic which estimates the benefit of adding new intermediate predicates. VCA starts from an initially correct plan where the root (predicate *true*) is directly connected to each predicate in the set *Preds* of target predicates, *e.g.* $Preds = \{a \wedge b, a \wedge b \wedge c, b \wedge c \wedge d, d \wedge e\}$ in Figure 3.7. This corresponds to a plan where all filtering predicates are evaluated independently on the source feed and whose *cost* is proportional to the number of predicates (the selectivity of root *true* multiplied by the number of filter predicates).

We note *Border* the set of children of the root in the current plan, *i.e.* $Border = Preds$ 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 3.7 $a \wedge b$ belongs at the beginning to both *Border* and *Cands* (it subsumes border nodes $a \wedge b$, itself, and $a \wedge b \wedge c$).

Based on these two sets, VCA iteratively tries to replace with some node in *Cands* all the nodes it subsumes in current *Border*. Namely, VCA checks if adding a node $n \in Cands$ in the tree is beneficial. In such case, the algorithm replaces by n all the nodes in the *Border* that are subsumed by n. 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 *benefit* function $benefit(n_1, n_2)$ estimating the benefit of adding n_1 as a child of an existing node n_2 in the tree, where k is the number of children of n_2 which are also subsumed by n_1. Inserting n_1 means replacing arcs from n_2 to the k children with (1) an arc from n_2 to n_1 and (2) arcs from n_1 to the k children. We depict this operation in Figure 3.9, where we see that node n_2 subsumes node n_1, which in turn subsumes nodes n_3, n_5 and n_6 ($k = 3$). Thus, if the *benefit* function of adding n_1 is positive, we remove the arcs (n_2, n_3), (n_2, n_5) and (n_2, n_6), and we insert an arc between n_2 and n_1 and the arcs (n_1, n_3),

(n_1, n_5) and (n_1, n_6). We define this operation as *expansion*.

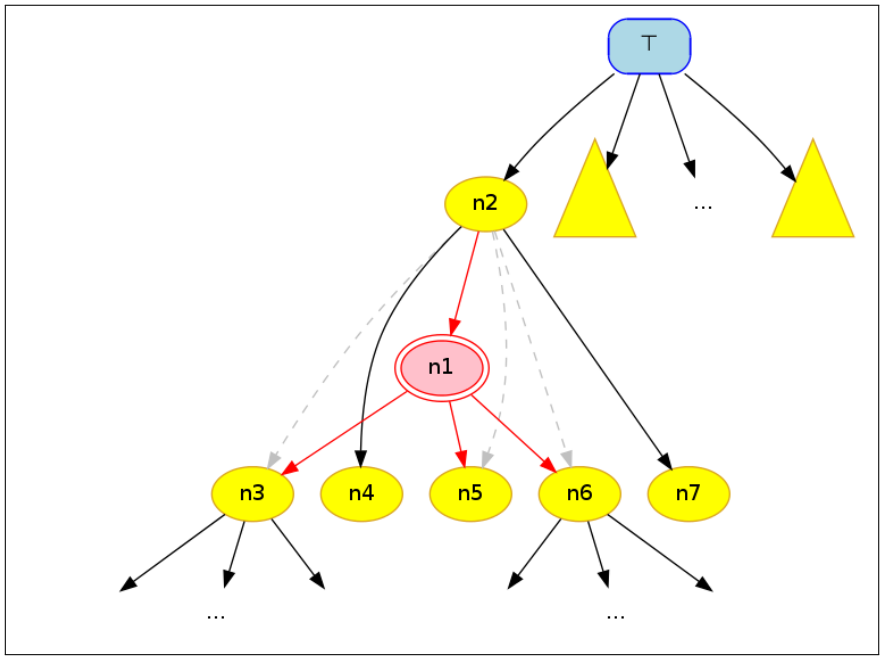


Figure 3.9 – Benefit of adding a node n_1 as a child of n_2

$$benefit(n_1, n_2) = (k-1) \cdot selectivity(n_2) - k \cdot selectivity(n_1) \tag{3.7}$$

This function expresses the difference between the costs of the filter trees before and after the insertion of n_1. A positive *benefit* means the filter tree is improved by the insertion of node n_1, a negative one means the plan may be improved by deleting n_1. This information is exploited by VCA in two ways. First, during the *expansion phase*, benefit 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 each candidate subsumes, which increases the computation cost of the algorithm. In fact, it is not necessary to compute all candidates, but only the border of the most selective ones or those with the highest benefit. We exploit this idea in the VCB data structure that we introduce in the following Section (3.4.2.3). As verified experimentally in Section 3.6, we can quickly obtain a good approximation of the minimal Steiner tree by choosing candidates only based on their selectivity.

In each step of the *expansion phase*, a candidate node replaces one or several nodes in the border with a positive benefit on the total cost. However, adding new nodes changes the context of existing ones, whose utility may be rediscussed. For instance, consider that a filter

plan contains a node n_1, that has been added with $benefit(n_1, true) > 0$. A new node n_3 is inserted as a parent of n_1. If the selectivities of n_1 and n_3 are close, it is possible that the benefit of n_3 to be the parent of n_1 becomes negative ($benefit(n_1, n_3) < 0$), *i.e.*, the plan cost improves if n_1 is discarded and n_3 is directly connected to n_1's children. This is the second use of the benefit function, during what is called the *reduction phase*. It is also noteworthy that *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 the children of n_1 have not been reduced when n_1 was added, they will also not be reduced by the n_3 node because $selectivity(n_3) > selectivity(n_1)$.

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: $\{true\}$.

3.4.2.3 Finding the best candidates with VCB

VCA reduces drastically the computation cost of the factorization process while offering similar quality to that of the general-purpose SMT algorithm of [CCC+98]. Whereas we have no theoretical 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 for different query workloads.

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 VCB (*Very Clever Border*), 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

Algorithm 3.5 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 filtering 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: $Tree \leftarrow Reduce(fact, bestCand, Tree, Preds)$
20: **end for**
21: $Border \leftarrow Children(true)$ // recompute $Border$
22: $Cands \leftarrow FindCandidates(Border)$ // recompute $Cands$
23: **else**
24: $Cands \leftarrow Cands - \{bestCand\}$ // remove $bestCand$ from $Cands$
25: **end if**
26: **end while**
27: **return** $Tree$

Algorithm 3.6 Reduce

Input: an intermediate filtering plan $Tree$, the set of filtering predicates $Preds$ and a couple of nodes $node_1, node_2 \in Tree$ that may be collapsed

Output: the filtering tree $Tree$ with $node_2$ collapsed into $node_1$ if prompted by the benefit function

1: $k \leftarrow outdegree(node_2)$
2: **if** $node_2 \notin Preds \wedge k \neq 0 \wedge benefit(node_2, node_1) < 0$ **then**
3: // Reduce : replace $node_2$ by $node_1$ as parent of all children of $node_2$
4: $Tree \leftarrow Tree - \{(node_1, node_2)\}$
5: **for all** $child \in Children(node_2)$ **do**
6: $Tree \leftarrow (Tree - \{(node_2, child)\}) \cup \{(node_1, child)\}$
7: **end for**
8: **end if**
9: **return** $Tree$

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.

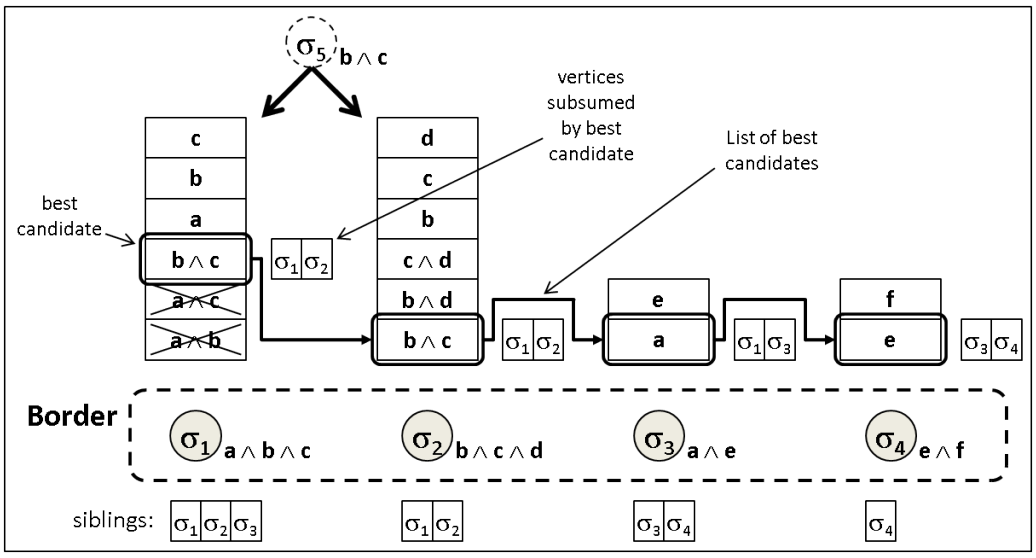


Figure 3.10 – Very Clever Border

Consider a source feed src which is associated with four filtering predicates: $a \wedge b \wedge c$, $b \wedge c \wedge d$, $a \wedge e$, $e \wedge f$ (see Figure 3.10). Initially, we generate for each filtering predicate $pred_i$ in the border the list of candidates $CandList(pred_i)$. To find the most selective candidate we traverse each list to find the first candidate subsuming at least another predicate in the border ($pred_j \neq pred_i$). This candidate will be called the best candidate for $pred_i$, denoted

$bestCand(pred_i)$. 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 the $CandList(a \wedge b \wedge c)$. Additionally, we store for each best candidate $bestCand(pred_i)$ the list of nodes in the border which are subsumed by $bestCand(pred_i)$ (σ_1 and σ_2 for best candidate of $a \wedge b \wedge c$).

Symmetrically, we also compute for each predicate $pred_i$ the list of nodes whose best candidate subsumes $pred_i$. This list is called the *sibling list* of $pred_i$. 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 $pred_i$ 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 ($b \wedge c$) 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 candidate list (σ_3). Indeed, σ_1 is not anymore in the border, so a (the best candidate of σ_3) subsumes now 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*.

3.5 Runtime optimization

The factorization algorithms we have proposed in previous Section (3.4) enable to produce optimal (STA) or near-optimal (VCA and VCB) filtering trees given a *static* set of queries. In particular, our VCA algorithm allows finding near-optimal solutions with a very good trade-off between *optimality* and optimization cost. However, this is only a partial solution for a dynamic system where publication queries are continuously created and removed by the users of the system, and where the publication behavior (term frequencies) of sources is not constant but evolving over time. For instance, a given term or word may suddenly become a *trending topic*, this makes its selectivity to vary and, thus, strongly degrade the execution cost of on running query plans.

In the first case, each time a publication query is created or deleted, the filtering trees of all sources involved in this query should be recomputed dynamically in order to maintain query graph optimality. This is unfeasible due to the cost of optimization algorithms. We do not want to reoptimize the filtering trees over and over but only when it is necessary, *i.e.*, when recomputing the filtering tree provides a substantial gain *w.r.t.* the ongoing query execution plan.

3.5.1 Runtime optimization strategy

In this Section we present our approach to handle this problem that we call *runtime optimization*.

In our context, the two main factors that may lead filtering trees to degrade are the following ones:

- *query arrival/departure*: the arrival of new publication queries as well as the suppression of ongoing queries, and
- *selectivity changes*: the evolution of selection predicate selectivities bred by changing term frequencies.

We have seen that the *publishing rate* of sources plays an important role in the *static* optimization process. In the same way that the term frequency on incoming items evolves over time, the publishing rate of the sources is not constant. with respect to their optimal plans. In some cases, behavior changes of some sources may be unpredictable. However, given that every source filtering tree is optimized independently, the dynamicity of the publishing rates do not affect the overall query graph optimality. Consequently, we do not need to take into account the source rates in the *runtime* optimization process.

The *runtime optimization* approach we propose is the following one: when new queries arrive (existing queries leave) we insert (remove) them into the current query graph in the best possible place of the existing filtering plan. We define the best possible place through the concept of the *best direct ancestor*.

Definition 3.3. *The* best direct ancestor *of a predicate* $pred_x^{new}$ *is the predicate* $pred_y^{old} \in T_i$ *that:*

1. subsumes $pred_x^{new}$ ($pred_y^{old} \models pred_x^{new}$), and
2. has the lowest selectivity factor in the filtering tree T_i.

This semi-naive strategy for query insertion obviously leads to graph degradation, thus we estimate periodically the need of reoptimizing the filtering trees through some heuristics. When this estimation exceeds a given threshold we recompute the corresponding trees.

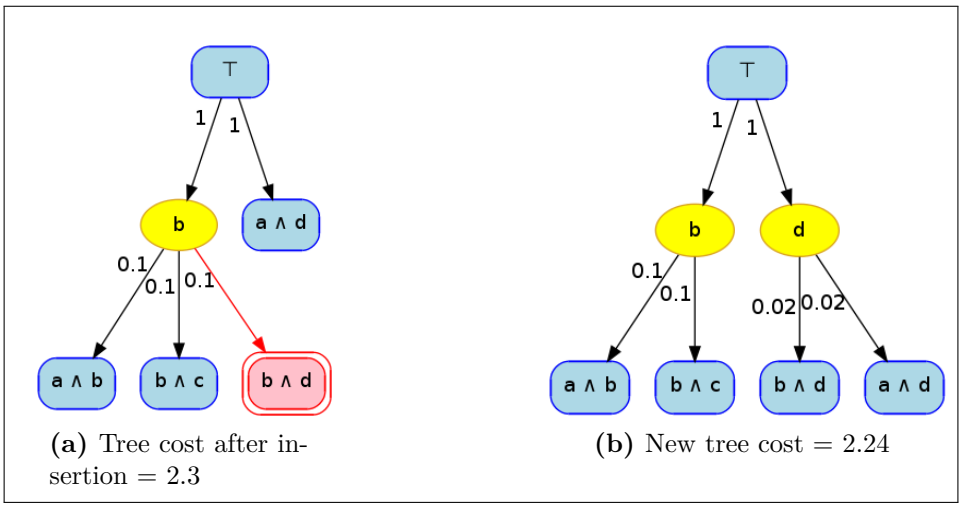
We can summarize our *runtime optimization* strategy as follows:

1. *Closest ancestor query insertion*: in a first phase, once a new query is submitted into the system, it is normalized and for each new selection predicate $pred_i$ we insert it into the corresponding filtering tree $T(src_i)$. New predicate insertion is done as follows: T_i is traversed in order to find the closest ancestor, *i.e.*, the most selective predicate subsuming $pred_i$, then $pred_i$ is inserted as its child (if it does not already exist in the tree). Query

removal is trivial since it consists simply in checking if the removed predicates are not used by other queries, *i.e.*, they do not have any child and there is no other publication query associated to that predicate.

2. *Filtering tree reoptimization*: in a second phase, once the degradation of any filtering tree surpass a fixed threshold, we recompute the factorization tree and we replace *on runtime* the physical filtering tree by new one.

Example. We illustrate our approach through a simple example that allows showing the degradation of the filtering trees and the need of periodically reoptimizing them. Suppose a filtering tree T_i attached to a source src_i. This filtering tree is composed of three *terminal* predicates ($a \wedge b$, $b \wedge c$ and $a \wedge d$) and an *intermediate* predicate (b), see Figure 3.11a. When a new query q^{new} is registered into the system, we first normalize it to get a new query [10]: $q_N^{new} = \sigma_{pred_1^{new}}(src_1) \cup \cdots \cup \sigma_{pred_n^{new}}(src_n)$. After normalization, suppose q^{new} comes with a new predicate $pred_i^{new} = b \wedge d$ for src_i, to be inserted into its filtering tree. So, we traverse the filtering tree T_i and we insert $pred_i^{new}$ as a child of its *best direct ancestor*. The new predicate $b \wedge d$ is inserted as child of *intermediate* predicate b. Nevertheless, new factorization opportunities appear with new queries and the evaluation performances decreases with time. As we can see in Figure 3.11b, with this new set of predicates we can obtain a better factorization tree, with a lower evaluation cost.



(a) Tree cost after insertion = 2.3

(b) New tree cost = 2.24

Figure 3.11 – Runtime Optimization example

Listing 3.7 details our algorithm step by step for the insertion of new queries. We can see that we maintain for each source a list of its predicates ordered by selectivity. This enables to

10. We consider only queries without join operators

quickly reach the closest predicate that subsumes the new one.

Algorithm 3.7 closest_ancestor_query_insertion

Input: a logical *Query Plan* QP, a new query q
Output: the *Query Plan* QP with new query q

1: $q^N \leftarrow normalize(q)$ // $q^N = \sigma_{pred_1}(src_1) \cup \cdots \cup \sigma_{pred_n}(src_n)$
2: **for all** $pred_i \in q^N$ **do**
3: $predicate\_list \leftarrow get\_ordered\_predicate\_list(QP, src_i)$
4: // $predicate\_list$ is ordered by descending *selectivity*
5: **if** $pred_i \notin predicate\_list$ **then**
6: $found \leftarrow$ **false**
7: $j \leftarrow 0$
8: **while** $\neg found$ **do**
9: **if** $subsumes(predicate\_list[j], pred_i)$ **then**
10: $parent_i \leftarrow predicate\_list[j]$
11: $found \leftarrow$ **true**
12: **else**
13: $j \leftarrow j + 1$
14: **end if**
15: **end while**
16: $add\_arc(QP, parent_i, pred_i)$
17: **end if**
18: **end for**
19: $u \leftarrow create\_union\_operator()$
20: **for all** $pred_i \in q^N$ **do**
21: $add\_arc(QP, pred_i, u)$
22: **end for**
23: **return** QP

3.5.2 When to recompute the filtering trees

So far we have explained how *dynamic* query optimization works in the *ROSES* system. But a fundamental question is still not answered: how do we decide to recompute a filtering tree? In this Section we present the heuristics we use to measure the degradation of filtering trees for taking this decision.

Given the filtering tree T_i of source src_i at time t, defined as $T_i(t)$, we define $T_i'(t')$ as the filtering tree produced by successively applying the preceding *closest_ancestor_query_insertion* algorithm (3.7) on T_i for queries arrived between t and t'. We define $T_i^*(t')$ as the filtering tree produced by rerunning the optimization algorithm on the set of predicates of src_i existing at

time instant t'. Theoretically, our goal is to recompute every filtering tree T_i once the ratio between the tree cost of T_i' and T_i^* exceeds a fixed *relative* threshold θ (*e.g.*, 80 %):

$$\frac{cost(T_i^*(t'))}{cost(T_i'(t'))} \leq \theta \tag{3.8}$$

However, as we have already said before it is not feasible to recompute the best filtering tree every time, *i.e.*, recompute $T_i^*(t')$ at each plan modification, therefore we have to estimate $cost(T_i^*(t'))$ without computing T_i^*. To this purpose we use a *heuristic* based on the following observation: we have seen that the selectivity of a predicate and its *popularity* (*i.e.* the number of predicates that may reuse its results) are the two key factors in order to find the best factorization tree. We can then define the *cost divergence* Δ for a given filtering tree T_i between two time instants t and t' as follows:

$$\Delta cost(T_i, t, t') = \sum_{n \in T_i'} |outdegree(n,t) \cdot selectivity(n,t) - outdegree(n,t') \cdot selectivity(n,t')| \tag{3.9}$$

where $outdegree(n,t)$ is equal to zero for all nodes added to T_i between t and t' (new predicates). Respectively, $outdegree(n,t') = 0$ for all nodes removed from T_i between t and t'. This heuristic allows considering both topological tree modification (this is arrival/removal of operators) and predicate selectivity changes. Moreover, cost divergence is easy to maintain incrementally (independently of the filter tree size).

We can now replace Equation 3.8 by a new equation comparing the cost divergence obtained over some period with the real plan cost at the beginning of the period. So, once the ratio between this *cost divergence* and the real cost of T_i observed at t exceeds a given threshold, tree refactorization is done.

$$\frac{\Delta cost(T_i, t, t')}{cost(T_i(t))} \geq \tau \tag{3.10}$$

This gain estimation test is done periodically, its periodicity may be defined through two different strategies:

- *globally*: either after a fixed number of query arrivals/departures into the system, or impacting a given source,
- *locally*: or when a source has produced a fixed number of items.

The first strategy might be enough if predicate selectivities were constant in time, *i.e.*, if we only have topological modifications of the filtering trees. The second strategy enables to

include both degradation factors. We decided to apply only the second criteria, *i.e.* we check the estimated gain after a source produces a given number of items. This has as a side effect that a source with a high publication rate (and high cost) will be checked very often, while a source with a lower rate (lower cost) is checked less often. However, this difference in the frequency of reoptimizing sources is not an issue since sources with low rates have less impact on the evaluation costs. Consequently, the system maintains for each source an item counter for automatically triggering the *cost divergence* computation every N items published by the source.

Estimating the selectivities

To estimate the cost divergence of a filtering tree T_i between two time instants t and t', we also must estimate the selectivities for all predicates appearing in T_i. We use an *exponential smoothing* function in order to update the selectivities:

$$selectivity(op, t') = \alpha \cdot selectivity(op, t) + (1 - \alpha) \cdot \frac{m}{N} \tag{3.11}$$

where m is the number of items produced by filter op between t and t', N corresponds to the estimation periodicity in terms of number of items ($\frac{m}{N}$ represents the selectivity of the operator between t and t'), and α ($\alpha \in [0, 1]$) is the smoothing factor. We use exponential smoothing in order to take into account past selectivity behavior. For instance, if in the past the selectivity of a given operator have been more or less constant, and at time t' it changes drastically, α parameter enables to smooth this punctual behavior by increasing its value.

Listing 3.8 shows the method run by the the system for each source operator.

3.6 Experimental evaluation

We have conducted an experimental evaluation on our multi-query optimization approach which is detailed in this Section. Thus, first we describe in Section 3.6.1 the synthetic query generator we have implemented to generate an extensive workbench of queries. Then Section 3.6.2 presents the results obtained by the different experiences we have carried out in an increasing order of complexity.

Algorithm 3.8 run_source

```
1: loop
2:    item ← poll(input_queue)
3:    put(output_queue, item)
4:    produced_items(src_i) ← produced_items(src_i) + 1
5:    if produced_items(src_i) ≥ K then
6:       δ ← cost_divergence(T_i)
7:       cost_ratio ← δ / cost(T_i(t))
8:       if cost_ratio ≥ τ then
9:          T_i^* ← recompute_best_filtering_tree(src_i)
10:         replace_physical_query_plan(src_i, T_i^*)
11:      else
12:         produced_items(src_i) ← 0
13:         update_statistics(T_i) // performs the selectivity smoothing for each op in T_i
14:      end if
15:   end if
16: end loop
```

Algorithm 3.9 cost_divergence

Input: a filtering tree T_i
Output: cost divergence δ of T_i between t and t'

```
1: δ ← 0
2: for all n ∈ T_i' do
3:    δ ← δ + |outdegree(n, t) · selectivity(n, t) − outdegree(n, t') · selectivity(n, t')|
4: end for
5: return δ
```

3.6.1 The ROSES query generator

An important issue for the experimental evaluation of our factorization algorithm was the generation of large collections of filtering queries over RSS feeds. We have developed a customizable *ROSES query generator* based on an existing *RSS Subscription Generator*. 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*[11] (which also corresponds to the selectivity) in the searched document collection. This hypothesis fits particularly in the context of RSS feeds, where users are mainly interested in recent news. News items are mostly characterized by the words that exhibit a higher frequency, in the same way user-defined queries contain usually these same words.

The queries generated by our query generator 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 (*src*$_1$ | *src*$_2$ | *src*$_3$ | ... | *src*$_N$) **AS** $var
WHERE $var[*predicate*];

We wanted that the queries produced by our generator were as realistic as possible. Thus, the first we have done is crawling a collection of more than 1300 RSS news feeds during a month, we have stored in a database all the items they have published (approximately 190.000 items), and we have extracted a representative collection of about 300 keywords for each RSS feed along with their document frequencies (*i.e.* the percentage of items which contain the keyword). These keyword distributions are used by the *query generator* for generating the filtering predicates of the queries. The output of the generator is a *query script*, a text file containing a collection of *ROSES* queries (like SQL scripts). The generation of one query is done as follows: first of all the generator chooses a random set of RSS feeds from the database, after that it merges the keyword distributions of these sources, this defines the probability of a term to appear in the query. Then the generator produces the filtering predicate of the query, which contains the most frequent terms of the chosen sources.

Besides the collection of sources and their statistics, the generator can be parametrized with other *parameters* controlling:

- the number of query scripts,
- the number of sources used in the generated scripts ($|S|$),
- the number of generated queries in the scripts ($|Q|$),

11. the probability for a word to appear in a document of the collection

- the number of sources for each query ($q_S \in [q_S^{min}, q_S^{max}]$),
- the syntactic form of the filtering predicates, and
- the percentage of most frequent keywords per source.

All the generated filtering predicates are in *conjunctive normal form* (CNF). They are composed of a random number d of disjunctive clauses (length of the conjunction) picked from a range $[d^{min}, d^{max}]$, where each disjunctive clause 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:

- general filters in CNF,
- simple conjunctive queries (with $a^{min} = a^{max} = 1$) and
- simple disjunctive queries (with $d^{min} = d^{max} = 1$).

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}).

3.6.2 Experiments

In the following we will describe some experimental 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.

3.6.2.1 Experience I: 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). Therefore we have generated different sets of queries over a single source:

$$|Q| \in \{10, 20, \ldots 100, 200, \ldots 1000, 2000, \ldots 10\,000\}$$

We limited the filtering predicates to simple conjunctions of one to three atomic filters, a realistic scenario for search engines ($d^{min} = 1$, $d^{max} = 3$, $a^{min} = a^{max} = 1$). Remind that these predicates are generated according to the distribution of 300 keywords in the filtered feed.

The obtained results are shown in Figure 3.12.

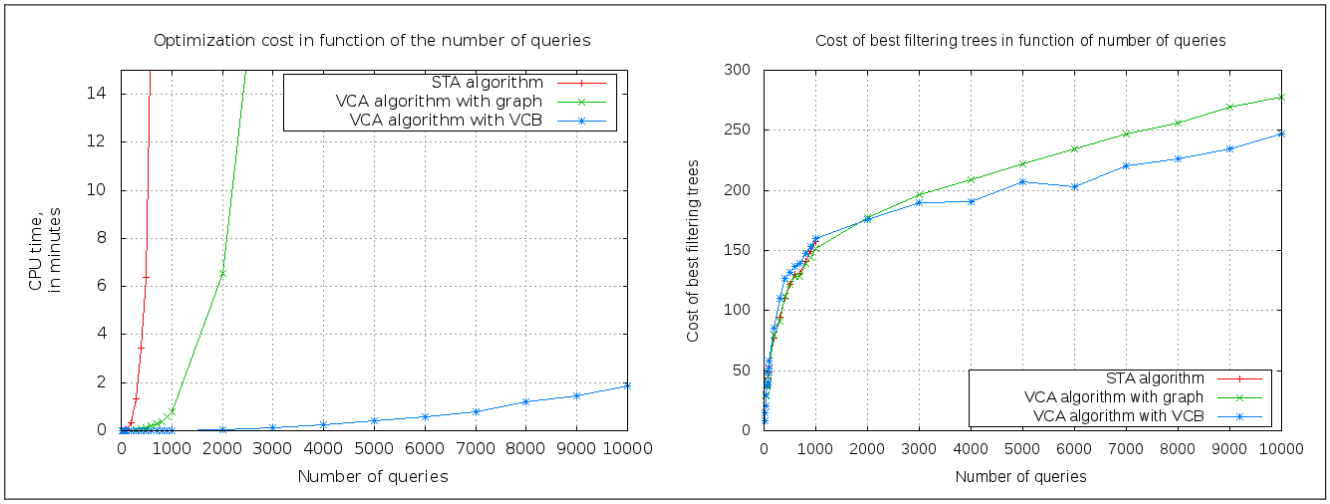


Figure 3.12 – CPU time & Cost of trees for experience I

The figure on the left side shows the evolution of the optimization cost with 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 queries in two minutes. On the right figure, we can see that the filter plans generated by all three algorithms have a similar *evaluation cost*. We might expect that both versions of VCA (with graph and with VCB) should generate the same multi-query plans. We explain the difference between the plan costs by the non-determinism of the choice of the best candidate based on its selectivity.

We have also measured the *memory usage* of both structures in these settings: the *subsumption graph* and the VCB. The size of the VCB data structure decreases during the execution of the factorization algorithm. Thus, memory cost for the VCB data structure is measured for first step of the factorization algorithm. Figure 3.13 shows that memory usage with VCB clearly is divided by a factor of 3.

We observe an *asymptotic* behavior in the *subsumption graph* curve. This is due to the fact that the number of keywords per source is limited to 300. Thus, at some moment the maximum number of conjunctive predicates of size 3 is reached and the *subsumption graph* attains its saturation state, *i.e.*, all possible predicates are already present in the graph.

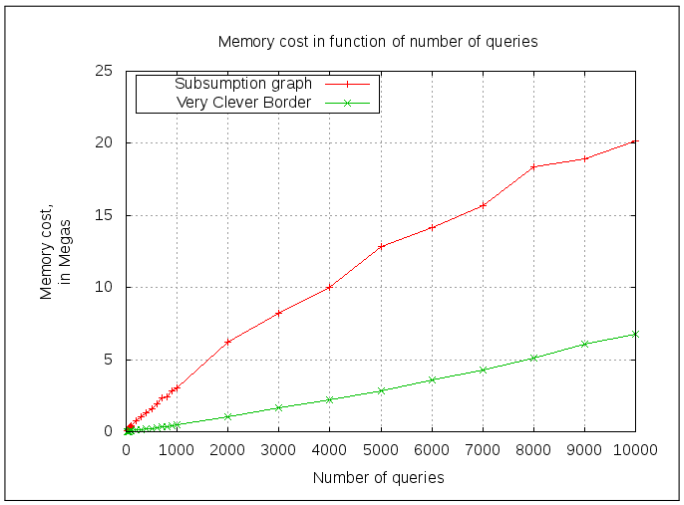


Figure 3.13 – Memory cost for experience I

3.6.2.2 Experience II: 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 *number of queries per script* ($|Q|$) to 1000 and generated five sets of queries with an increasing number of disjunctive clauses per predicate (d^{max} goes from 1 to 5). Meanwhile the maximum length of these disjunctive clauses is fixed to 5. In other words: $d^{min} = d^{max} = 1$, $a^{min} = 1$, $a^{max} = 5$ for the first test; $d^{min} = 1$, $d^{max} = 2$, $a^{min} = 1$, $a^{max} = 5$ for the second on, and so on.

Our results are shown in Figure 3.14.

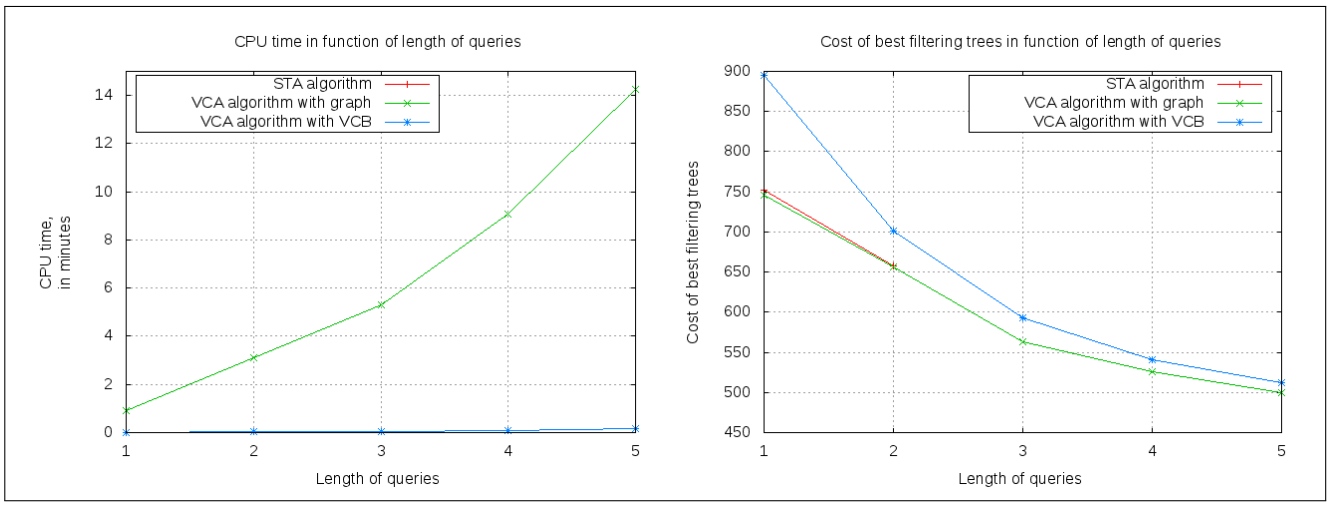


Figure 3.14 – CPU time & Cost of trees for experience II

In the left figure, we can see that the cost of *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 filtering 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*.

3.6.2.3 Experience III: Multiple sources

Our third experiment corresponds to a more realistic scenario where queries are defined on multiple sources. We have generated a workload of 10 000 queries over a fixed set of 500 sources, where each publication query is defined on a random number q_S of sources ($q_S \in [1, 10]$). Filtering predicates associated to each query are defined as conjunctive predicates of random length $d \in [1, 3]$. However, in order to reduce the generation of empty filtering conditions ($selectivity = 0$), we have used the top 5% of the most frequent keywords in each source. Figure 3.15 shows the CPU cost associated to the normalization process for workload subsets of different size, while Figure 3.16 illustrates the optimization cost and the query plan cost for each factorization algorithm.

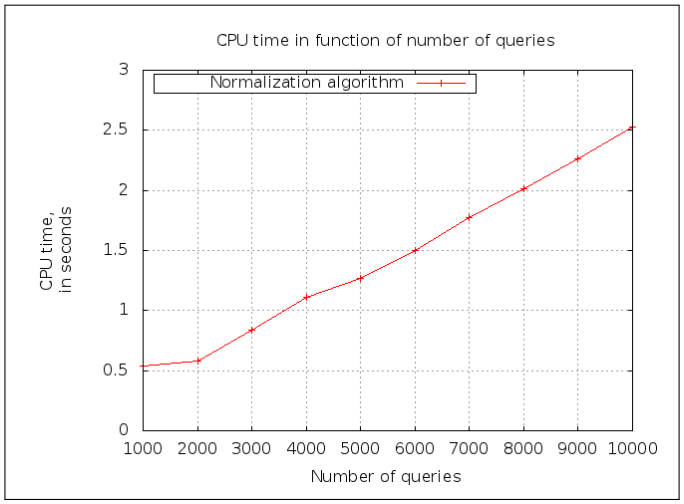


Figure 3.15 – CPU time of Normalization algorithm for experience III

First, we can see that the CPU cost of the normalization process (Figure 3.15) grows linearly with the number of queries. For 10 000 queries we observe a normalization cost of 2.5 seconds.

On the other hand, we can see in Figure 3.16 that the optimization cost is lower than in previous experiments (with one source only) for the same number of queries. This is due to the fact that each query only concerns a limited number of sources, which decreases the average number of filters for each source (optimization cost is exponential *w.r.t.* the number of filtering

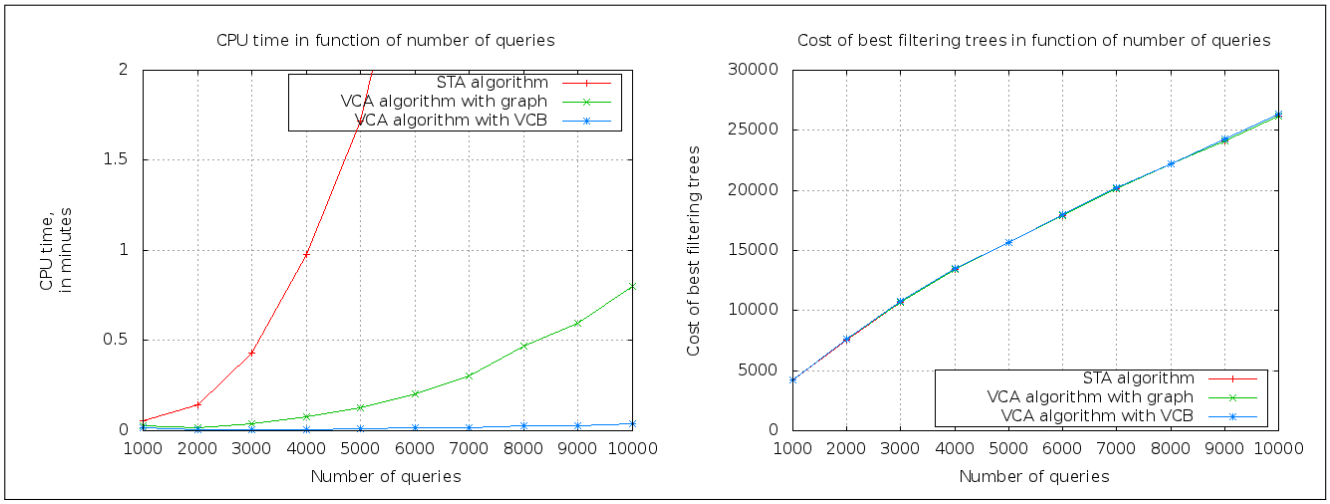


Figure 3.16 – CPU time & Cost of trees for experience III

predicates and linear in the number of sources). We can see that the resulting query plan costs obtained by all three algorithms are very close (in the right graph).

In conclusion, for a workload of queries aggregating many heterogeneous sources and with different conjunctive predicates, our experimental evaluation shows that all three algorithms produce very close results (the filtering trees they generate have similar evaluation costs), but VCA+VCB factorization finishes much faster than the others.

3.6.2.4 Experience IV: Cost model validation

Finally, our last experiment searches to validate the *cost model* that we have proposed. This is why we have developed an *RSS Publishing Emulator*. This *emulator* may replay the item publication history of a given set of RSS sources. It respects the original time elapsed between two item publications, however it may be tweaked in order to speed up the whole emulation process.

We use our *Emulator* to reproduce the RSS publication history that we have crawled during a month. This allows computing the CPU time spent by the *query graph* and its operators when processing real items produced by the emulator. In this experience, we reuse the parameter settings of the preceding experience, except for the number of queries that goes until 50 000. Table 3.5 summarizes this configuration settings.

Thus, we have replayed our *one-month* feed experience through our system without query graph optimization first, and then with optimization. The "*without optimization*" experience means that query plans follow the syntactic form of the queries, *i.e.*, first union operators and then the corresponding filtering operator. In the "*with optimization*" experience we have used

$\|Q\|$	50 000
$\|S\|$	500
q_S	$\in [1, 10]$
d	$\in [1, 3]$
a	1
top 5% most frequent keywords per source	

Tableau 3.5 – Experience IV parameter configuration

the VCA + VCB algorithm. We have measured the CPU usage for all operators composing the query graph for each experience. Figure 3.17 shows the results we have obtained.

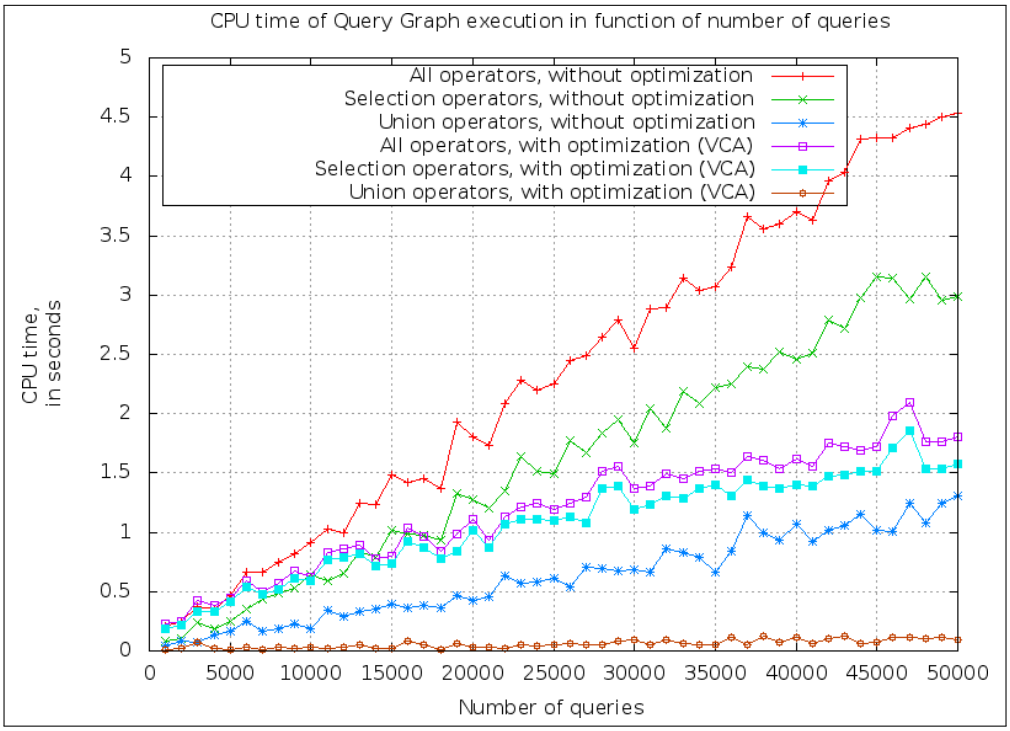


Figure 3.17 – CPU time of the Query Graph execution with & without optimization

The red line (+ symbol) corresponds to CPU usage of both the selection ($\times$) and union ($*$) operators. Respectively, the violet line ($\square$ symbol) corresponds to the sum of the cyan (selection) and brown (union) lines. We can see that the gain reached by the optimization grows with the number of queries.

Finally, Figure 3.18 shows the *theoretical cost* estimated by our cost model (red line) as well as the *real CPU cost* measured through this experience (green). Theoretical cost considers only the estimated cost of selection operators, this is the sum of arc weights of the filtering trees

generated by the optimization algorithm. Thus real cost considers only CPU usage of selection operators (*i.e.* it corresponds to the cyan line (■) on Figure 3.17). We can see that the curve of the estimated cost fits the CPU cost measured by the system.

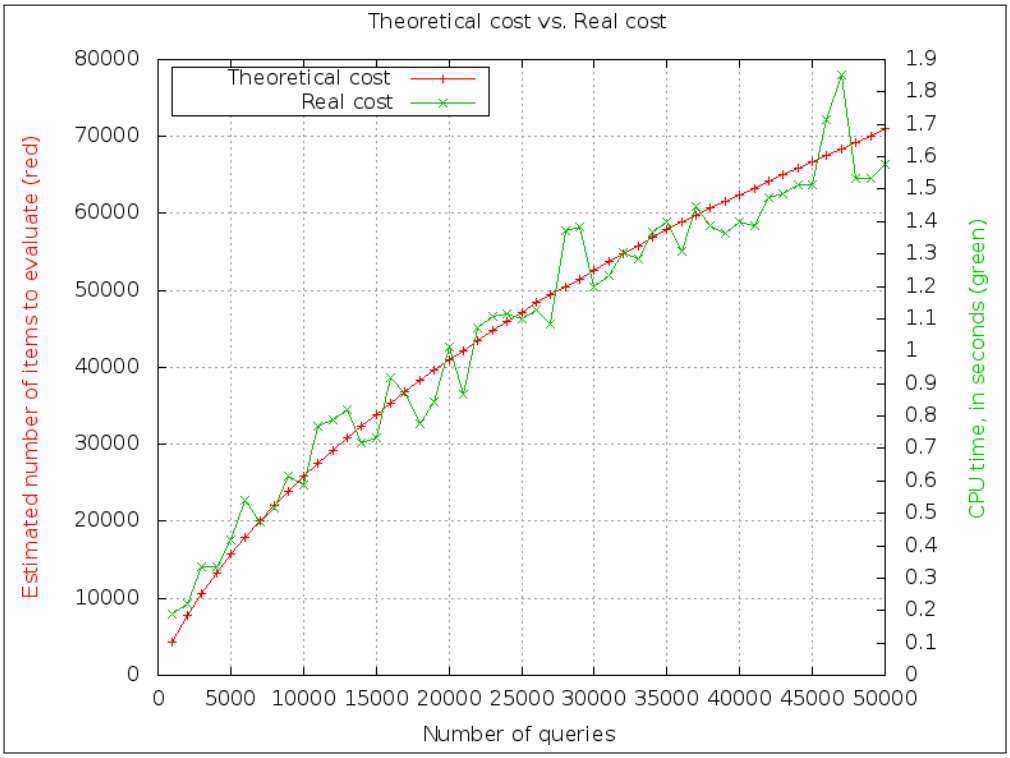


Figure 3.18 – Estimated number of evaluated items by our Cost model vs. real cost

Chapter 4

ROSES System Architecture and Prototype

The processing and optimization principles presented in this thesis have been validated by the implementation of a prototype of the *ROSES* system. In this Chapter, we first describe the architecture of the *ROSES* system (Section 4.1). Then, Section 4.2 presents some of the difficulties arisen during the development of the *ROSES* prototype and how we have addressed them. Finally we give an overview of the prototype and the implemented functionalities in Section 4.3.

4.1 ROSES system architecture

The *ROSES System* is composed of five modules responsible for processing RSS feed queries and managing meta-data about users, publications and subscriptions. As shown in Figure 4.1, RSS feeds are processed by a three layered architecture where the top layer (*Acquisition*) is in charge of crawling the set of RSS and Atom feeds used by the publication queries (the source feed set) and to detect and transmit new items to the *Evaluation* module. The second layer (*Evaluation*) maintains a continuous *query plan* which comprises all publication queries and it is responsible for the processing and optimization of such query plan. Finally, the third layer (*Dissemination*) deals with publishing the results according to the registered subscriptions. 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.

4.1.1 Acquisition module

The *Acquisition* module is responsible for crawling the *sources* used by the user-defined queries. These sources may be of very different kind: *Syndication Feeds* (RSS and Atom), Web Services, periodic database queries, etc.[1] Thereby the ROSES System enables the definition

1. Only RSS and Atom crawling has been implemented.

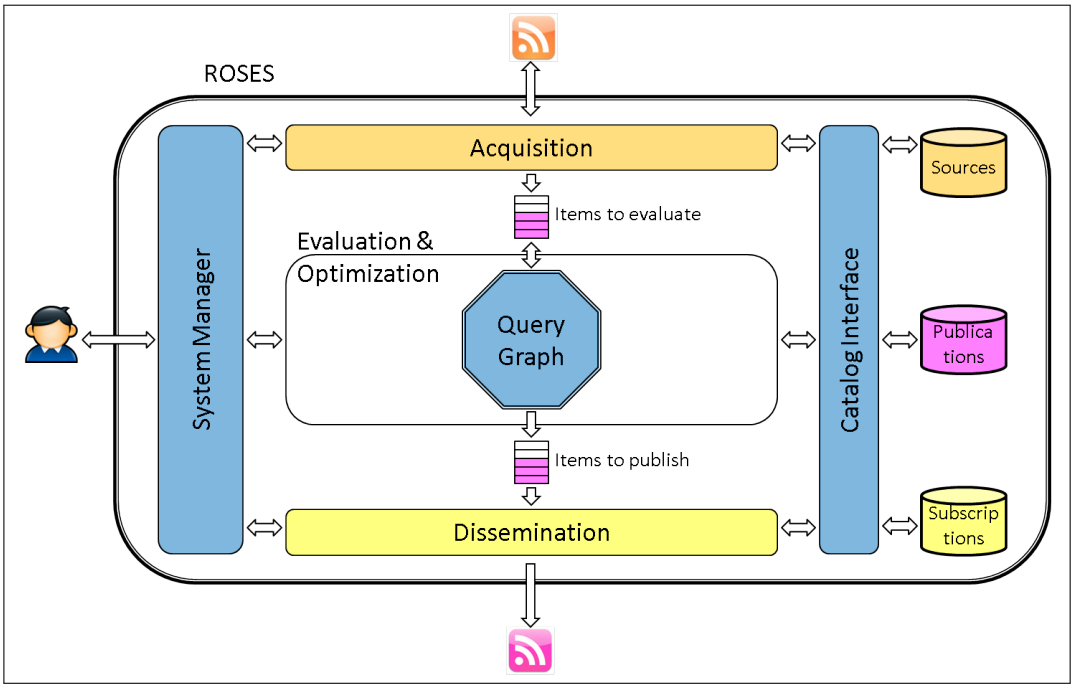


Figure 4.1 – ROSES System Architecture

of wrappers for each kind of source. These wrappers transform the crawled items into a homogeneous format (the ROSES item, see Section 2.2.1), which is used thereafter by the rest of modules.

The main issue handled by the Acquisition module is to detect new items published by external sources using a refresh protocol in pull mode. This protocol has to estimate the date of the next update for every source. In fact, every RSS feed has a different update periodicity and the crawler must progressively adapt the crawling to the frequency of each source, in order to guarantee the completeness of the source without unnecessarily requesting them too often. This means: minimize the data loss rate as well as minimize the bandwidth usage for requesting the sources.

An efficient refresh strategy optimizing the bandwidth usage can be found in [HAA10], where the authors propose a best-effort strategy for refreshing RSS documents under limited bandwidth and introduce the notion of saturation for reducing information loss below a certain bandwidth threshold.

4.1.2 Evaluation module

The core of the ROSES System is an *algebraic multi-query plan* encoding all registered publication queries. As presented in Chapter 3, the evaluation of this query plan follows an

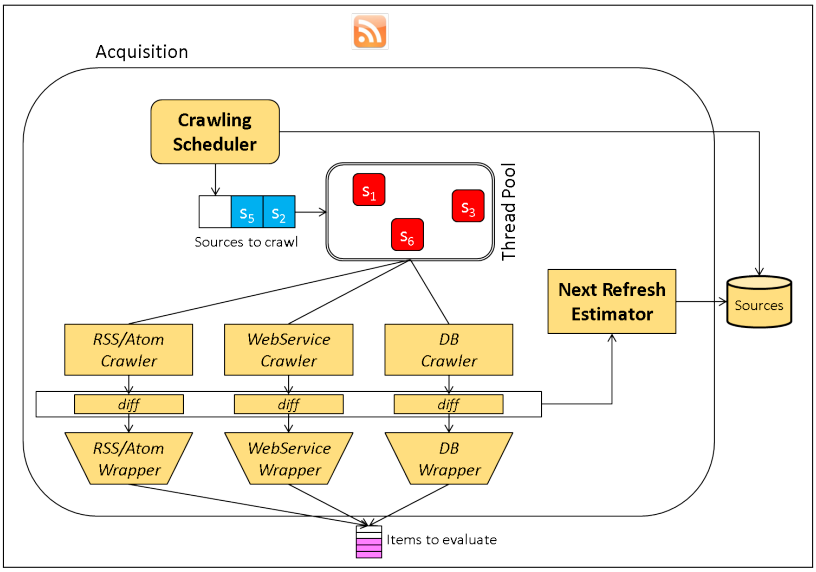


Figure 4.2 – Acquisition Layer overview

asynchronous pipe-lined execution model where the Evaluation module:

- continuously evaluates the set of algebraic operations according to the incoming stream of ROSES items, and
- manages the addition, modification and deletion of publication queries.

We can see in Figure 3.1 an overall image of this module. Here, the query graph is composed of three simple publications: $p_1 = \sigma_1(src_1 \cup src_2)$, $p_2 = (src_3 \cup src_4) \bowtie_1 (\omega_1(src_5))$ and $p_3 = \sigma_2(p_1 \cup src_6)$. Notice that publication p_3 is defined on another publication (p_1), that is why *publication operators* forward their input items to both (i) the *Dissemination* module and (ii) to an intermediate queue in order to be used by operators of other publications. Notice as well that the kind of queue between the operators ω_1 and $\bowtie_1$ is different from the rest of queues. A detailed description of the query engine module is given in previous Section 3.1 and later in Section 4.2.

The black boxes with rounded corners represent the threads that are continuously executed. The *Dispatcher* component retrieves new items detected by the Acquisition module and pushes them in the corresponding source queue of the query graph.

The *query graph* itself is processed by continuously executing its constituent operators. Our query execution approach is based on a multi-threaded model where every operator is processed by a different thread asynchronously. Technical constraints make it unfeasible to assign a thread to each operator, since the number of operators in the query graph is generally much larger than the number of threads that might be run simultaneously (*ROSES* is implemented in Java which is limited to about a thousand threads per process). Thus, we have resorted to a *Thread*

Pool mechanism.

An operator *Scheduler* continuously observes the operator graph and decides which operator should be evaluated next and sends it to the *thread pool*'s task queue. Executing an operator consists in creating an execution task which is sent to the Thread Pool. A Thread Pool contains a variable number of threads bounded by a minimum and maximum value. Each thread of the Thread Pool may be either running a task or idle. Tasks are inserted into a waiting list (FIFO) and executing a task consists in assigning it to a waiting thread. When an idle thread exceeds a fixed time-to-live, the thread is removed from the Thread Pool. Respectively, when all available threads are running some task and the waiting list exceeds a given size, the system automatically creates new threads (in the pool) to process the waiting tasks.

When an operator is picked by the scheduler, it is immediately marked as *active*, and it remains in this state until it is taken by one thread of the *thread pool* and completely processed. Then it is marked back to *passive*. An *active* operator can not be selected again by the scheduler even if it is not being executed but just waiting in the task queue. This allows to prevent that two different threads simultaneously run the same operator, which might entail (a) altering item's ordering on the feeds, or (b) read/write conflicts on its input/output inter-operator queues. In Figure 3.1, red colored operators represent the active operators, *i.e.* those operators that are currently executed by a thread; orange operators represent the operators scheduled to be executed and waiting for a free thread; and blue operators represent the passive ones. This approach enables an asynchronous graph processing.

We have implemented two different *scheduling strategies* for assigning operators to tasks, a *Random* and a *Round-robin* strategy:

- Under the *Random strategy* the Scheduler picks a random *passive* operator from the physical query graph and verifies if its input inter-operator queue has items to evaluate, before sending it to the thread pool. This trivial approach guarantees that *all* query graph operators are uniformly evaluated at a given moment.
- The *Round-robin strategy* chooses the next operator to evaluate in a fixed order determined by the operator arrival in the query graph. However, this strategy may be used to implement a *priority* evaluation ordering, where operators are processed in function of different criteria, *e.g.* the number of items in the operator's input queue or the time elapsed since last operator execution.

The *Manager* component handles the insertion, suppression and update of publication queries in the operator graph. Graph updates are done online during graph processing, namely graph evaluation is not stopped in order to insert/delete queries. In fact, query insertion is performed through the *smallest ancestor* algorithm presented in Section 3.5.

Finally, the *Runtime Optimizer* is in charge of updating the operator graph in order to keep it optimized. The query plan efficiency/optimality degrades over time due to: (1) the continuous arrival of new publication queries and/or removal of existing queries, (2) the changing publication rate of the sources and (3) the predicate selectivity variation over time. For this purpose, the system must periodically reoptimize the query plan at runtime. In Section 3.5 we have presented our runtime optimization technique.

4.1.3 Dissemination module

The *Dissemination* module is in charge of notifying users with new items produced by the publications that they have subscribed. Users can define several subscription types: by SMS, email, RSS/Atom feed..., as well as the notification frequency [2]. In an analogous manner to the Acquisition module, this module has a different *Diffuser* for every type of subscription, so once new items are produced by a publication, the system checks which subscriptions are concerned by those items and it dispatches them to the appropriate Diffusers.

4.2 Prototype implementation details

The *ROSES Prototype* is composed of a server and a client and has been implemented on *Java* 6, it makes extensive use of *java generics* and the new concurrent *thread-safe* objects introduced in this java version. The *ROSES* Prototype is composed of 231 classes and more than 25 900 lines of code. Both, the *ROSES* server and the client, are publicly available on the *ROSES* Website [3] and can be downloaded from http://www-bd.lip6.fr/roses/doku.php?id=prototypes.

Inter-operator queues

A first technical problem we faced in the development of the *ROSES* prototype concerns *inter-operator queues*, the buffers connecting physical operators and which enable the asynchronous execution model of our *system*. Our inter-operator queues do not obey the classic behavior of multi-threaded queues, *i.e.*, with multiple producers and multiple consumers and where each element of the queue is processed by just *one* of the consumers. Instead our inter-operator queues have always one unique producer operator and multiple consumers but every

2. Only RSS/Atom subscription output has been implemented.
3. http://www-bd.lip6.fr/roses/

element of the queue must be processed by *every* consumer. Java's concurrent queues rely on first kind of queues (multiple producers and consumers), furthermore they add a processing overhead due to the synchronization of threads on push and poll operations. We wanted our queues to be as fast as possible since they are a crucial component of our evaluation module. Thus we have implemented them ourselves.

The particular behavior of our *inter-operator queues* avoids thread synchronization overhead by exploiting the fact that: (1) only one thread (the producer operator) may add items to the queue, and (2) each consumer thread may access the queue separately through the use of independent iterators. Figure 4.3 depicts a schema of our inter-operator queues. They are

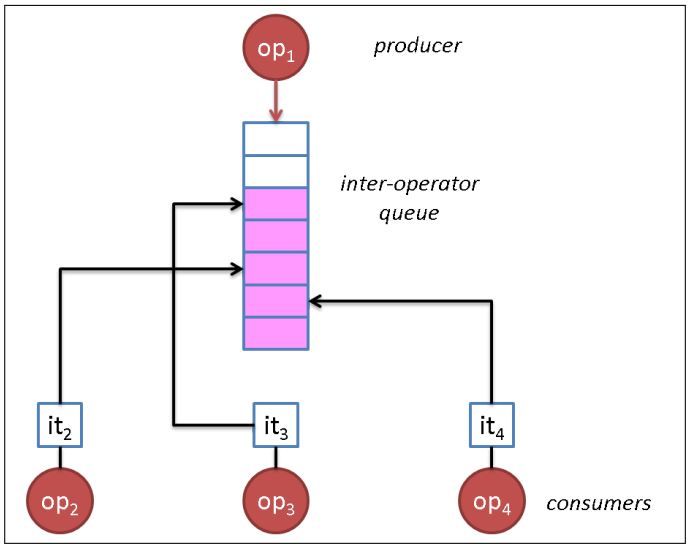


Figure 4.3 – Consumer iterators schema

implemented as *linked lists* with a pointer to the tail (where new items are added) used by the producer operator, and every consumer has its own iterator with its pointer to some node of the queue. With this implementation when a new consumer is added on-runtime to the queue (a new selection operator for instance), its iterator will point to the tail node, and when an item has been evaluated by all queue consumers, its node will be automatically removed by java's *garbage collector*.

Time-based window queues

Another interesting issue concerns the time-based window buffers, *i.e.*, the time-varying buffers produced by time-based window operators. These buffers vary over time in function of the publishing rate of the corresponding input feed and the window length specification. As

well as for inter-operator queues, we wanted to avoid conflicts between threads accessing this queue, *i.e.* the thread running the window operator and any thread running a join operator on this window. Figure 4.4 illustrates the schema of a window-join operation.

We have implemented time-based window queues also as *linked-lists*, with both pointers *head* and *tail*. The tail pointer is only modified by the window thread when adding new items to the queue. The head pointer (pointing the oldest item in the queue) must be periodically updated in order to throw away items already out-of-date. Thus, join evaluation is performed as follows: when a thread runs a join operator, it first updates the head pointer of the corresponding window queue, that is it shifts the pointer until the first valid queue node according to the current time. Then for each item of the join's input queue the *window iterator* traverses the window queue from the current head to the current tail and evaluates the join predicate on the corresponding input item and each window item. This evaluation model guarantees that any thread involved in the execution of a join (*i.e.* the thread running the window operator and any thread running a join operator on that window) accesses different parts of the window queue. Thereby they can not incur in an access conflict.

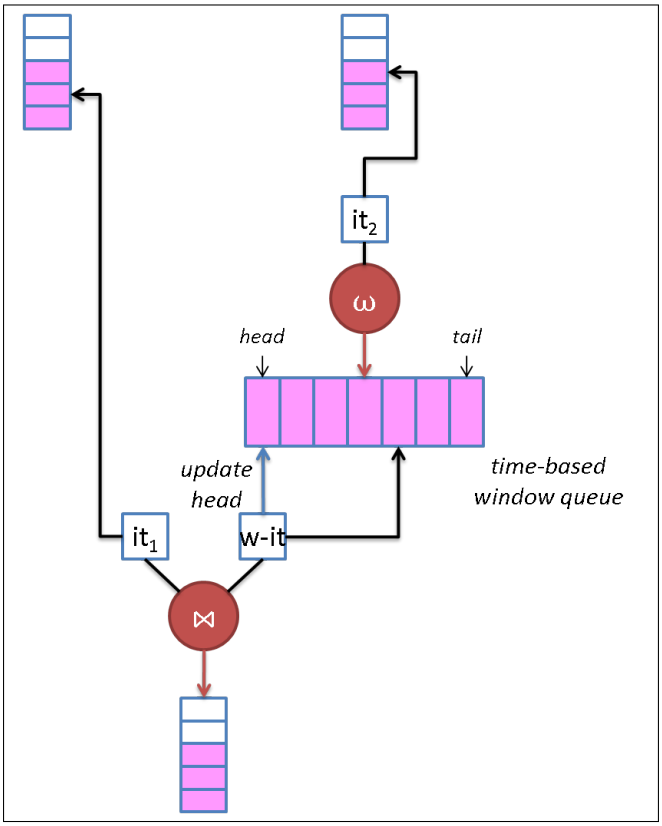


Figure 4.4 – Window-join schema

Similarity predicate on join operators

We have implemented a similarity predicate for join operators that allows to evaluate if two textual fields of different items are similar or not (*e.g.* items' titles or descriptions). To this purpose, we have made use of the *SimMetrics* library [4], an open-source library developed at the University of Sheffield by Sam Chapman that furnishes a complete collection of string distance functions: *Jaccard* Similarity, Q-grams distance, *Levenshtein* distance, Overlap coefficient, etc., through different string tokenizers: simple whitespace tokenizer, 2-gram and 3-gram tokenizer.

4. http://www.aktors.org/technologies/simmetrics/

After some experimentation on real newspapers feeds, we found that the overlap coefficient metric used on 3-gram tokenized strings performs well (in quality and time) on short RSS item titles (two or three words), while this same metric function but with the whitespace tokenizer is more appropriate (and faster) for longer titles. On the other hand for items' description fields, usually much longer than title attributes, we use the Q-grams distance with the 3-gram tokenizer since it appeared to produce better results than the rest.

This kind of join predicate allows to easily aggregate items published by different feeds but talking about the same topic into a single item, for instance, when users are interested in seeing how different newspapers talk about the same current affairs.

Feed refresh strategy of the Acquisition module

An important piece of the Acquisition module is the RSS/Atom Crawler and its feed refresh scheduler. We have adapted an RSS/Atom Crawler provided by *2or3things*, an industrial partner of the *ROSES* Project, in order to fit into our requirements [5]. The main problem with feeds' refreshing it that each source feed has a different refresh frequency and furthermore there is no uniform way to update RSS/Atom files:

- Sometimes they are updated by entirely replacing the file by a new one with new items.
- At other times new items are directly added to current file and older items are periodically removed.
- Sometimes items are listed in ascending order in the XML document and sometimes in descending order or even with no order at all.

Another problem is that the *GUID* of RSS items is not mandatory on RSS Specification [6] and often is missing in RSS files. Thus, we have created our own identifier based on the *MD5* hashcode of the title of the item, its publishing date and the *guid* (if present), in order to uniquely identify feed items. This identifier is used therefore every time we refresh a source feed, in fact, along with other statistics and metadata, we store the identifier of last crawled item for each source. This enables to rapidly identify if last crawled item is still in the RSS/Atom file the next time that we refresh the source, and avoid sending it twice in the corresponding stream.

On the other hand, the feed refresh strategy has to continuously adapt the estimated time of the next RSS/Atom file update for each feed, in order to fit as well as possible to real feed updates. To this purpose, we have defined a simple yet effective *feed update estimation* based

5. Thanks to Nicolas TRAVERS from CNAM
6. http://www.rssboard.org/rss-specification

on the number of new items found at every feed refresh. Thus we define the estimated duration δ between two source updates as follows:

$$\delta_{new} = \begin{cases} \delta_{old} \cdot 1.5 & \text{if error} \\ \delta_{old} \cdot 1.35 & \text{if no new items} \\ \delta_{old} & \text{if between 1 and 5 new items} \\ \delta_{old} \cdot 0.33 & \text{if all items are new} \end{cases}$$

At each refresh of a given source *src*, we reestimate the time interval δ_{new} we will run the next refresh. This estimation depends on the old refresh interval δ_{old} and the source behavior observed at the refresh instant. So the next refresh date is delayed when an error is encountered, for instance if the RSS file is not currently available. Otherwise, it is modified in order to accommodate itself to the real update frequency of the feed. If the source feed has no new items, we increase the refresh time interval. When it provides 1 to 5 new items, we leave the interval value untouched. Otherwise we decrease it if all items are new.

Inter-module communication

All three modules composing the *ROSES* System (Acquisition, Evaluation and Dissemination) are coupled via large *thread-safe* java concurrent queues. The Acquisition and Evaluation modules are connected through the *items to evaluate* queue and the Evaluation and Dissemination modules through *items to publish* queue (see Figure 4.1). Despite the processing overhead of these synchronized queues, we were compelled to use these data structures since they enable multiple threads to access them concurrently on write mode (in both cases, Acquisition and Evaluation modules). In the Acquisition module, multiple threads may run feed refresh tasks in parallel and, thus, send their results to the *items to evaluate* queue, while in the Evaluation module multiple threads may run the ultimate operator of a query and, hence, access concurrently to the *items to publish* queue.

Whereas the use of these queues might be seen as a bottleneck in the System, the advantage of such an approach is that these modules may be entirely uncoupled and hence be run in a distributed environment, where every peer might run different modules. For instance, in a *peer-to-peer* environment a subset of peers might be in charge of the acquisition tasks, another subset of peers might be responsible for the evaluation of the query graph, and a last subset of the dissemination of the resulting publications.

Replacing filtering trees

During the *runtime optimization* process a delicate issue is the replacement of the currently running filtering tree by a new tree produced by the optimizer. Given an existing filtering tree T_i and a new optimized tree T_i^*, first we have to do is blocking the item production of its source operator src_i. Then all filtering operators from T_i are forced to consume all current items in their inter-operator queues. Once all items consumed, all filtering operators appearing in T_i may be removed and replaced by T_i^*. Then src_i item production can be resumed. A detailed algorithm is given next in Listing 4.1.

Algorithm 4.1 replace_physical_filtering_tree

Input: a source src_i, the new optimal plan T_i^*
Output: physical filtering tree T_i' of source src_i is *safely* replaced by new T_i^*

1: $pause(src_i)$ // blocks src_i
2: $force\_operator\_execution(T_i')$ // all $op \in T_i'$ are executed until their queues are empty
3: $remove\_all(T_i')$
4: $generate\_physical\_operators(T_i^*)$
5: $resume(src_i)$ // unblocks src_i

Finally, we also must take care that filtering tree replacement is not performed at the same time that we try to insert/remove a new/existing predicate on the same filtering tree. That is *closest ancestor query insertion* algorithm (3.7) and *replace physical filtering tree* algorithm (4.1) may enter in conflict if executed in parallel. This may be easily addressed through a synchronization strategy. These tasks can be scheduled in order to execute them separately.

4.3 Overview of the ROSES client functionalities

The *ROSES Client* graphical interface has been realized with the IBM's *Standard Widget Toolkit*[7], a widget toolkit alternative to *java*'s AWT and Swing that performs faster than its java competitors by using *native* operating system (OS) widgets.

Figure 4.5 shows a screenshot of *ROSES Client* main window. It is composed of three panels. The left panel lists all available *feeds* (sources and publications). When a user double-clicks one feed, a new tab is open in the main panel. This new tab contains different information about the feed depending on whether it is a source or a publication. Source tabs show a *description* of the source including its ID, name, URL, set of keywords, last update date and the next scheduled refresh date, as well as an *embedded browser* displaying the contents of the

7. http://www.eclipse.org/swt/

source feed (see Figure 4.6). Concerning publication tabs, its *description* panel includes the publication ID, its name, the query that defines the publication, the creation date and the URL of the feed generated by the query. Users may directly visualize the contents of the feed thanks to the *embedded browser*, furthermore, the publication tab contains a visual representation of the logical query plan corresponding to the query (see Figure 4.7).

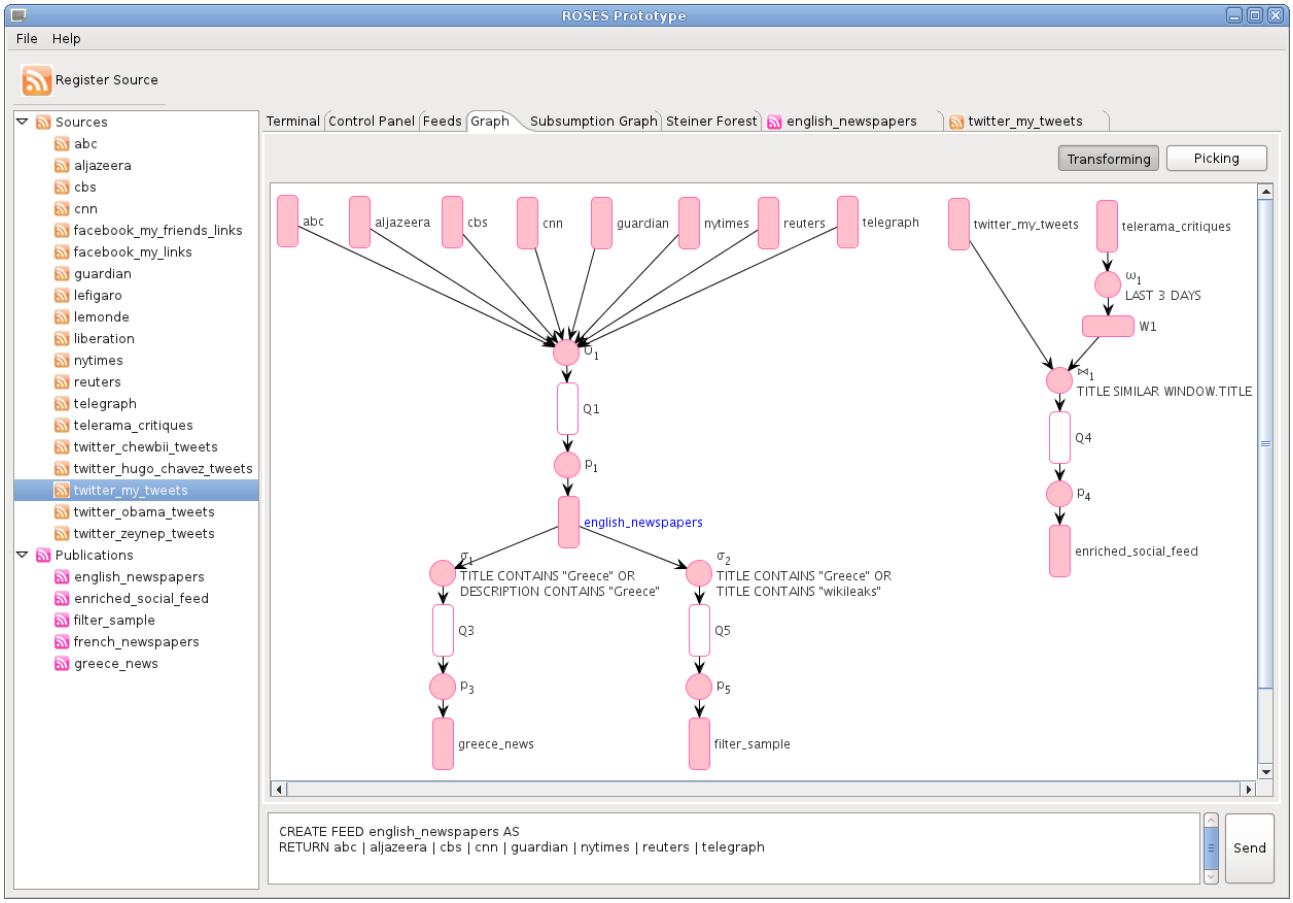


Figure 4.5 – Main window screenshot of ROSES client Prototype

Apart from the source and publication tabs, the main panel contains other fixed tabs such as the *terminal* or the *physical graph* tabs. On the physical graph tab, users can visualize the currently running physical operator graph which comprises all publication queries. Physical graph visualization, as well as single logical query plan visualization, has been implemented using *dot*, an *AT&T*'s GraphViz tool[8]. *dot* enables to easily generate pretty visual graphs through the *DOT Language*[9] without worrying about graph layout issues.

8. http://www.graphviz.org
9. http://www.graphviz.org/content/dot-language

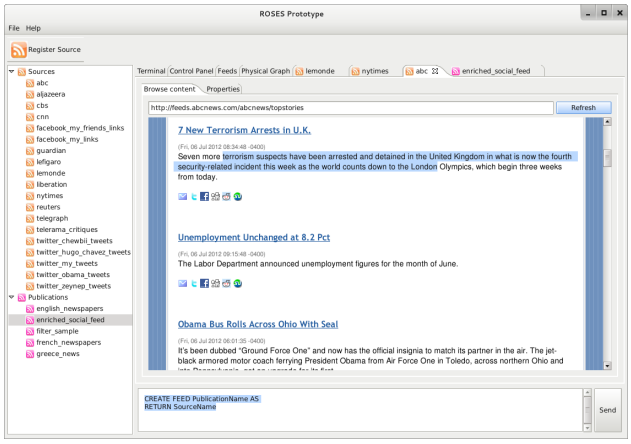


Figure 4.6 – Browsing *ABC*'s source feed contents

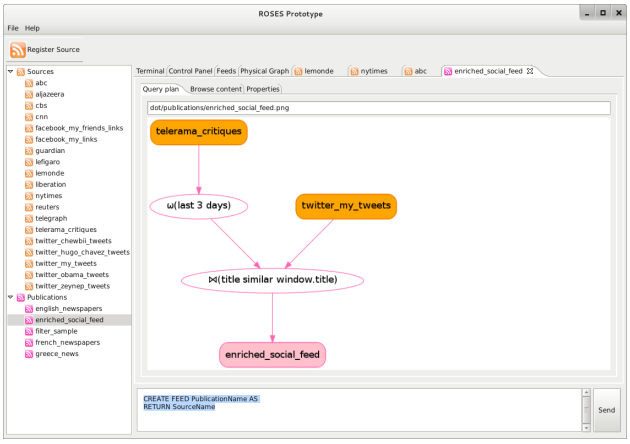


Figure 4.7 – Visualizing logical query plan for *enriched_social_feed* publication

On the other hand, the *terminal* tab enables users to directly interact with the *ROSES* server. In fact, users can submit new publication queries to the System through the little panel situated at the bottom of the client's window. The new query is parsed and, when everything is fine, the publication is inserted into the system and its physical query plan is inserted into the query graph; then a confirmation message is shown into the *terminal*. Otherwise, when there is any syntactic or semantic problem with the query or any other error (*e.g.*, there already exists a publication with the same name, or there is an unknown source feed in the query), the corresponding error message is displayed into the terminal.

Users can also register new sources into the system via a simple form dialog, where they give the name and URL of the feed, and some optional descriptive keywords (Figure 4.8).

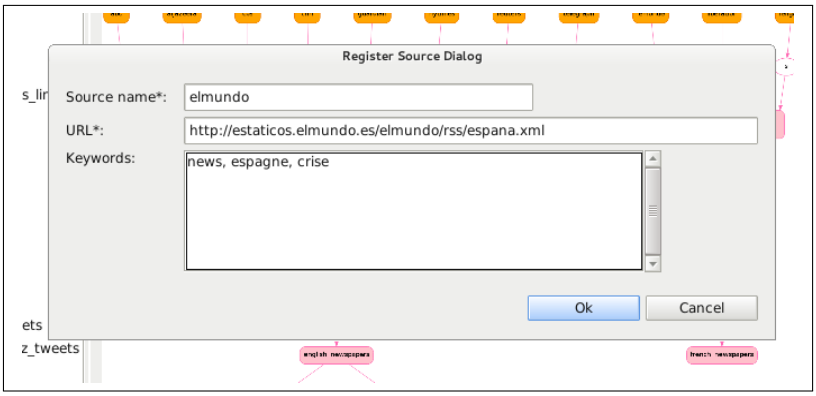


Figure 4.8 – Registering a new source feed

Another functionality implemented in the ROSES prototype is the possibility to import *roses scripts* through the *File* menu. A *roses script* is a text file containing *ROSES* queries.

Once the user chooses the corresponding file, all queries are parsed and dynamically inserted into the execution engine. Again, error messages are displayed within the terminal if necessary.

Finally, all the parameters concerning the *ROSES Prototype* can be adjusted through an XML file. This file, listed in Appendix A.2, allows defining for instance the back-end database used by the prototype (MySQL or H2), the evaluation *thread-pool* characteristics, the scheduling strategy, the type of files produced by ROSES publications (RSS, XML, text, etc.), or yet the factorization algorithm used by the optimizer.

4.3.1 The ROSES Query Builder web application

Besides our desktop application, we have developed a web application that implements an easy-to-use *ROSES Query Builder*. This ROSES Query Builder enables users with no knowledge on declarative languages to create complex publication queries by means of an easy-to-use visual programming interface.

Figure 4.9 shows a screenshot of the *ROSES Query Builder*'s main page. This web interface has been developed using Google's GWT (*Google Web Toolkit*) [10], a novel paradigm on AJAX programming introduced by *Google Developers*. GWT allows developers to implement complex interactive AJAX applications with no need to write a line of AJAX. GWT applications are written in Java and afterward transformed into HTML and JavaScript files thanks to the GWT's *Java-to-JavaScript* compiler. Furthermore, AJAX applications produced by GWT ensure cross-platform, cross-browser and javascript code obfuscation and optimality.

The *ROSES Query Builder* web application implements multi-user access: once a user connects to the system, he can see his own sources/publications. He can see as well the public sources/publications shared by other users. Sources and publications can be easily filtered out through the *adaptive filter* implemented above the feed list (*e.g.*, all French feeds within the *sports* category). Then the user can create a new publication defined on the resulting source list (he can also fine-tune the resulting source list by manually choosing the feeds he is interested in). After that, he can define filter and/or join operations on the union of the selected feeds (*e.g.*, filter out all those feed items whose title contains '*Raymond Domenech*', or join all selected sources with the RSS feed of his own *blog*). He may repeat these steps in order to compose the different operators and build more complex publications. Finally, when the user validates the publication, the corresponding *ROSES* query is generated, the publication is inserted into the *ROSES* query engine and he can already begin to check the first publication results.

10. https://developers.google.com/web-toolkit

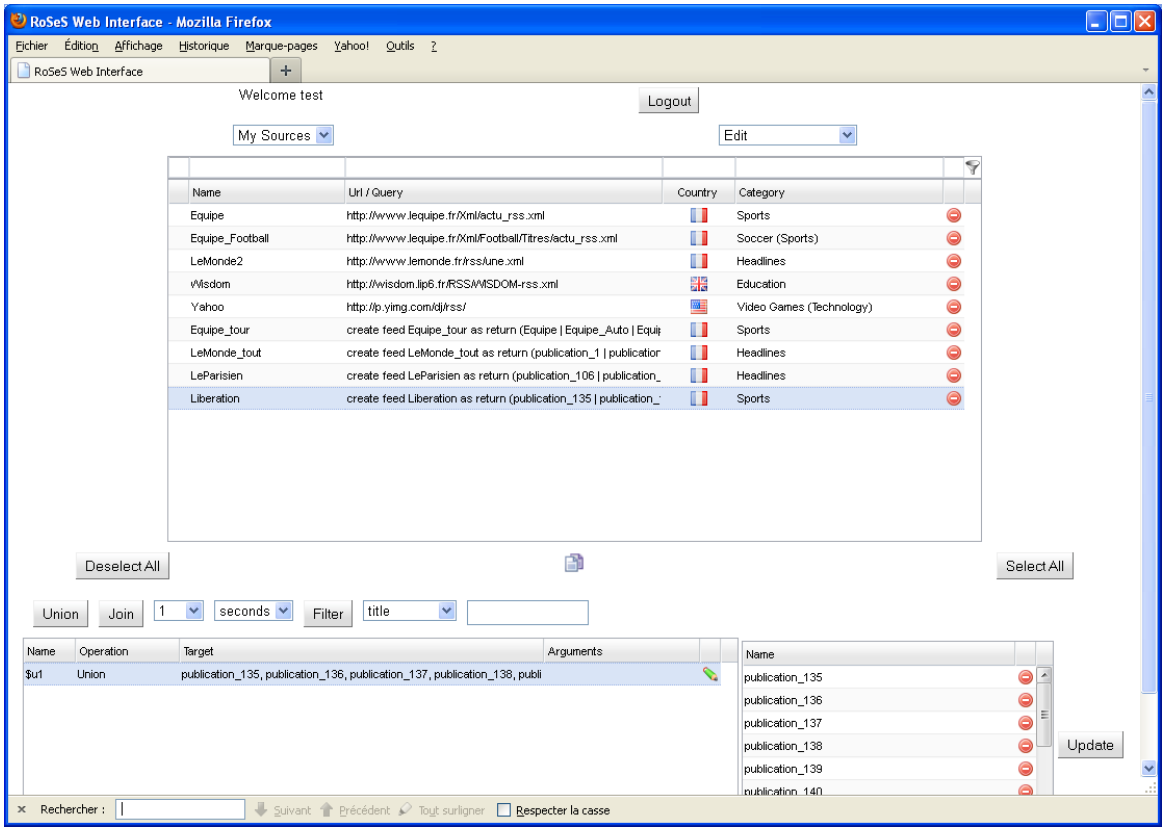


Figure 4.9 – The *ROSES Query Builder* web interface

Conclusion and Future Research Directions

In this thesis we have presented *ROSES*, a large-scale RSS feed aggregation system. 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 includes as well the Steiner tree-based multi-query optimization algorithms described in Section 3.4 and the dynamic multi-query optimization strategy described in 3.5 [TATV11]. We have illustrated experimentally that the system is able to manage thousands of publications within reasonable system requirements and that the optimization phase scales well with respect to the number of queries and filtering predicates.

The *ROSES* System is not a closed framework and it might be easily adapted to other continuous data management environments like, for instance, RFID monitoring systems. Further, it provides a modular and extensible framework, enabling new multi-query optimization techniques to be developed and incorporated incrementally into the system (*e.g.*, efficient evaluation techniques for selection or join physical operators).

There exist many avenues for future work. First of all, we intend to conduct an experimental evaluation on the *runtime* optimization approach proposed in Section 3.5. These experiments would have to evaluate both kinds of events changing the global cost of a multi-query plan, *i.e.*, (i) the insertion and removal of queries and, specially, the arrival of new selection predicates with a high selectivity rate, and (ii) the evolution of feed contents changing the selectivity rates of existing filtering predicates. In both cases, when a new very selective predicate arrives or an existing predicate with a low selectivity becomes very selective (and vice versa), an existing optimal filtering tree may become obsolete and has to be replaced by a new tree which might have a completely different structure. Thus our runtime optimization strategy should be evaluated on a set of sources whose keywords present a changing frequency over the time.

On the other hand, in this work we mainly focus on the generation of optimal query plans, statically or dynamically, independently of the underlying operator scheduling policy. Whereas this is sufficient for a large class of filtering queries, it does not take account of more complex operators like windows and joins. One type of approach in this direction is to study in more detail different kinds of operator scheduling strategies for reducing memory cost and processing overload. A good scheduling strategy could take into account many criteria: queries defined on sources with a higher publication rate could take priority over others where, at the same time, publication queries subscribed by a large number of users (or used by other publication queries) could be given as well a higher priority. At the physical level we may translate this by allocating different priorities to the operators in function of these parameters: (1) the number of items in the input queue of the operator, (2) the elapsed time since last execution of the operator, or (3) the number of subscriptions to a publication that depends on the operator. The scheduling strategy should consider all these parameters together and find the optimal calibration of each one.

Yet another interesting challenge, we are currently working on, is the optimization of the query graphs in a distributed setting. The processing of the overall query graph in a distributed environment allows indeed to improve the scalability of the system. The main idea is that the global query graph is evaluated in a distributed environment, a *P2P network* for instance. In this context there are two possibilities for distributing a global plan, (i) either the query graph is partitioned and each peer handles its own partition, *i.e.*, there are not replicated computations, or (ii) computing replication is allowed, which might make sense on large networks with geographically distant peers. Anyhow, in both cases our cost model should be extended in order to include the communication costs between peers as well as the workload capabilities of each peer which would ensure an optimal load-balancing within the system. We are currently working on the design of a variant of the greedy algorithm *VCA* that enables to incorporate the fact that an operator may be executed in different peers.

Appendices

A.1 Complete extended-BNF grammar for ROSES query language

```
<publication-query> ::= "CREATE" "FEED" <publication-name>
                        "FROM" <union>
                        ( <join-clause> )*
                        [ <where-clause> ]

            <union> ::= <flow> ( "|" <flow> )*
             <flow> ::= ( <url> | <source-name> | <publication-name> | "(" <union> ")" )
                        ( "[" <selection-predicate> "]" )*
                        [ "AS" <variable> ]
              <url> ::= '"' url '"'
      <source-name> ::= identifier
 <publication-name> ::= identifier

         <variable> ::= "$" identifier

-------------------------------------------------------------------------------

      <join-clause> ::= "JOIN" <window-predicate> "ON" <union>
                        "WITH" <join-operation>
 <window-predicate> ::= <time-based>
                      | <count-based>
       <time-based> ::= "LAST" integer <time-unit>
      <count-based> ::= "LAST" integer "ITEMS"
```

```
        <time-unit> ::= "SECONDS"
                      | "MINUTES"
                      | "HOURS"
                      | "DAYS"
                      | "WEEKS"
                      | "MONTHS"
   <join-operation> ::= <variable> "[" <join-predicate> "]"
```

```
       <where-clause> ::= "WHERE" <selection-operation> ( "AND" <selection-operation> )*
<selection-operation> ::= <variable> "[" <selection-predicate> "]"
```

```
  <selection-predicate> ::= <selection-disjunction>
<selection-disjunction> ::= <selection-conjunction> ( "OR" <selection-conjunction> )*
<selection-conjunction> ::= <selection-literal> ( "AND" <selection-literal> )*
    <selection-literal> ::= <selection-atom>
                          | "NOT" <selection-atom>
       <selection-atom> ::= <contains>
                          | ...
                          | "(" <selection-predicate> ")"
            <contains> ::= <attribute> "CONTAINS" '"' string '"'
           <attribute> ::= "title"
                          | "link"
                          | "description"
                          | "author"
                          | "category"
                          | "pubDate"
                          | "keywords"
```

```
   <join-predicate> ::= <join-disjunction>
 <join-disjunction> ::= <join-conjunction> ( "OR" <join-conjunction> )*
 <join-conjunction> ::= <join-literal> ( "AND" <join-literal> )*
```

```
     <join-literal> ::= <join-atom>
                      | "NOT" <join-atom>
        <join-atom> ::= <similar>
                      | ...
                      | "(" <join-predicate> ")"
          <similar> ::= <attribute> "SIMILAR" <window-attribute>
                      | <window-attribute> "SIMILAR" <attribute>
 <window-attribute> ::= "WINDOW" "." <attribute>
```

A.2 *properties.xml*: the ROSES configuration file

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE properties SYSTEM "http://java.sun.com/dtd/properties.dtd">
<properties>
    <comment>ROSES Configuration file</comment>

    <!-- DB: MySQL webia roses -->

    <entry key="Catalog.dbms">mysql</entry>
    <entry key="Catalog.driver">com.mysql.jdbc.Driver</entry>
    <entry key="Catalog.connection">jdbc:mysql://webia.lip6.fr/roses</entry>
    <entry key="Catalog.host">webia.lip6.fr</entry>
    <entry key="Catalog.database">roses</entry>
    <entry key="Catalog.user">roses</entry>
    <entry key="Catalog.password">XXX</entry>

    <entry key="ItemsToEvaluate.blockingQueue">LinkedBlockingQueue</entry>
    <entry key="ItemsToEvaluate.poll.timeout">1</entry>
    <entry key="ItemsToEvaluate.poll.timeUnit">SECONDS</entry>

    <entry key="Dispatcher.pauseTime">1000</entry>

    <entry key="Source.pauseTime">10000</entry> <!-- physical operator -->

    <entry key="ThreadPool.corePoolSize">4</entry>
    <entry key="ThreadPool.maximumPoolSize">10</entry>
    <entry key="ThreadPool.keepAliveTime">20</entry>
```

```
<entry key="ThreadPool.timeUnit">SECONDS</entry>
<entry key="ThreadPool.queuingStrategy">Unbounded queues</entry>

<!-- Random | RoundRobin -->
<entry key="Scheduler">RoundRobin</entry>
<entry key="Scheduler.ralentiTime">1</entry>
<entry key="Scheduler.pauseTime">1000</entry>

<entry key="BufferedQueue.bufferSize">1000</entry>

<entry key="Publication.packetSize">1000</entry>

<entry key="ItemsToPublish.blockingQueue">LinkedBlockingQueue</entry>
<entry key="ItemsToPublish.poll.timeout">1</entry>
<entry key="ItemsToPublish.poll.timeUnit">SECONDS</entry>

<!-- TXT | XML | RSS | LOG | NOOP -->
<entry key="Outputter">RSS</entry>

<entry key="Outputter.directory">/web/creusj/public_html/roses/output</entry>

<entry key="Outputter.url">http://webia.lip6.fr/~creusj/roses/output</entry>
<entry key="Outputter.xsl">http://webia.lip6.fr/~creusj/roses/xsl/htmljoin.xsl</entry>

<entry key="XMLizer.directory">xml</entry>
<entry key="GraphMLizer.directory">graphml</entry>

<!-- Yes | No -->
<entry key="dottify">Yes</entry>
<entry key="DotGenerator.directory">dot</entry>
<entry key="GraphViz.dot">/usr/bin/dot</entry>

<!-- None | Steiner | VCA | OVCA -->
<entry key="Optimizer.algorithm">VCA</entry>

<entry key="SubsumptionGraph.feedRate">1</entry>
<entry key="CostEstimator.unknownSelectivity">0.01</entry>
```

```
    <entry key="DynamicTaskExecutor.pauseTime">1000</entry>

    <entry key="RuntimeOptimizer.checkTreeCostDivergenceEvery">10</entry>
    <entry key="RuntimeOptimizer.smoothingFactor">0.5</entry>
    <entry key="RuntimeOptimizer.divergenceThreshold">0.2</entry>
    <!-- All | OnlyConcernedSources | OnlyConcernedSourcesIfDontHaveNewPredicates -->
    <entry key="Reoptimizer.strategy">All</entry>

    <entry key="UserSimulator.ralentiTime">1000</entry>
</properties>
```

A.3 Résumé en français

Les nouvelles technologies Web 2.0 ont transformé le Web en un espace vivant de partage d'information où les internautes sont devenus eux-mêmes collecteurs et créateurs de données. Le contenu du Web 2.0 évolue rapidement et produit en continu de grandes quantités de flux d'information. La prolifération de contenus générés par les applications Web 2.0 donne de nouvelles opportunités de collecte, de filtrage, d'agrégation et de partage de ces flux. Au vu de la quantité et de la diversité d'informations générées quotidiennement par le Web 2.0, il devient indispensable de développer des outils pour traiter de manière efficace les données et pour permettre aux utilisateurs de filtrer et d'agréger des informations intéressantes et personnalisées.

Les formats de syndication RSS et Atom ont émergé comme un moyen standardisé pour délivrer « en temps réel » les mises-à-jour du contenu du Web. Le format RSS (ou Atom) représente les flux d'information sous forme de documents XML contenant une liste de simples résumés textuels (items) avec des liens vers le contenu publié. En souscrivant aux ressources RSS/Atom, les lecteurs rafraîchissent régulièrement ces documents et reçoivent ainsi des flux d'items qui leur permettent de rester informés en continu. La plupart des blogs personnels, des portails d'actualités ou des forums de discussions utilisent les flux RSS/Atom pour compléter la recherche par mots-clés et la navigation entre les pages Web avec un mode de diffusion plus ciblé et efficace. De plus, les réseaux sociaux (*ex : Facebook, Twitter, Flickr*) utilisent également RSS pour notifier les utilisateurs de nouvelles informations publiées par leurs amis (ou *followers*).

Les agrégateurs RSS/Atom comme *GoogleReader*[1], *Yahoo! Pipes*[2], ainsi que *feedrinse.com,*

1. www.google.com/reader
2. pipes.yahoo.com

newsgator.com, bloglines.com, ou *pluck.com* permettent aux utilisateurs de personnaliser leurs souscriptions avec un ensemble de mots-clés. De la même manière, les systèmes d'alerte de *Google*[3] et de *Yahoo!*[4] permettent de filtrer et de notifier par courriel le contenu auquel ils ont souscrit. Toutefois, tous les agrégateurs RSS/Atom existants réalisent ce filtrage d'une manière non continue, soit en interrogeant un moteur de recherche sous-jacent où en interrogeant directement les flux au moment où l'utilisateur se connecte. Ce fonctionnement ne permet pas de garantir une diffusion performante et équilibrée des milliards de flux produits par le Web2.0. On estime ainsi qu'il existe plus de 60 millions de blogs (sans compter les forums), comparé aux 5000 sources d'information professionnelle qui sont rafraîchies périodiquement par *Google* et *Yahoo!*. De plus, des estimations officielles récentes donnent un total de presque 1 milliard de comptes utilisateurs sur des réseaux sociaux comme Facebook. Ces chiffres gigantesques illustrent le besoin croissant d'outils nouveaux et performants qui soient capables de traiter en temps réel des millions de flux d'informations sur le web.

Dans cette thèse, nous présentons *ROSES* (*Really Open Simple and Efficient Syndication*), un cadre générique pour l'interrogation et l'agrégation de flux RSS basés sur leur contenu. Il repose sur une approche orientée données, capable de supporter des expressions de requêtes continues (sélections, jointures, unions) sur des informations provenant de flux textuels et factuels. *ROSES* permet également de publier des vues RSS fusionnant et filtrant un grand nombre de flux. Ces vues peuvent être réutilisées pour construire de nouveaux flux et le résultat est un graphe acyclique avec des requêtes d'unions et de filtrages sur les flux impliqués. La taille de ces graphes de requêtes peut croître très rapidement (en nombre de sources et d'opérateurs), ce qui nécessite des stratégies d'optimisation multi-requête efficaces pour réduire le coût d'évaluation.

ROSES se base sur un modèle de données simple et sur un langage de requêtes expressif pour définir des requêtes continues sur des flux RSS. Combiné à des stratégies de récupération de données efficaces et des techniques d'optimisation multi-requête, *ROSES* peut être utilisé comme un intergiciel de flux RSS continus produisant des sources dynamiques d'information. Les principales contributions de ce travail sont :

- Un langage déclaratif pour l'agrégation et la publication de larges collections RSS sous forme de requêtes/vues ;
- Une algèbre RSS extensible pour construire des plans d'exécution multi-requête continus pour les flux RSS ;
- Une stratégie d'optimisation multi-requête efficace et passant à l'échelle en termes de

3. www.google.com/alerts
4. alerts.yahoo.com

sources et de publications ;

- Un prototype d'exécution basé sur un moteur d'exécution multi-processus asynchrone.

Le langage et l'algèbre que nous proposons dans cette thèse ont fortement été influencés par les travaux de [GÖ03] sur lesquels nous proposons un cadre plus flexible et plus adapté au contenu de la syndication du Web, ainsi qu'un nouvel opérateur de *jointure d'annotation* que nous introduisons par la suite. Quant à la stratégie d'optimisation multi-requête efficace, elle se base sur les travaux de [SG90] avec l'algorithme de factorisation *HA*, et sur l'optimisation d'arbres de *Steiner* [CD02] qui nous a amené à proposer les algorithmes VCA et VCA+VCB que nous développons lors de la factorisation des requêtes *ROSES*.

A.3.1 Le langage ROSES

Les principales fonctionnalités offertes par le langage de manipulation de données *ROSES* sont *l'enregistrement* de nouveaux flux sources (flux de données produits par le module d'acquisition), *la publication* de flux virtuels définis par des requêtes et *la souscription* à des flux (flux produit par manipulation et transformation d'autres flux). Nous présentons en premier lieu le langage de publication de flux, suivi d'une brève présentation de la manière dont on peut enregistrer des flux sources et souscrire à des flux sources ou virtuels.

Le langage de publication *ROSES* a été conçu pour atteindre plusieurs objectifs : être expressif mais simple d'utilisation, faciliter l'expression des opérations d'agrégation et de filtrage les plus courantes, tout en étant adapté à l'optimisation, afin de supporter des systèmes de syndication web à grande échelle. La forme d'agrégation utilisée le plus souvent est l'union d'items issus d'un grand nombre de flux RSS, filtrés ensuite par des conditions booléennes concernant surtout le contenu textuel. En dehors de ces opérations, une caractéristique originale du langage de publication *ROSES* est la possibilité d'exprimer des (semi-)jointures entre les items de différents flux, tout en gardant la trace des items correspondants sous forme d'annotations. En résumé, une requête de publication contient trois clauses : (a) Une clause `from` obligatoire, qui spécifie les *flux primaires*, c'est-à-dire les flux d'entrée qui vont produire les items du *flux virtuel* ainsi défini ; (b) Zéro, une ou plusieurs clauses `join`, chacune spécifiant une jointure avec un *flux secondaire* ; (c) Une clause `where` optionnelle pour les conditions de filtrage sur les flux primaires ou secondaires.

Considérons par exemple que Bob souhaite organiser une sortie avec ses amis pour aller à un concert rock. Il publie un flux virtuel appelé *RockConcertStream*, avec des items à propos de concerts, issus des flux publiés par ses amis (*FriendsFacebookStream* et *FollowedTwitterStream*), ainsi que des annonces de concerts rock publiées par le flux *EventAnnounces*. À noter que la

clause `from` permet de construire des groupes imbriqués de flux (unions identifiées par des variables), qui peuvent être référencés dans la clause `where` pour y attacher des conditions de filtrage. Ici, les conditions de filtrage sont exprimées sur le flux individuel *EventAnnounces*, représenté par la variable $ca (title contains "rock"), et sur le groupe de tous les flux de la clause `from` représenté par la 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"];
```

Ensuite Bob, qui est un fan du groupe rock Muse, publie un flux virtuel *MusePhotoStream* avec des items à propos de ce groupe, annotés avec les photos correspondantes. Les items viennent des flux *RockConcertStream* (ceux parlant de "Muse") et *MuseNews*, tandis que les photos sont issues de deux flux secondaires : *FriendsPhotos*, avec des photos publiées par ses amis et *MusicPhotos* (seulement pour la catégorie "rock").

L'annotation de flux est réalisée par l'opération de jointure ; dans notre exemple, chaque item du flux principal ($main) est annoté avec les photos des items des trois derniers mois du flux secondaire ayant des titres similaires. À noter que la jointure spécifie une opération de fenêtrage sur un groupe de flux secondaires, un flux principal (à travers une variable) et un prédicat de jointure.

```
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"];
```

Le langage d'enregistrement ROSES permet de décrire des flux sources issus soit de flux RSS/Atom existant sur le Web, soit de la matérialisation interne de flux virtuels. En particulier, pour les flux virtuels publiés à travers des requêtes il est également possible d'utiliser des feuilles de style XSLT pour transformer la structure des items. Les transformations peuvent aussi utiliser les annotations produites par les jointures, par exemple pour inclure au moment de la matérialisation les liens correspondants vers les photos du flux *MusePhotoStream*. Afin de simplifier l'optimisation, les opérations de transformation n'ont pas été incluses dans le langage de publication *ROSES* pour définir des flux virtuels. L'exemple suivant illustre la façon d'enregistrer dans *ROSES* un flux source (*http://muse.mu/rss/news*) et de matérialiser un flux virtuel (*MusePhotoStream*) modifié par une transformation ("IncludePhotos.xsl").

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

Le langage de souscription ROSES permet de déclarer des souscriptions à des flux sources ou virtuels. Une souscription spécifie essentiellement un flux, un mode de notification (RSS, mail, etc.), une périodicité et éventuellement une transformation d'items. Comme pour les transformations à l'enregistrement de flux virtuels matérialisés, les transformations à la souscription sont également exprimées par des feuilles de style XSLT, mais à la différence des premières le format de sortie est libre. L'exemple suivant présente deux souscriptions à la publication *RockConcertStream* : la première extrait les titres des items (transformation "Title.xsl") et les envoie par mail toutes les trois heures, la seconde produit simplement un flux RSS réactualisé toutes les dix 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;
```

A.3.2 Modèle de Données et Algèbre

Le modèle de données *ROSES* s'inspire des modèles de flux de données de l'état de l'art, tout en proposant des choix spécifiques de modélisation adaptés à la syndication RSS/Atom et à l'agrégation.

Un flux ROSES correspond soit à un *flux source* RSS/Atom existant sur le Web, soit à un *flux virtuel* publié à travers une requête dans *ROSES*. Formellement, un flux enregistré est un couple $f = (d, s)$, où d est le *descripteur de flux* et s est le flux d'*items* ROSES. Le descripteur de flux d est un n-uplet représentant les propriétés habituelles des flux RSS/Atom: titre, description, URL, etc. Les items *ROSES* représentent le contenu d'information diffusée par les flux RSS/Atom. Malgré l'adoption d'une syntaxe XML, les formats RSS et Atom sont utilisés essentiellement avec du contenu textuel non-imbriqué. Comme les extensions et la structuration XML imbriquée sont très rarement utilisées, nous avons fait le choix d'une représentation "à plat", sous forme de couples typés attribut-valeur, incluant les propriétés usuelles des items RSS/Atom, comme le titre, la description, le lien, l'auteur, la date de publication, etc. L'extensibilité peut être gérée par l'inclusion d'attributs nouveaux, spécifiques aux items *ROSES*, ce qui permet à la fois d'interroger tous les flux à travers les attributs communs et d'accéder aux attributs spécifiques (quand ils sont connus) pour les flux étendus.

Un flux d'items *ROSES* est un flux de données au sens général composé d'items *ROSES* annotés. Formellement, un flux d'items *ROSES* est un ensemble (éventuellement infini) d'*éléments*

ROSES $e = (t, i, a)$, où t est une estampille temporelle, i est un *item* ROSES et a une annotation, l'ensemble d'éléments pour une estampille donnée étant fini.

L'annotation a fait référence à tous les items des flux secondaires qui ont été joints avec l'élément e : une *annotation* est un ensemble de couples (j, A), où j est un *identifiant de jointure* et A un ensemble d'items. La sémantique de l'annotation sera détaillée par la suite lors de la description de l'opérateur de jointure.

Une *fenêtre ROSES* regroupe des sous-ensembles des items du flux qui sont valides à différents moments. Formellement, une fenêtre w sur le flux s est un ensemble fini de couples (t, I), où t est une estampille temporelle et I est l'ensemble d'items de s valides au moment t. À noter que (i) une estampille temporelle ne peut apparaître qu'une seule fois dans w et (ii) I contient seulement des items qui apparaissent dans s avant (ou au moment de) l'estampille temporelle t. On note $w(t)$ l'ensemble d'items de w pour l'estampille temporelle t. Les fenêtres sont utilisées dans *ROSES* seulement pour calculer des jointures entre flux. *ROSES* utilise deux types de fenêtres glissantes : basées sur le temps (les dernières n unités de temps) et basées sur le nombre (les derniers n items).

Les requêtes de publication de flux virtuels sont basées sur *cinq opérateurs* de composition de flux d'items *ROSES*. On distingue les *opérateurs conservateurs* (filtrage, union, fenêtrage, jointure), qui ne produisent pas de nouveaux items et ne changent pas le contenu des items d'entrée, des *opérateurs altérants* d'items (transformation). Un choix central dans la conception du langage de publication *ROSES* est de *se limiter à des opérateurs conservateurs*. Ce choix est justifié par le fait que les opérateurs conservateurs possèdent de bonnes propriétés de réécriture (commutativité, distributivité, etc.) et favorisent ainsi à la fois l'optimisation des requêtes et le caractère déclaratif du langage (toute expression algébrique pouvant être réécrite en une forme normalisée correspondant aux clauses déclaratives du langage). Les requêtes de publication sont donc traduites en opérations de filtrage, union, fenêtrage et jointure. Ensuite, des transformations peuvent être introduites, mais seulement lors de la matérialisation de flux virtuels, qui constitueront de nouveaux flux sources. À noter que les transformations peuvent utiliser les annotations de jointure et enrichir ainsi (indirectement) le pouvoir d'expression des jointures. Les requêtes de publication (pour lesquelles il existe des souscriptions) sont traduites en des expressions algébriques, comme illustré par l'exemple suivant pour *RockConcertStream* et *MusePhotoStream* (pour des raisons de place, nous utilisons des noms de flux abrégés et une syntaxe simplifiée).

$$RCS := \sigma_{'concert' \in des}(FFS \cup \sigma_{'rock' \in title} EA \cup FTS)$$
$$MPS := (\sigma_{'Muse' \in des} RCS \cup MNs) \underset{title \sim w.title}{\bowtie} \omega_{last\,3\,m}(\sigma_{cat='rock'} MP \cup FP)$$

L'opérateur de *filtrage* σ_P produit en sortie seulement les éléments du flux d'entrée qui satisfont le prédicat d'item P:

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

Un *prédicat d'item* P est une expression booléenne (utilisant des conjonctions, des disjonctions et des négations) composée de prédicats d'item atomiques, qui expriment une condition sur un attribut de l'item. En fonction du type d'attribut, les prédicats atomiques peuvent être :

- Pour des types simples : comparaison avec une valeur (égalité, inégalité) ;
- Pour des dates/temps : comparaison avec des valeurs date/temps (ou année, mois, jour, etc.) ;
- Pour du texte : opérateurs *contains* (mot(s) contenu(s) dans un attribut textuel), *similar* (texte similaire à un autre texte) ;
- Pour des liens : opérateurs *references*/*extends* (le lien référence/étend une URL ou un site), *shareslink* (l'attribut contient un lien vers un des URL dans une liste).

À noter que *ROSES* permet d'appliquer les prédicats de texte ou de liens *à tout l'item* – dans ce cas le prédicat considère tout le texte ou tous les liens des différents attributs de l'item. À remarquer aussi qu'il n'est pas possible de filtrer les éléments du flux par leur estampille temporelle ou par les annotations.

L'union produit en sortie tous les éléments des flux d'entrée :

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

Le fenêtrage produit une fenêtre glissante sur le flux d'entrée, basée sur le temps ou sur le nombre d'items, suivant la définition de la fenêtre :

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

où *spec* exprime l'estampille temporelle limite des items (fenêtres basées sur le temps), respectivement la position limite des items dans le flux (fenêtres basées sur le nombre d'items).

La jointure prend en entrée un flux principal et une fenêtre sur un flux secondaire. *ROSES*

utilise une variante conservative de l'opération de jointure, appelée *jointure d'annotation*, qui agit comme une semi-jointure (le flux principal est filtré suivant le contenu de la fenêtre), mais qui garde la provenance des items du flux secondaire qui joignent l'item principal, sous la forme d'une entrée d'annotation. À chaque jointure, on associe un identifiant qui permet de la référencer dans une annotation. Une *jointure* $\ltimes_P^j\ (s, w)$ *d'identifiant* j produit ainsi en sortie tout élément de s pour lequel le prédicat de jointure P est satisfait pour un sous-ensemble non vide I des items de la fenêtre w. La jointure rajoute à l'élément de sortie une entrée d'annotation (j, I). Plus précisément :

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

La transformation modifie chaque élément d'entrée suivant une fonction de transformation donnée :

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

μ_T est le seul opérateur altérant, dont l'utilisation est limitée à la production des résultats de souscription ou à la matérialisation de flux virtuels, comme expliqué ci-dessus.

A.3.3 Évaluation de Requêtes

L'exécution de requêtes *ROSES* consiste en une évaluation continue d'une collection de requêtes de publication. Cette collection est représentée par un *plan multi-requêtes* composé de différents opérateurs physiques qui reflètent les opérateurs algébriques présentés dans la Section A.3.2 (filtrage, union, jointure et fenêtrage). *ROSES* adopte un modèle standard d'exécution de requêtes continues [ABW06, CKSV08], où un plan d'exécution est transformé en un graphe de sources, opérateurs et publications interconnectés à l'aide de files d'attente ou par des tampons de fenêtrage.

À partir d'un ensemble de requêtes Q, un plan d'exécution peut donc être représenté sous la forme d'un graphe dirigé acyclique $G(Q)$. La Figure A.1 illustre un des plans d'exécution possibles pour trois publications $p_1 = \sigma_1(s_1 \cup s_2)$, $p_2 = (s_3 \cup s_4) \bowtie_1 \omega_1(s_5)$, $p_3 = \sigma_2(p_1 \cup s_6)$. On peut remarquer que les opérateurs de fenêtrage produisent une sortie de type différent, les tampons de fenêtrage, consommés par des opérateurs de jointure. La composition de vues par publication/souscription est illustrée par un arc qui connecte un opérateur de publication à un opérateur algébrique (p_1 est utilisé comme entrée par la publication p_3). À noter aussi que

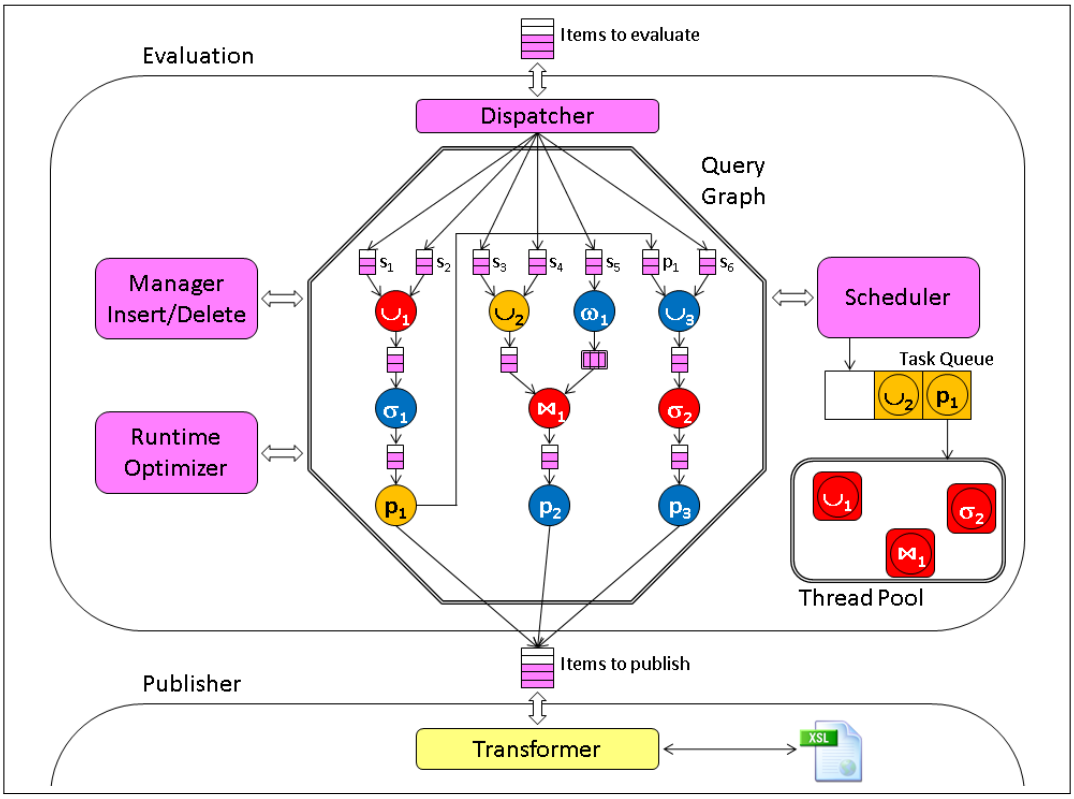


Figure A.1 – Architecture du moteur d'évaluation de requêtes *ROSES*

toutes les transformations sont appliquées après la publication.

Un plan multi-requêtes est composé d'opérateurs connectés par des files d'attente de lecture/écriture ou par des tampons de fenêtrage. De nouveaux items arrivent continuellement dans ce graphe et doivent être consommés par les différents opérateurs. À cet égard, nous avons adopté un mode d'exécution multi-tâche et en pipe-line des requêtes continues. L'exécution des requêtes est réalisée de la façon suivante. Le graphe de requêtes est surveillé par un ordonnanceur, qui décide en permanence quels opérateurs (tâches) doivent être exécutés (voir la Figure A.1). L'ordonnanceur dispose d'une réserve de threads pour exécuter en parallèle un nombre fixe de tâches [5]. La décision d'activation d'un opérateur pour son évaluation est influencée par différents facteurs dépendants du tampon d'entrée de chaque opérateur (le nombre et/ou l'âge des items dans la file d'attente d'entrée).

Dans ce contexte, nous nous basons sur un modèle de coût pour estimer les ressources (mémoire, processeur) nécessaires à l'exécution du plan. Comparés au coût estimé d'un plan classique, qui est basé sur la taille des données d'entrée, les paramètres d'estimation pour une

5. La solution naïve d'associer un thread à chaque opérateur devient rapidement inefficace/impossible à cause du surcoût de gestion des threads ou des limitations du système.

requête continue doivent refléter les caractéristiques de type flux. Nous adaptons une version simplifiée du modèle présenté dans [CKSV08] et définissons le coût de chaque opérateur *op* comme une fonction du débit de publication $R(b)$ dans son (ses) tampon(s) d'entrée b (et de la taille $S(w)$ de la fenêtre d'entrée w, pour les opérateurs de jointure).

Opérateur	**Débit de sortie**	**Mémoire**	**Coût de traitement**
$\sigma_p(b)$	$sel(p) \times R(b)$	$const$	$const \times R(b)$
$\cup(b_1, ..., b_n)$	$\sum_{1 \leq i \leq n} R(b_i)$	0	0
$\bowtie_p (b, w)$	$sel(p) \times R(b)$	$const$	$R(b) \times S(w)$
$\omega_d(b)$	0	$S = const$ ou $S = d \times R(b)$	$const$

Tableau A.1 – Le modèle de coût *ROSES*

Comme on peut le constater dans le tableau A.1, le coût de chaque opérateur dépend principalement du débit de publication $R(b)$ dans son (ses) tampon(s) d'entrée b (b_i). Nous considérons un coût d'exécution constant par item (indépendant du contenu de l'item) pour l'opérateur de filtrage, puisque les items sont des petits fragments de texte de tailles similaires. Le coût mémoire du filtrage est également constant, car un seul item est traité à la fois. Le débit de sortie de l'opérateur de filtrage correspond au débit d'entrée réduit par le facteur de sélectivité $sel(p) \in [0, 1]$, dépendant du prédicat de filtrage p.

L'union génère un débit de sortie égal à la somme des débits d'entrée [6]. Nous considérons ici des coûts mémoire et de traitement nuls car l'union, comparée aux autres opérateurs, est implémentée simplement par un ensemble d'itérateurs, un pour chaque tampon d'entrée.

L'opérateur de jointure génère un débit de sortie de $sel(p) \times R(b)$, où $R(b)$ est le débit du flux primaire d'entrée et $sel(p) \in [0, 1]$ correspond à la probabilité qu'un item de b joigne un item de la fenêtre w par le prédicat de jointure p. Ceci est la conséquence du comportement de la jointure d'annotation : un item est produit par l'opérateur de jointure quand un nouvel item arrive dans le flux principal et correspond à au moins un des items de la fenêtre. Ensuite, l'item est annoté avec tous les items correspondants dans la fenêtre. Ainsi, le coût de traitement de l'opérateur de jointure est $R(b) \times S(w)$, où $S(w)$ est la taille de la fenêtre de jointure w.

L'opérateur de fenêtrage transforme le flux en un tampon de fenêtre dont la taille dépend du nombre d'items spécifié (fenêtres basées sur le nombre) ou de la durée d et du débit d'entrée $R(b)$ (fenêtres basées sur le temps).

6. Dans l'implémentation courante l'union n'élimine pas les doublons. Pour éliminer les doublons, nous devons rajouter une fenêtre glissante de taille fixe et vérifier pour chaque nouvel item s'il apparaît déjà dans cette fenêtre.

On remarque aisément que le coût global du plan d'exécution (la somme des coûts de tous les opérateurs) est influencé principalement par l'ordre des opérateurs et par le débit d'entrée de chaque opérateur. Nous décrivons dans la section suivante la façon de réduire ce coût en poussant les filtrages et les jointures vers les flux sources du plan d'exécution.

A.3.4 Optimisation de requêtes

La principale originalité de notre approche par rapport à d'autres solutions d'optimisation multi-requêtes consiste en l'utilisation explicite d'un modèle de coût pour requêtes continues. Le modèle de coût *ROSES* (Tableau A.1) montre que le coût d'exécution de la plupart des opérateurs est proportionnel au débit d'entrée. Nous exploitons deux idées principales, communes à d'autres approches d'optimisation, mais guidées dans *ROSES* par le modèle de coût : (i) diminuer rapidement le débit d'entrée en appliquant tout d'abord le filtrage (ensuite la jointure) et (ii) factoriser les opérations communes parmi les publications.

Le processus d'optimisation est décomposé en deux phases principales :

- Une phase de *normalisation*, qui applique les règles de réécriture pour pousser tous les opérateurs de filtrage vers leurs flux sources ;
- Une phase de *factorisation* des prédicats de filtrage de chaque source, grâce à une nouvelle technique de factorisation basée sur le coût.

Le processus complet d'optimisation est décrit ci-dessous.

Normalisation des requêtes : Le but de la première phase est d'obtenir un plan d'exécution normalisé, où tous les filtrages sont évalués d'abord pour chaque source, avant d'évaluer les jointures et les unions. Ceci est possible en appliquant itérativement (a) la distributivité des sélections par rapport aux unions et (b) la commutativité entre filtrage et jointure (à remarquer que les opérations de filtrage ne peuvent pas être appliquées aux annotations générées par les jointures). On peut démontrer que sous une sémantique "*snapshot-reducible*" et en appliquant les règles de réécriture (a) et (b), on peut obtenir pour chaque requête (publication) un plan d'exécution équivalent sous forme d'un arbre à quatre niveaux. Le premier niveau de l'arbre (feuilles) est constitué des flux sources impliqués dans la requête, les nœuds de deuxième niveau sont les filtrages à appliquer à chaque flux source, le troisième niveau contient les opérateurs de fenêtrage/jointure évalués sur les résultats des filtrages et le niveau final (quatrième) est constitué des unions évaluées sur les résultats des filtrages et des jointures. La normalisation aplatit également tous les chemins de filtrage en cascade en prédicats élémentaires de filtrage en forme normale conjonctive. Il faut souligner que la normalisation peut augmenter le coût du graphe d'exécution résultant par rapport à celui initial. Toutefois, comme nous le verrons par la

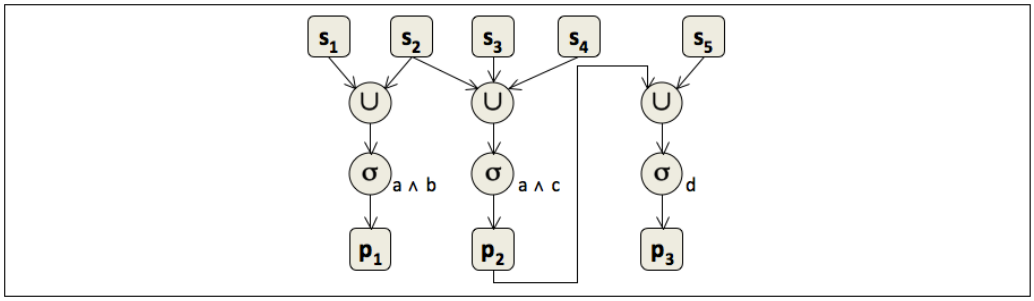


Figure A.2 – Un plan d'exécution pour les publications p_1, p_2 et p_3

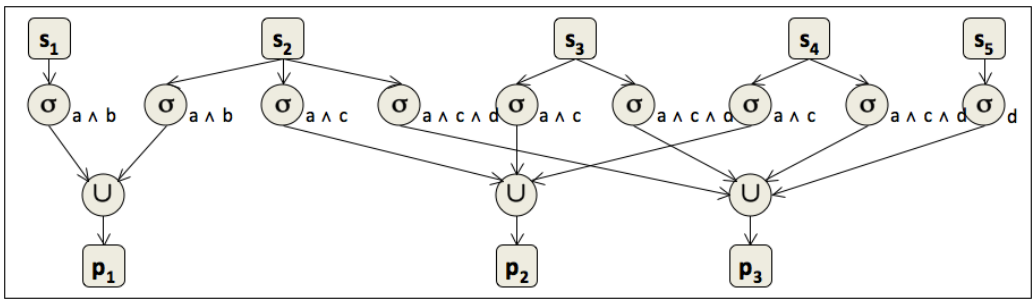


Figure A.3 – Plan d'exécution normalisé dans *ROSES*

suite, ceci est temporaire, car la phase suivante de factorisation disposera de plus d'opportunités d'optimisation.

Un exemple simple de normalisation de trois publications sans jointures est montré dans les Figures A.2 et A.3. La publication p_3 ($p_3 = \sigma_d(p_2 \cup s_5)$) est définie à partir d'une autre publication p_2, avec une opération de filtrage ($\sigma_{a \wedge c}$), tandis que les deux autres publications p_1 et p_2 sont définies sur des flux sources. Ici le processus de normalisation consiste à pousser toutes les opérations de filtrage à travers l'arbre de la publication vers les sources pour obtenir un plan normalisé tel que montré dans la Figure A.3.

Factorisation des requêtes : *La factorisation des filtrages* est la technique d'optimisation la plus efficace dans notre contexte. La normalisation génère un plan d'exécution global où chaque source s est connectée à un ensemble de prédicats de filtrage $P(s)$. La factorisation considère chaque source séparément et construit un *plan de filtrage* efficace pour chacune.

Pour trouver la meilleure factorisation on procède en deux étapes : tout d'abord on génère pour chaque flux source s un *graphe de subsomption de prédicats* qui contient *tous* les prédicats subsumant l'ensemble de prédicats $P(s)$. Le poids de chaque arc de subsomption dans ce graphe correspond au débit de sortie du nœud d'origine (source ou opérateur de filtrage) et exprime *le coût* du nœud destination. On constate aisément que tout sous-arbre de ce graphe, couvrant la source s (racine) *et tous les prédicats de* $P(s)$, correspond à un plan de filtrage équivalent au

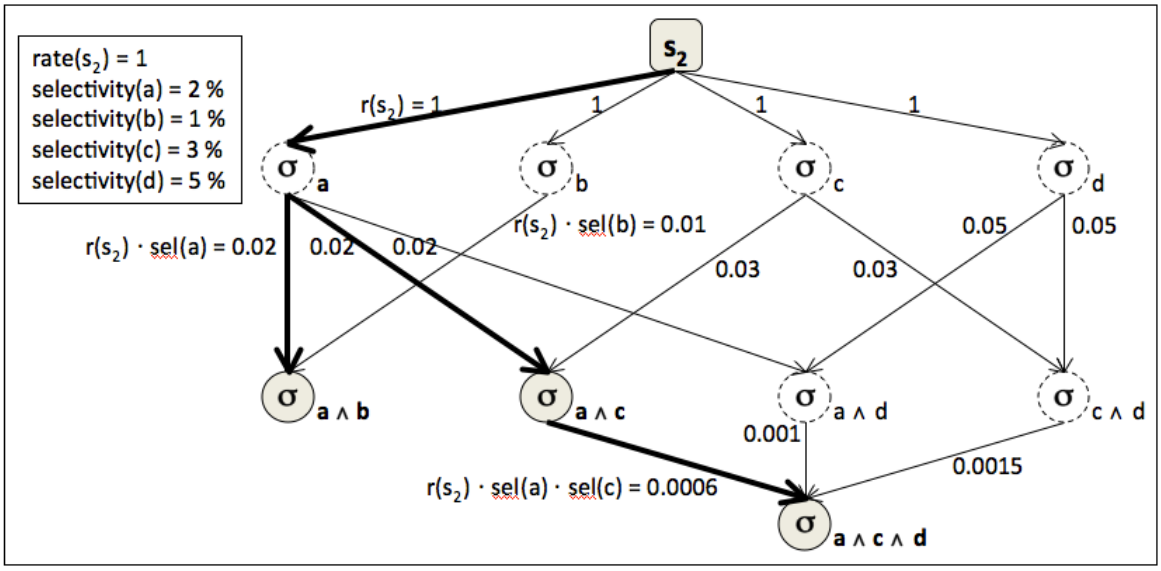


Figure A.4 – Graphe de subsomption pour s_2 et arbre de *Steiner*

plan initial. Le coût de ce plan est la somme des coûts des arcs de l'arbre.

Le problème d'optimisation dans ce contexte est de trouver un *arbre de Steiner de coût minimal* de racine s qui couvre $P(s)$, étant donné un graphe pondéré sur les arcs $G = (V, E, w)$ et un sous-ensemble $S \subseteq V$ de nœuds obligatoires (correspondant à $P(s)$). Un arbre de *Steiner* est un arbre t dans G qui relie tous les nœuds de S. Le problème d'optimisation associé aux arbres de *Steiner* est de trouver un arbre de *Steiner* de coût minimal. Si $S = V$, un arbre de *Steiner* de coût minimal correspond à un arbre couvrant de G de poids minimal.

La Figure A.4 illustre un graphe de subsomption pour le flux source s_2 (les arcs transitifs ne sont pas montrés) et un arbre de *Steiner* minimal de ce graphe (en gras).

Les opérateurs de filtrage concernant s_2 sont $\sigma_{a\wedge b}$, $\sigma_{a\wedge c}$, $\sigma_{a\wedge c\wedge d}$ (voir la Figure A.3). Le graphe de subsomption est composé de ces trois opérateurs, ainsi que de toutes leurs sous-expressions : σ_a, σ_b, σ_c, σ_d, $\sigma_{a\wedge d}$ et $\sigma_{c\wedge d}$, tandis que les arcs expriment la subsomption entre prédicats: $\sigma_a \rightarrow \sigma_{a\wedge b}$, $\sigma_b \rightarrow \sigma_{a\wedge b}$, etc. À noter que seuls les arcs de subsomption "directs" sont représentés dans la Figure A.4, bien que le graphe inclut aussi les arcs de la fermeture transitive de la relation de subsomption, par exemple $\sigma_a \rightarrow \sigma_{a\wedge c\wedge d}$. Le poids des arcs représente *la sélectivité* des prédicats d'origine, proportionnelle au débit des items arrivant de la source.

On remarque facilement que le plan de filtrage résultant est moins coûteux que le plan d'origine, malgré la normalisation qui augmente le coût du plan de filtrage en remplaçant les chemins de filtrage en cascade par une conjonction de tous les prédicats du chemin. En réalité, il peut être démontré que le graphe de subsomption régénère tous ces chemins et que le plan d'origine est toujours un sous-arbre de ce graphe. Puisque l'arbre de *Steiner* choisi est un

sous-arbre minimal pour l'évaluation de l'ensemble de prédicats initial, son coût sera au pire le coût du graphe d'origine. Par exemple, le coût de filtrage pour la source s_2 dans le plan d'origine (Figure A.2) est à peu près le double du débit de s_2 (les deux filtrages $\sigma_{a \wedge b}$ et $\sigma_{a \wedge c}$ sont appliqués à tous les items générés par s_2). Ce coût est réduit à la moitié dans l'arbre de *Steiner* final en introduisant le filtrage supplémentaire σ_a.

A.3.5 Conclusion et Perspectives

Dans cette thèse nous avons présenté *ROSES*, un système d'agrégation de flux RSS à large échelle basé sur le traitement multi-requêtes en continu et son optimisation. Nos principales contributions sont un langage d'agrégation simple et expressif et une algèbre pour RSS, que nous avons combiné à une optimisation multi-requêtes efficace. L'intégralité de l'architecture de *ROSES*, le langage d'agrégation de flux et l'algèbre de requêtes continues ont été implémentés [CATV10]. Ce prototype intègre également la stratégie d'optimisation multi-requêtes basée sur l'arbre de *Steiner* que nous avons décrit dans la section A.3.4.

Il existe des nombreux défis pour les travaux futurs. Actuellement, nous sommes en train de travailler sur l'optimisation des graphes de requêtes dans des environnements répartis. Le traitement du graphe global dans un environnement réparti permet d'améliorer le passage à l'échelle du système. L'idée principale est que le graphe de requêtes est évalué dans un environnement réparti comme un réseau pair-à-pair. Dans ce contexte, il y a deux possibilités pour répartir un plan global, (i) soit par partitionnement du graphe de requêtes et dans ce cas chaque pair gère sa propre partition, c.-à-d., il n'y a pas de la duplication, (ii) soit en permettant la duplication des traitements, qui devient intéressant dans des réseaux très larges où les pairs sont très éloignés les uns des autres. Dans les deux approches, nous devrons étendre notre modèle de coût pour inclure le coût de communication entre les pairs, mais aussi la charge de travail de chaque pair. Ceci est nécessaire pour garantir un équilibrage de charge optimal dans tout le système.

Bibliography References

[ABW06] Arvind Arasu, Shivnath Babu, and Jennifer Widom. The CQL continuous query language: semantic foundations and query execution. *The VLDB Journal*, 15(2):121–142, June 2006. 16, 21, 44, 114

[ACc+03] Daniel J. Abadi, Don Carney, Ugur Çetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Michael Stonebraker, Nesime Tatbul, and Stan Zdonik. Aurora: a new model and architecture for data stream management. *The VLDB Journal*, 12(2):120–139, August 2003. 15, 17

[AH00] Ron Avnur and Joseph M. Hellerstein. Eddies: continuously adaptive query processing. *SIGMOD Rec.*, 29(2):261–272, May 2000. 19

[AM07] Serge Abiteboul and Bogdan Marinoiu. Distributed monitoring of peer to peer systems. In *Proceedings of the 9th annual ACM international workshop on Web information and data management*, WIDM '07, pages 41–48, New York, NY, USA, 2007. ACM. 20

[AMZ07] Serge Abiteboul, Ioana Manolescu, and Spyros Zoupanos. OptimAX: optimizing distributed continuous queries. In *BDA 2007*, Marseille, France, October 2007. 19

[AMZ08] Serge Abiteboul, Ioana Manolescu, and Spyros Zoupanos. OptimAX: Optimizing Distributed ActiveXML Applications. In Daniel Schwabe, Francisco Curbera, and Paul Dantzig, editors, *ICWE*, pages 299–310. IEEE, 2008. 19

[Ara06] Arvind Arasu. *Continuous Queries over Data Streams*. PhD thesis, Stanford University, February 2006. 39

[ASS+99] Marcos K. Aguilera, Robert E. Strom, Daniel C. Sturman, Mark Astley, and Tushar D. Chandra. Matching events in a content-based subscription system. In *Proceedings of the eighteenth annual ACM symposium on Principles of distributed computing*, PODC '99, pages 53–61, New York, NY, USA, 1999. ACM. 15

[AXYY09] Pankaj K. Agarwal, Junyi Xie, Jun Yang, and Hai Yu. Input-sensitive scalable continuous join query processing. *ACM Trans. Database Syst.*, 34:1–41, September 2009. 21, 22

[BBJ98] M.H. Bohlen, R. Busatto, and C.S. Jensen. Point-versus interval-based temporal data models. In *Data Engineering, 1998. Proceedings., 14th International Conference on*, pages 192 –200, feb 1998. 41

[BBMD03] Brian Babcock, Shivnath Babu, Rajeev Motwani, and Mayur Datar. Chain: operator scheduling for memory minimization in data stream systems. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, SIGMOD '03, pages 253–264, New York, NY, USA, 2003. ACM. 17

[BDE+97] Claudio Bettini, Curtis E. Dyreson, William S. Evans, Richard T. Snodgrass, and Xiaoyang Sean Wang. A Glossary of Time Granularity Concepts. In *Temporal Databases, Dagstuhl*, pages 406–413, 1997. 34

[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. 120

[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, Philadelphia, PA, USA, 1998. Society for Industrial and Applied Mathematics. i, iii, 6, 25, 64, 65, 69

[CCD+03a] 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. 19, 21

[CCD+03b] Sirish Chandrasekaran, Owen Cooper, Amol Deshpande, Michael J. Franklin, Joseph M. Hellerstein, Wei Hong, Sailesh Krishnamurthy, Samuel R. Madden, Fred Reiss, and Mehul A. Shah. TelegraphCQ: continuous dataflow processing. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, SIGMOD '03, pages 668–668, New York, NY, USA, 2003. ACM. 16

[CCdM+] C. Constantin, J. Creus, C. du Mouza, R. Horincar, and N. Travers. D2.1 State-of-the art of XML data stream models, Livrable 2.1 ANR RoSeS. Technical Report CEDRIC-09-1799, CEDRIC laboratory, CNAM-Paris, France. 14

[CD02] X. Cheng and D.Z. Du. *Steiner Trees in Industry*, volume 11 of *Combinatorial Optimization*, pages 235–279. Kluwer Academic Publishers, 2002. 25, 65, 109

[CDTW00] Jianjun Chen, David J. DeWitt, Feng Tian, and Yuan Wang. NiagaraCQ: a scalable continuous query system for Internet databases. *SIGMOD Rec.*, 29(2):379–390, May 2000. 19, 21

[CJSS03] Chuck Cranor, Theodore Johnson, Oliver Spataschek, and Vladislav Shkapenyuk. Gigascope: a stream database for network applications. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, SIGMOD '03, pages 647–651, New York, NY, USA, 2003. ACM. 16

[CKSV08] Michael Cammert, Jürgen Krämer, Bernhard Seeger, and Sonny Vaupel. A Cost-Based Approach to Adaptive Resource Management in Data Stream Systems. *IEEE Trans. Knowl. Data Eng.*, 20(2):230–245, 2008. 18, 44, 47, 114, 116

[DFFT02] Yanlei Diao, P. Fischer, M.J. Franklin, and R. To. YFilter: efficient and scalable filtering of XML documents. In *Data Engineering, 2002. Proceedings. 18th International Conference on*, pages 341 –342, 2002. 20, 21, 22

[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. 21

[DGK82] Umeshwar Dayal, Nathan Goodman, and Randy H. Katz. An extended relational algebra with control over duplicate elimination. In *Proceedings of the 1st ACM SIGACT-SIGMOD symposium on Principles of database systems*, PODS '82, pages 117–123, New York, NY, USA, 1982. ACM. 52

[DXYW+07] B. Ding, J. Xu Yu, S. Wang, L. Qin, X. Zhang, and X. Lin. Finding Top-k Min-Cost Connected Trees in Databases. In *ICDE*, pages 836–845, 2007. 25

[FKM+07] B. Fuchs, W. Kern, D. Molle, S. Richter, P. Rossmanith, and X. Wang. Dynamic Programming for Minimum Steiner Trees. *Theor. Comp. Sys.*, 41:493–500, 2007. 25

[GÖ03] L. Golab and M. T. Özsu. Issues in Data Stream Management. *SIGMOD Record*, 32(2):5–14, 2003. 109

[HAA10] R. Horincar, B. Amann, and T. Artières. Best-Effort Refresh Strategies for Content-Based RSS Feed Aggregation. In *WISE*, pages 262–270, 2010. 9, 88

[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. 21, 22

[HKC+12] Zeinab Hmedeh, Harris Kourdounakis, Vassilis Christophides, Cedric du Mouza, Michel Scholl, and Nicolas Travers. Subscription indexes for web syndication systems. In *Proceedings of the 15th International Conference on Extending Database Technology*, EDBT '12, pages 312–323, New York, NY, USA, 2012. ACM. 13, 15

[HRK+09] Mingsheng Hong, Mirek Riedewald, Christoph Koch, Johannes Gehrke, and Alan Demers. Rule-based multi-query optimization. In *Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology*, EDBT '09, pages 120–131, New York, NY, USA, 2009. ACM. xi, 21, 22, 23

[HRW92] F.K. Hwang, D.S. Richards, and P. Winter. The Steiner Tree Problem. *Annals of Discrete Mathematics*, (53), 1992. 6, 25, 65

[Ihl91a] E. Ihler. Bounds on the quality of approximate solutions to the group Steiner problem. In Rolf Möhring, editor, *Graph-Theoretic Concepts in Computer Science*, volume 484 of *LNCS*, pages 109–118. Springer Berlin / Heidelberg, 1991. 25

[Ihl91b] E. Ihler. The Complexity of Approximating the Class Steiner Tree Problem. In *In Graph-Theoretic Concepts in Computer Science WG91*, pages 85–96. Springer, 1991. 25

[KKMZ12] Konstantinos Karanasos, Asterios Katsifodimos, Ioana Manolescu, and Spyros Zoupanos. ViP2P: Efficient XML Management in DHT Networks. In *ICWE - 12th International Conference on Web Engineering*, Berlin, Allemagne, July 2012. Springer. 20

[KMB81] L. Kou, G. Markowsky, and L. Berman. A fast algorithm for Steiner trees. *Acta Informatica*, 15:141–145, 1981. 25

[KP05] Georgia Koloniari and Evaggelia Pitoura. Peer-to-peer management of XML data: issues and research challenges. *SIGMOD Rec.*, 34(2):6–17, June 2005. 20

[Krä07] Jürgen Krämer. *Continuous queries over data stream - semantics and implementation*. PhD thesis, 2007. 34

[Krä09] Jürgen Krämer. Continuous Queries over Data Streams - Semantics and Implementation. In Johann Christoph Freytag, Thomas Ruf, Wolfgang Lehner, and Gottfried Vossen, editors, *BTW*, volume 144 of *LNI*, pages 438–448. GI, 2009. 18

[KS04] Jürgen Krämer and Bernhard Seeger. PIPES: a public infrastructure for processing and exploring streams. In *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, SIGMOD '04, pages 925–926, New York, NY, USA, 2004. ACM. 16

[LCVA02] W. Li, K. Selcëuk C, Q. Vu, and D. Agrawal. Query relaxation by structure and semantics for retrieval of logical web documents. *TKDE*, 14:768–791, 2002. 25

[LP08] Kostas Lillis and Evaggelia Pitoura. Cooperative XPath caching. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, SIGMOD '08, pages 327–338, New York, NY, USA, 2008. ACM. 20

[LPRY08] Zhen Liu, Srinivasan Parthasarathy, Anand Ranganathan, and Hao Yang. Near-optimal algorithms for shared filter evaluation in data stream systems. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data - SIGMOD '08*, page 133, New York, New York, USA, June 2008. ACM Press. 24

[MSW07] Kamesh Munagala, Utkarsh Srivastava, and Jennifer Widom. Optimization of continuous queries with shared expensive filters. *Proceedings of the twenty-sixth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems - PODS '07*, page 215, 2007. 23, 24

[MWA+02] Rajeev Motwani, Jennifer Widom, Arvind Arasu, Brian Babcock, Shivnath Babu, Mayur Datar, Gurmeet Manku, Chris Olston, Justin Rosenstein, and Rohit Varma. Query Processing, Resource Management, and Approximation in a Data Stream Management System. Technical Report 2002-41, Stanford InfoLab, 2002. 16

[MZ09] Ioana Manolescu and Spyros Zoupanos. XML materialized views in P2P networks. In *in "Fourth International Workshop on Database Technologies for Handling XML Information on the Web, Russie Saint Petersburg", 2009, http://hal.inria.fr/inria-00425627/en/. Scientific Books (or Scientific Book chapters*, 2009. 20

[NACP01] Benjamin Nguyen, Serge Abiteboul, Grégory Cobena, and Mihaí Preda. Monitoring XML data on the Web. In *Proceedings of the 2001 ACM SIGMOD international conference on Management of data*, SIGMOD '01, pages 437–448, New York, NY, USA, 2001. ACM. 20, 21

[NDM+01] Jeffrey Naughton, David Dewitt, David Maier, Ashraf Aboulnaga, Jianjun Chen, Leonidas Galanis, Jaewoo Kang, Rajasekar Krishnamurthy, Qiong Luo, Naveen Prakash, Ravishankar Ramamurthy, Jayavel Shanmugasundaram, Feng Tian, Kristin Tufte, and Stratis Viglas. The Niagara Internet Query System. *IEEE Data Engineering Bulletin*, 24:27–33, 2001. 19

[OU05] Selma Ayşe Özalp and Özgür Ulusoy. Effective early termination techniques for text similarity join operator. In *Proceedings of the 20th international conference on Computer and Information Sciences*, ISCIS'05, pages 791–801, Berlin, Heidelberg, 2005. Springer-Verlag. 49

[PAP+03] Evaggelia Pitoura, Serge Abiteboul, Dieter Pfoser, George Samaras, and Michalis Vazirgiannis. DBGlobe: a service-oriented P2P system for global computing. *SIGMOD Rec.*, 32(3):77–82, September 2003. 20

[PFL+00] João Pereira, Françoise Fabret, François Llirbat, Radu Preotiuc-Pietro, Kenneth A. Ross, and Dennis Shasha. Publish/Subscribe on the Web at Extreme Speed. In *Proceedings of the 26th International Conference on Very Large Data*

Bases, VLDB '00, pages 627–630, San Francisco, CA, USA, 2000. Morgan Kaufmann Publishers Inc. 14, 29

[RDS+04] Elke A. Rundensteiner, Luping Ding, Timothy Sutherland, Yali Zhu, Brad Pielech, and Nishant Mehta. CAPE: continuous query engine with heterogeneous-grained adaptivity. In *Proceedings of the Thirtieth international conference on Very large data bases - Volume 30*, VLDB '04, pages 1353–1356. VLDB Endowment, 2004. 15

[Sel88] Timos K. Sellis. Multiple-query optimization. *ACM Trans. Database Syst.*, 13(1):23–52, March 1988. 5, 21

[SG90] T. Sellis and S. Ghosh. On the Multiple-Query Optimization Problem. *IEEE Trans. on Knowl. and Data Eng.*, 2(2):262–266, June 1990. 21, 22, 109

[SJS00] G. Slivinskas, C.S. Jensen, and R.T. Snodgras. Query plans for conventional and temporal queries involving duplicates and ordering. In *Data Engineering, 2000. Proceedings. 16th International Conference on*, pages 547 –558, 2000. 41

[SJS01] Giedrius Slivinskas, Christian S. Jensen, and Richard Thomas Snodgrass. A Foundation for Conventional and Temporal Query Optimization Addressing Duplicates and Ordering. *IEEE Trans. on Knowl. and Data Eng.*, 13(1):21–49, January 2001. 18

[SW04] Utkarsh Srivastava and Jennifer Widom. Flexible time management in data stream systems. In *Proceedings of the twenty-third ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, PODS '04, pages 263–274, New York, NY, USA, 2004. ACM. 34

[TATV11] Jordi Creus Tomàs, Bernd Amann, Nicolas Travers, and Dan Vodislav. RoSeS: a continuous query processor for large-scale RSS filtering and aggregation. In *CIKM*, pages 2549–2552, 2011. 101

[TGNO92] Douglas Terry, David Goldberg, David Nichols, and Brian Oki. Continuous queries over append-only databases. *SIGMOD Rec.*, 21(2):321–330, June 1992. 21

[TRFZ07] W. Tan, F. Rao, Y. Fan, and J. Zhu. Compatibility Analysis and Mediation-Aided Composition for BPEL Services. In *DASFAA*, pages 1062–1065, 2007. 21, 22

[URLa] Google Reader is available at `http://wwww.google.com/reader`. 10

[URLb] Yahoo! Pipes is available at `http://pipes.yahoo.com`. 11

[URLc] Digg is available at `http://digg.com`. 12

[URLd] NetVibes is available at http://www.netvibes.com. 12

[URLe] Feedzilla is available at http://www.feedzilla.com. 12

[URLf] NewsIsFree is available at http://www.newsisfree.com. 12

[VN02] Stratis D. Viglas and Jeffrey F. Naughton. Rate-based query optimization for streaming information sources. In *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, SIGMOD '02, pages 37–48, New York, NY, USA, 2002. ACM. 19

[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. 21, 22

[Win87] P. Winter. Steiner problem in networks: A survey. *Networks*, 17(2):129–167, 1987. 25

[YGM99] Tak W. Yan and Hector Garcia-Molina. The SIFT information dissemination system. *ACM Trans. Database Syst.*, 24(4):529–565, December 1999. 9, 15

[ZK02] L. Zosin and S. Khuller. On directed Steiner trees. In *ACM-SIAM - SODA*, pages 59–63, 2002. 25