

Search Computing Challenges and Directions

S. Ceri, M. Brambilla eds.

LNCS State-of-Art Survey

Volume Editors

Stefano Ceri

Politecnico di Milano
Dipartimento di Elettronica e Informazione
P.za L. Da Vinci, 32
20133 Milano, Italy
email: stefano.ceri@polimi.it
phone: +39 02 2399 3532
fax: +39 02 2399 3411

Marco Brambilla

Politecnico di Milano
Dipartimento di Elettronica e Informazione
P.za L. Da Vinci, 32
20133 Milano, Italy
email: marco.brambilla@polimi.it
phone: +39 02 2399 7621
fax: +39 02 2399 7321

Preface

Who are the strongest European competitors on software ideas? Who is the best doctor to cure insomnia in a nearby hospital? Where can I attend an interesting conference in my field close to a sunny beach? This information is available on the Web, but no software system can accept such queries nor compute the answer. At most, users can identify sub-problems that can be addressed by specific search engines and interact with each of them serially, but then they have the responsibility of building global answers by manually composing results. Search Computing is a new multi-disciplinary discipline which will provide the abstractions, methods, tools and computing systems required to express these queries and to build their answer.

The emerging paradigm of software services has so far been neutral to search. Search Computing is an evolution of service computing focused on building the answers of complex queries by interacting with a constellation of cooperating search services, using ranking as the dominant factor for service composition. New language and description paradigms are required for interconnecting services and for expressing queries. Semantic domain knowledge helps enriching terminological knowledge about objects being searched. New protocols help capturing ranking preferences and their refinement; new interfaces present complex results with simple visual descriptions. Ranking is relative to individuals and context and therefore reflects personal and social contributions. Economical and legal implications of Search Computing must be understood and mastered. In summary, Search Computing is a multi-disciplinary effort which requires adding to sound software principles contributions from other sciences such as knowledge representation, human-computer interfaces, psychology, sociology, economical and legal sciences.

The Search Computing (Seco) Project is funded by the European Research Council (ERC), responding to the 2008 Call for “IDEAS Advanced Grants”, a program dedicated to the support of investigation-driven frontier research. SeCo started on November 1st, 2008 and will last until October 31, 2013 (see www.search-computing.eu.) This book describes the outcome of the first SeCo “Workshop on Search Computing Challenges and Directions”, held in Como on June 17-19, 2009.

The book is divided in three parts. The **first part** gives **visions** of the current evolution in search, which is becoming more and more task-oriented and is now starting to use ontological knowledge in order to manage complex queries; these visions are marking the new trends in search.

The **second part** will present some **background and related technologies**. These can be considered as parallel fields of research, useful both for setting the theoretical premises for Search Computing and for providing a technological framework for building Search Computing systems and applications.

The **third part** dwells into the **technological problems and issues** which arise when dealing with Search Computing as a new search paradigm. It provides a unified

view of the results of Search Computing as achieved exactly after one year since its starting date.

The book is the result of a collective effort of all the project participants and has been reviewed with the help of the project's advisory board members and of several other experts. We thank all of them for their effort.

Stefano Ceri and Marco Brambilla

Milano, January 2nd, 2010

Table of Contents

Preface	1
PART I: Visions	3
Chapter 1: Search Computing	4
<i>Stefano Ceri</i>	
Chapter 2: Next Generation Web Search	12
<i>Ricardo Baeza-Yates, Prabhakar Raghavan</i>	
Chapter 3: Search for Knowledge.....	25
<i>Gerhard Weikum</i>	
Part II: Technology Watch for Search Computing	42
Chapter 4: The Search Engine Industry	44
<i>Tommaso Buganza, Emanuele Della Valle</i>	
Chapter 5: From Mashup Technologies to Universal Integration: Search Computing the Imperative Way	71
<i>Florian Daniel, Stefano Soi, Fabio Casati</i>	
Chapter 6: Web Data Extraction.....	93
<i>Robert Baumgartner, Alessandro Campi, Marcus Herzog, Georg Gottlob</i>	
Chapter 7: Dataspaces	113
<i>Cornelia Hedeler, Khalid Belhajjame, Alessandro Campi, Suzanne Embury, Alvaro Fernandez, Norman Paton</i>	
Chapter 8: Multimedia and Multimodal Information Retrieval	134
<i>Alessandro Bozzon, Piero Fraternali</i>	
Part III: Issues in Search Computing	156
Chapter 9: Service Marts	161
<i>Alessandro Campi, Stefano Ceri, Georg Gottlob, Andrea Maesani, Stefania Ronchi</i>	
Chapter 10: Join Methods and Query Optimization	186
<i>Daniele Braga, Stefano Ceri, Michael Grossniklaus</i>	
Chapter 11: Rank-join Algorithms for Search Computing.....	210
<i>Ihab Ilyas, Davide Martinenghi, Marco Tagliasacchi</i>	

Chapter 12: Panta Rhei: A Query Execution Environment	224
<i>Daniele Braga, Stefano Ceri, Francesco Corcoglioniti, Michael Grossniklaus</i>	
Chapter 13: Liquid Queries and Liquid Results in Search Computing	244
<i>Alessandro Bozzon, Marco Brambilla, Stefano Ceri, Piero Fraternali, Ioana Manolescu</i>	
Chapter 14: Building Search Computing Applications	270
<i>Marco Brambilla, Stefano Ceri, Alessandro Bozzon, Francesco Corcoglioniti</i>	
Chapter 15: Search Computing and the Life Sciences	293
<i>Marco Masseroli, Norman W Paton, Irena Spasic</i>	
Appendixes	309
Appendix A: Search Computing Dictionary	309

Authors' Index

Baeza-Yates, Ricardo 12
Baumgartner, Robert 93
Belhajjame, Khalid 113
Bozzon, Alessandro 134, 244, 270
Braga, Daniele 186, 224
Brambilla, Marco 244, 270
Buganza, Tommaso 44
Campi, Alessandro 93, 113, 161
Casati, Fabio 71
Ceri, Stefano 4, 161, 186, 224, 244, 270
Corcoglioniti, Francesco 224, 270
Daniel, Florian 71
Della Valle, Emanuele 44
Embury, Suzanne 113
Fernandez, Alvaro 113
Fraternali, Piero 134, 244
Gottlob, Georg 93, 161
Grossniklaus, Michael 186, 224
Hedeler, Cornelia 113
Herzog, Marcus 93
Ilyas, Ihab 210
Maesani, Andrea 161
Manolescu, Ioana 244
Martinenghi, Davide 210
Masseroli, Marco 293
Paton, Norman W. 113, 293
Raghavan, Prabhakar 12
Ronchi, Stefania 161
Soi, Stefano 71
Spasic, Irena 293
Tagliasacchi, Marco 210
Weikum, Gerhard 25

Reviewers

Luciano Baresi	Politecnico di Milano
Marco Brambilla	Politecnico di Milano
Fabio Casati	Università di Trento
Tiziana Catarci	Università di Roma “La sapienza”
Stefano Ceri	Politecnico di Milano
Georg Gottlob	Oxford University
Ihab Ilyas	University of Waterloo
Ioana Manolescu	INRIA and Université de Paris Sud
Giansalvatore Mecca	Università della Basilicata
Roberto Verganti	Politecnico di Milano
Gerhard Weikum	Max-Planck Institute for Informatics

Sponsoring Institutions

The Search Computing (Seco) Project is funded by the European Research Council (ERC), responding to the 2008 Call for “IDEAS Advanced Grants”, a program dedicated to the support of investigation-driven frontier research. SeCo started on November 1st, 2008 and will last until October 31, 2013.

Part I

Visions

Chapter 1: Search Computing

Stefano Ceri

Politecnico di Milano, Dipartimento di Elettronica ed Informazione,
V. Ponzio 34/5, 20133 Milano, Italy
stefano.ceri@polimi.it

Abstract. Search Computing is a new paradigm for composing search services. While state-of-art search systems answer generic or domain-specific queries, Search Computing enables answering questions via a constellation of dynamically selected, cooperating search services, which are correlated by means of join operations. The idea is simple, yet pervasive. New language and description paradigms are required for expressing queries and for connecting services. New user interfaces and protocols help capturing ranking preferences and enabling their refinement.

Keywords: Complex queries, multi-dimensional queries, search services, join operation, data integration, data visualization, process composition.

1 Beyond Page Search

Throughout the last decade, Internet search has been primarily performed by routing users towards the specific Web page that best answered their information needs. Major search engines, such as *Google*, *Yahoo* and *Bing*, crawl the Web and index Web pages, highlighting worldwide candidate “best” pages with excellent precision and recall; such ability has proven adequate to fulfill users’ needs, to the point that Web search is customarily performed by millions of users, both for work and leisure.

However, not all information needs can be satisfied by individual pages on the surface Web. On one hand, the so-called “deep Web” contains information which is perhaps more valuable than what can be crawled on the surface Web; on another side, as the users get confident in the use of search engines, their queries become more and more complex, to the point that their formulation goes beyond what can be expressed with a few keywords, their answers require more than a list of Web pages, and general-purpose search engines perform poorly upon them. According to search company’s experts, the number of complex queries that are not answered well by major search engines due to their intrinsic complexity is remarkably high and increasing. Many search interactions can be considered as part of a more complex process of expressing goals and achieving tasks, as discussed in the vision paper by Ricardo Baeza Yates (Chapter 2).

When a query addresses a specific domain (e.g., travels, music, shows, food, movies, health, and genetic diseases), domain-specific search engines do a better job

Chapter 2: Next Generation Web Search

Ricardo Baeza-Yates¹ and Prabhakar Raghavan¹

Yahoo! Research
Barcelona, Spain & Sunnyvale, USA
rbaeza@acm.org & pragh@yahoo-inc.com

Abstract. In this chapter we provide our personal vision of what could be the next generation of Web search engines, outlining the main research challenges that derive from it. This vision is based on a single premise: people do not really want to search, they want to get tasks done. We motivate our work by the current trends in the Web and, in particular, Web search.

1 Introduction

Web search has become the starting point of many on-line user activities. The first generation of Web search engines faced two primary challenges: (1) to scale known information retrieval techniques to millions of documents, far beyond the capacity of search engines of the time; (2) to contend with document publishers that were diverse, non-uniform in quality/authority/style, and in some cases unreliable or even dishonest in their intent (known as Web spammers). The first of these challenges was addressed by employing coarse-grained parallel hardware to create robust, high-capacity computing services – these gave rise in time to what we now think of as cloud computing. The second challenge was addressed using various approaches from machine learning and link analysis, but the issue of spammers has never been completely solved. To this date, search engines fight an interactive battle with spammers using increasingly sophisticated techniques. As search engines devised better techniques to combat spam, they were able to adapt many of the same ideas to improve their ranking functions. Almost a decade ago, Google and other search engines began to perfect their responses to *navigational queries*: queries (such as “british airways”) whose goal is to take the user to a single target page (in this case, the home page of British Airways).

Navigational queries became the genesis of a new way of thinking about Web search: namely, the goal of the engine is not to retrieve relevant documents (the classical metaphor for three decades of information retrieval). Rather, the goal is to identify a user’s intent (navigating to a specific target page being one of many possible intents), and to synthesize a page that directly addresses the user’s intent. For instance, the query “Frankfurt temperature” is arguably not demanding a ranked list of websites any of which could provide the temperature in Frankfurt; rather, the user is best satisfied by a number that shows the current temperature in Frankfurt. Thus, Web search ceases to be about document

Chapter 3: Search for Knowledge

Gerhard Weikum

Max-Planck Institute for Informatics
Saarbruecken, Germany
weikum@mpi-inf.mpg.de

Abstract

There are major trends to advance the functionality of search engines to a more expressive semantic level. This is enabled by the advent of knowledge-sharing communities such as Wikipedia and the progress in automatically extracting entities and relationships from semistructured as well as natural-language Web sources. In addition, Semantic-Web-style ontologies, structured Deep-Web sources, and Social-Web networks and tagging communities can contribute towards a grand vision of turning the Web into a comprehensive knowledge base that can be efficiently searched with high precision. This vision and position paper discusses opportunities and challenges along this research avenue. The technical issues to be looked into include knowledge harvesting to construct large knowledge bases, searching for knowledge in terms of entities and relationships, and ranking the results of such queries.

1 Trends and Opportunities

It is widely believed that queries posed to Web search engines are very simple: one or two keywords to express the user's information need, and millions of matching results including many excellent hits, so that well-known ranking techniques can easily achieve high precision for the top-10 Web pages seen by the user. While this may indeed be true for the large mass of popular queries, each asked by many thousands of users, the picture is different for the long tail of individual queries about professional needs, rare hobbies, local music concerts, or personal health issues. Not only do these queries contain more keywords and return fewer results, but the user would often expect a concise answer with relevant facts rather than merely being pointed to potentially interesting Web pages. The following are examples of such advanced queries (we will later use some of these to illustrate technical challenges):

1. A student of natural history may want to know about explorers on river expeditions for some project work. A botanics student may be interested in succulents that grow in both America and Africa. While perhaps resembling quiz questions, this type of queries arises in the daily work of millions of university students.
2. As many people like watching TV game shows, true quiz questions may indeed be a use case as well. Which king was married to Eleanor of Aquitaine? Who was

Part II

Technology Watch for Search Computing

Introduction to part II

Technology Watch for Search Computing

The second part of the book presents surveys of the technologies providing foundations to Search Computing. These chapters offer state-of-the-art and research trends within strongly related fields of research, useful both for setting the theoretical premises for Search Computing, and for providing a technological framework for building Search Computing systems and applications.

Chapter 4 includes the analysis and classification of **search systems**, which are facing a time of extremely rapid development. The study will discuss a methodological framework for clustering search systems within categories; as a byproduct, the study detects decision variables and search engine features which are most likely to produce innovation and value in the search engine industry.

Chapter 5 deals with **mashup languages and systems**, a new way of describing computer processes through visual abstractions; mashup interfaces are very relevant to Search Computing, given that queries aim at the efficient interconnection of search engines and are primarily addressing expert users or developers.

Chapter 6 deals with **data extraction on the Web**, describing mechanisms for extracting information which is available in Web pages and materialize it into repositories, by capitalizing on the experience of the Lixto project; data extraction technology is essential for building and exposing data services. This chapter will also deal with monitoring Web content, alerting users when information is updated.

Chapter 7 focuses on **data spaces** as new concept for gluing loose and flexible approaches to data management, which give rise to a variety of new services for exposing data to wider usage; indeed Search Computing is primarily pursuing a data-driven approach to search service compositions and takes advantage from flexible technologies for exposing data sources.

Chapter 8 presents a review of **search technologies for multimedia content**, by showing the processes and tools for augmenting audio and video content with meta-data, so as to facilitate search upon multimedia content and its integration within search results.

Chapter 4: The Search Engine Industry

Tommaso Buganza¹ and Emanuele Della Valle²

¹ Dipartimento di Ingegneria Gestionale, Politecnico di Milano, 20133 Milano, Italy

² Dipartimento di Elettronica e Informazione, Politecnico di Milano, 20133 Milano, Italy
{tommaso.buganza | emanuele.dellavalle}@polimi.it

Abstract. In this chapter we present the main trends in the search engine industry. Being such industry technology based, its dynamics can be assessed by applying theories such as (a) dominant design, (b) complementary assets, (c) product and service architecture and (d) disruptive technologies. We dedicate the first section of this chapter to reviewing such literature and explaining how to apply it to identify trends in the search engine industry competition. As preliminary result we position the search engine industry among those that are probably entering in a new fluid phase. In this industry the Google architecture already emerged as dominant design, but after 2005 many new players entered the market (e.g. Cuil, Kosmix, Powerset, Wolfram Alpha, Bing) and most of them are not following the dominant design but are really trying to propose something radically new. Then, we present the data gathering tool we build to use analyze a sample of 26 search engines. In particular, we describe the dimensions, relevant to study the search engine industry, and the metrics for measuring the features of different search engines along those dimensions. We consider three types of metrics: (a) user based – what the user can perceive and act upon; (b) machinery related – what the search engine does internally; and (c) business model oriented – what makes the business profitable. Then we analyze the data using three methods: principal component analysis, two steps cluster analysis, and post hoc analysis on the business models categorization. We close the chapter discussing the results of our analysis.

1 Problem Setting

It is commonly recognized that the search engine industry started in 1990 with the release of the very first tool used for searching on the (pre-web) Internet: Archie. Since then, many different companies launched their own solutions in order to fulfill the market need for searching on the web. Search engines have become a usual service for the large majority of us to the point that more than 13 billion searches are made every month just in the US¹. As for many Internet related industries, though, the companies operating on this market found it difficult to transform the value they had into real cash flows. Still, nowadays the search engine industry is probably one of the

¹ http://www.comscore.com/Press_Events/Press_Releases/2009/3/US_Search_Engine_Ranking

Chapter 5: From Mashup Technologies to Universal Integration: Search Computing the Imperative Way

Florian Daniel, Stefano Soi, Fabio Casati

University of Trento - Via Sommarive 14, 38123 Trento - Italy
{daniel,soi,casati}@disi.unitn.it

Abstract. Mashups, i.e., web applications that are developed by integrating data, application logic, and user interfaces sourced from the Web, represent one of the innovations that characterize Web 2.0. Novel content wrapping technologies, the availability of so-called web APIs (e.g., web services), and the increasing sophistication of mashup tools allow also the less skilled programmer (or even the average web user) to compose personal applications on the Web. In many cases, such applications also feature search capabilities, achieved by explicitly integrating search services, such as Google or Yahoo!, into the overall logic of the composite application.

In this chapter, we first overview the state of the art in mashup development by looking at which technologies a mashup developer should master and which instruments exist that facilitate the overall development process. Then we specifically focus on our own mashup platform, *mashArt*, and discuss its approach to what we call universal integration, i.e., integration at the data, application, and user interface layer inside one and the same mashup environment. To better explain the novel ideas of the platform and its value in the context of search computing, we discuss an example inspired by the idea of search computing.

1 Introduction

The advent of Web 2.0 led to the participation of the user into the content creation and application development processes, also thanks to the wealth of social web applications (e.g., wikis, blogs, photo sharing applications, etc.) that allow users to become an active contributor of content rather than just a passive consumer, and thanks to *web mashups* [1]. Mashup tools enable fairly sophisticated development tasks inside the web browser. They allow users to develop their own applications starting from existing content and functionality. Some applications focus on integrating RSS¹ or Atom² feeds, others on integrating RESTful services [20], others on simple UI widgets, etc. Mashup approaches are innovative especially in that they tackle integration at the user interface level and do not “just” focus on data and in that they aim at simplicity more than robustness or completeness of features (up to the point to enable also non-

¹ <http://cyber.law.harvard.edu/rss/rss.html>

² <http://www.ietf.org/rfc/rfc4287.txt>

Chapter 6:

Web data extraction for service creation

Robert Baumgartner¹, Alessandro Campi²,
Georg Gottlob³, and Marcus Herzog¹

¹ Lixto Software GmbH, Favoritenstrasse 9-11, 1040 Wien, Austria
{robert.baumgartner,marcus.herzog}@lixto.com

² Politecnico di Milano, DEI, Piazza Leonardo da Vinci 32, 20133 Milano, Italy
campi@elet.polimi.it

³ Computing Laboratory, Oxford University, U.K.
gottlob@comlab.ox.ac.uk

Abstract. Web data extraction is an enabling technique in the search computing scenario. In this chapter, we first review the state of the art in wrapper technologies focusing on how wrapper generators can be used to create unified services that integrate data from Web Applications and Web services in various domains. Next, we describe the Lixto approach and we present the Lixto Suite as one example of Web Process Integration. Finally, application areas and future challenges and the usage of wrapper technologies in the search computing context is discussed.

1 Introduction

Although in today's Web much data is available via APIs, light-weight and heavy-weight Web service techniques, the larger amount of data is still only available in semi-structured formats such as HTML. In the recent years, Web pages became more complex and turned into Web Applications, using a lot of Web 2.0 and Rich Internet Application technologies. As a consequence, new research and technical challenges emerged, related to automated Web navigation and data extraction.

To use Web data in Enterprise Applications and service-oriented architectures, it is crucial to provide means for automatically turning Web Applications and Web sites into Web Services, allowing structured and unified access to heterogeneous sources. This includes to understand the logic of the Web application, to fill out form values, and to grab relevant data – all these aspects need to be reflected accordingly in the generated Web Service.

In a number of business areas, Web applications are predominant among business partners for communication and business processes. Various types of processes are carried out on Web portals, covering activities such as purchase, sales, or quality management, by manually interacting with Web sites.

Wrapper Generators enable the automation of processes and operations of Web Applications. They pave the way for *Web Process Integration*, i.e. the seamless integration of Web applications into a corporate infrastructure or service

Chapter 7:

Dataspaces

Cornelia Hedeler¹, Khalid Belhajjame¹, Norman W. Paton¹,
Alessandro Campi², Alvaro A.A. Fernandes¹, and Suzanne M. Embury¹

¹ School of Computer Science, University of Manchester, UK.
{chedeler, npaton, alvaro, embury}@cs.manchester.ac.uk

² Dipartimento di Elettronica e Informazione, Politecnico di Milano, Italy.
campi@elet.polimi.it

Abstract. The vision of dataspace is to provide various of the benefits of classical data integration, but with reduced up-front costs, combined with opportunities for incremental refinement, enabling a “pay as you go” approach. As such, dataspace join a long stream of research activities that aim to build tools that simplify integrated access to distributed data. To address dataspace challenges, many different techniques may need to be considered: data integration from multiple sources, machine learning approaches to resolving schema heterogeneity, integration of structured and unstructured data, management of uncertainty, and query processing and optimization. Results that seek to realize the different visions exhibit considerable variety in their contexts, priorities and techniques. This chapter presents a classification of the key concepts in the area, encouraging the use of consistent terminology, and enabling a systematic comparison of proposals. This chapter also seeks to identify common and complementary ideas in the dataspace and search computing literatures, in so doing identifying opportunities for both areas and open issues for further research.

1 Introduction

Data integration, in various guises, has been the focus of ongoing research in the database community for over 20 years. The objective of this activity has generally been to provide the illusion that a single database is being accessed, when in fact data may be stored in a range of different locations and managed using a diverse collection of technologies. Providing this illusion typically involves the development of a single central schema to which the schemas of individual resources are related using some form of mapping. Given a query over the central schema, the mappings, and information about the capabilities of the resources, a distributed query processor optimizes and evaluates the query.

Data integration software is impressive when it works; declarative access is provided over heterogeneous resources, in a setting where the infrastructure takes responsibility for efficient evaluation of potentially complex requests. However, in a world in which there are ever more networked data resources, data integration

Chapter 8: Multimedia and Multimodal Information Retrieval

Alessandro Bozzon and Piero Fraternali

Dipartimento di Elettronica e Informazione, Politecnico di Milano,
Piazza Leonardo da Vinci 32, 20133 Milano, Italy
{alessandro.bozzon, piero.fraternali}@polimi.it

Abstract. The Web is progressively becoming a multimedia content delivery platform. This trend poses severe challenges to the information retrieval theories, techniques and tools. This chapter defines the problem of multimedia information retrieval with its challenges and application areas, overviews its major technical issues, proposes a reference architecture unifying the aspects of content processing and querying, exemplifies a next-generation platform for multimedia search, and concludes by showing the close ties between multi-domain search investigated in Search Computing and multimodal/multimedia search.

Keywords: multimedia information retrieval, digital signal processing, video search engines, multi-modal query interfaces.

1 Introduction

The growth of digital content has reached impressive rates in the last decade, fuelled by the advent of the so-called “Web 2.0” and the emergence of user-generated content. At the same time, the convergence of the fixed-network Web, mobile access, and digital television has boosted the production and consumption of audio-visual materials, making the Web a truly multimedia platform.

This trend challenges search as we know it today, due to the more complex nature of multimedia with respect to text, in all the phases of the search process: from the expression of the user’s information need to the indexing of content and the processing of queries by search engines.

This Chapter gives a concise overview of Multimedia Information Retrieval (MIR), the long-standing discipline at the base of audio-visual search engines, and connects the research challenges in this area to the objectives and research goals of Search Computing.

MIR amplifies many of the research problems at the base of search over textual data. The grand challenge of MIR is bridging the gap between queries and content: the former are either expressed by keywords, like in text search engines, or, by extension, with non-textual samples (e.g., an image or a piece of music). Unlike in text search engines, where the query has the same format of content and can be matched almost directly to it, query processing in MIR must fill an enormous gap. To

Part III

Issues in Search Computing

Introduction to part III

Search Computing in a Nutshell

Prior to dwelling into chapters discussing Search Computing in greater detail, we give a bird's eye's view upon its various phases and components, by providing an architectural view of the Search Computing prototyping environment.

Search Computing systems support their users in asking multi-domain queries; for instance, “Where can I attend a DB scientific conference close to a beautiful beach reachable with cheap flights?”. A system *decomposes* the query into sub-queries (in this case: “Where can I attend a DB scientific conference?”; “which place is close to a beautiful beach?”; “which place is reachable from my home location with cheap flights?”) and maps each sub-query to a domain-expert server (in this case, calls to servers named “Conference”, “Tourism”, “Low-Cost-Flights”); it then *analyzes* the query and translates it into an internal format, which then is optimized, thereby yielding to an optimal *plan* for query execution; plan execution is supported by an *execution engine*, which submits service calls to services through a *service invocation framework*, builds the *query results* by combining the outputs produced by service calls, computes the *global rankings* of query results, and outputs query results in an order that reflects, although with some approximation, their global ranking.

These transformation steps are shown in the bottom-left side of Fig. 1; they are performed by the *query mapper*, *query analyzer*, *query planner*, and *execution engine*, under the responsibility of a *query orchestrator* that starts query execution and collects query results. The figure shows that each of the four modules directly accepts user-provided input through suitable interfaces; in this way, prototype implementation in Search Computing can take place bottom-up, by starting with the *execution engine*, which can execute a given plan, then adding the *query planner*, which produces the optimal plan for a given internal query, then adding the *query analyzer*, which reads an abstract query, checks that the query is legal, and produces an internal query; and finally adding a *query mapper*, capable to decompose a multi-domain query into several domain-specific queries. In this book we do not address query mapping, while we address the other steps. The Search Computing prototyping architecture is currently well-defined in terms of interactions and of functionalities; prototypes will be delivered throughout the course of the SeCo Project

Services are made available to Search Computing through a standard format, called *service mart*; by this term we mean an abstraction that masks the different implementation styles of services and is tailored to the specific need of exposing *search services* – i.e., services whose primary purpose is to produce ranked lists of results. Moreover, service marts offer a classification of service properties (that represent either the call or the result of a service invocation; given output results may represent the ranking values) and a definition of composition patterns allowing to combine service marts.

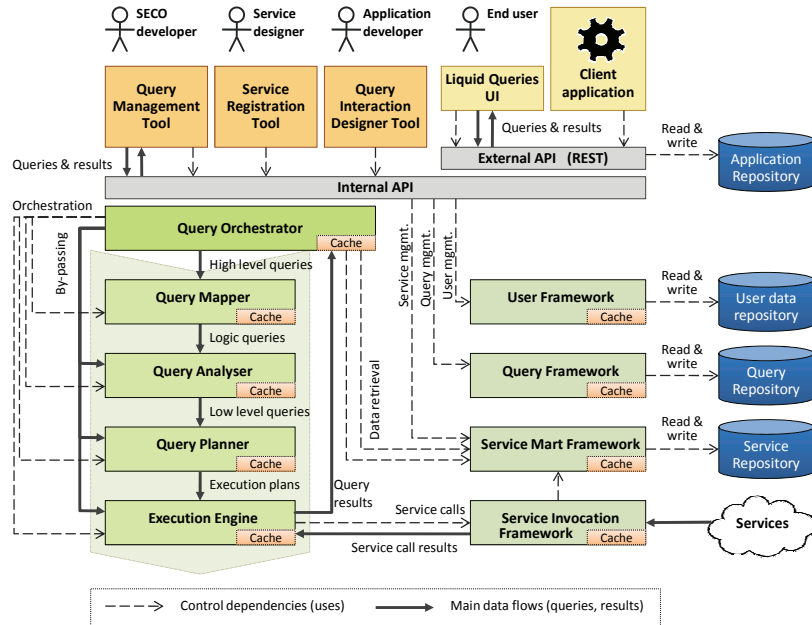


Fig. 11. Overview of the Search Computing framework

Search Computing users grossly belong to two categories. *End users* can only launch predefined applications and submit input to them through forms; *expert users* may also compose queries in the context of repositories of service marts and of their composition patterns (we say in such cases that users can build liquid queries, where their liquid nature comes from the fact that queries extend upon service marts more or less as stains over surfaces). In both case, however, we expect users to have some experience in data analysis (similar, e.g., to the basic skills required by spreadsheets) and we expect them to use such skills in manipulating results, which are shown in tabular format, and can be dynamically augmented online – we call them liquid results to highlight such dynamic and plastic nature of results, which can be manipulated by means of user controls.

Tools are intended to support three kinds of experts:

- *Service designers* register data sources in the system through the Service Mart Framework, by either interacting with existing Web services, or by exposing existing data sources, or by wrapping existing Web pages. They play the role of “data providers”.
- *Application developers* preselect some of the services and configure them so as to turn them into applications; specifically, they build user’s interfaces which either expose to expert users service marts and their connections or expose to end users simple forms accepting typed input. They play the role of “data brokers”.

- *SeCo developers* install, open and configure the SeCo modules upon suitable hardware resources and may perform fine tuning (or creation from scratch) of query and execution plans.

The upper part of Figure 1 shows that the tools provided to the designer and developer communities plug to an internal API, while end-user applications and interfaces in turn are accessible via an external API and therefore callable from any client environment. Finally, the bottom right part of Figure 1 shows three kinds of repositories, called *service repositories* (i.e., cache memories storing inputs and outputs of recent calls), *query repositories* (i.e., queries that were saved by user for subsequent restore operations), and *user data repositories* (i.e., profiles and administrative information). An additional application repository loads applications and stores user's interactions, so as to be able to remember and re-apply such interactions to new queries or to new results.

The various architectural elements forming the Search Computing prototype architecture defined above are described in different, autonomous chapters.

Chapter 9 deals with **service marts**, a novel concept for enabling the engineering and deployment of search services, i.e. of services whose main feature is the ability to respond ranked results organized by chunks (so as to enable a fine-grain control by the execution engine). Such results are produced by interacting with concrete data sources, which are made available through service interfaces, wrappers, or direct access to extensional data collections (databases, excel files, and so on). Thus, service marts are a conceptual abstractions providing information hiding, mapped to service interfaces which directly interact with concrete data sources.

Chapter 10 describes our **framework for query execution**; specifically it address the description of a query language for Search Computing, then the mapping of queries to service interfaces, then the composition methods that have been defined so far in Search Computing under the classical format of join methods, suitably extended to the search and web context. This chapter discusses query formulation and optimization up to the choice of join methods.

Chapter 11 deals with **ranking aggregation** in its most general formalization, and shows how ranking aggregation methods can be adapted to Search Computing in generating a join method which is capable of guaranteeing that the top-k results are selected.

Chapter 12 describes a **flexible architecture for Search Computing** (named Panta Rhei) which includes suitable abstractions for data production, consumption, and caching, with both data-driven and event-driven synchronization. Operations and flows of the Panta Rhei model are described at a high level, but they are designed for supporting the scalable execution of Search Computing queries in a variety of deployment architectures.

Chapter 13 shows a paradigm for asking Search Computing queries, called **liquid queries**, that can be articulated upon such flexible architecture, where a liquid query is capable of run-time modification by addition or dropping of sub-queries and by drilling down and rolling up information, much in the same way as with a data cube expressing the results in data analysis environments.

Chapter 14 shows how to **build and deploy applications** by means of a software engineering environment involving both “data providers”, who will register service marts, and “data brokers”, who will assemble applications. SeCo servers can be deployed upon a variety of computing architectures, hinting to future prototypes running upon highly scalable architectures and/or cloud computing systems. The chapter also discusses the business models that may favor the spreading of both data providers and data brokers.

Chapter 15 discusses **ranking opportunities in the context of life sciences**, which are characterized by a wide use of ranked information, thereby anticipating some of the specific issues featured by an appealing Search Computing application.

Chapter 9: Service Marts

Alessandro Campi¹, Stefano Ceri¹, Georg Gottlob²,
Andrea Maesani¹, Stefania Ronchi¹

¹Politecnico di Milano, Italy

{campi,ceri,maesani,ronchi}@elet.polimi.it

²University of Oxford, United Kingdom

georg.gottlob@comlab.ox.ac.uk

Abstract: The use of patterns in data management is not new: in data warehousing, data marts are simple conceptual schemas with exactly one core entity, describing facts, surrounded by multiple entities, describing data analysis dimensions; data marts support special analysis operations, such as roll up, drill down, and cube. Similarly, service marts are simple schemas which match "Web objects" by hiding the underlying data source structures and presenting a simple interface, consisting of input, output, and rank attributes; attributes may have multiple values and be clustered within repeating group. Service marts support Search Computing operations, such as ranked access and joins. When objects are accessed through service marts, responses are ranked lists of objects, which are presented subdivided in chunks, so as to avoid receiving too many irrelevant objects – cutting results and showing only the best ones is typical of search services. This chapter includes a survey of service definition standards (discussing the standards for service description and the current state-of-the-art for service registration and discovery), then introduces a formal definition of service marts and of connection patterns at the conceptual, logical, and physical levels. Then, we show how service marts can be implemented, by taking into account different kinds of data sources, and taking advantage of components (written in Java and SQL) and tools (such as a materialize specifically developed to help service mart implementation). We use such components and tools to build a collection of services used in a running example throughout the chapters of this part.

1. Introduction

The goal of Search Computing is to support search service integration, but a prerequisite for setting such goal is the availability of a large number of valuable search services. With a passive attitude, we could just wait for SOA (Service Oriented Architecture) to become widespread, and then use available services within our framework. However, few software services are currently designed to support search, and moreover a huge number of valuable data sources (the so called "long tail" of Web information) are not provided with a service interface. In this chapter, we therefore focus on the important issue of publishing service interfaces suitable for Search Computing on the data sources, so as to facilitate their use on the Web and at the same time to create the premises for their integration within the Search Computing framework.

Chapter 10: Join Methods and Query Optimization

Daniele Braga, Stefano Ceri, and Michael Grossniklaus

Dipartimento di Elettronica e Informazione, Politecnico di Milano,
Piazza Leonardo da Vinci 32, 20133 Milano, Italy
{braga|ceri|grossniklaus}@elet.polimi.it

Abstract. Joins between data sources are an essential ingredient of multi-domain queries, as they exploit connection patterns defined between service marts or between service interfaces. This chapter moves from the definition of a query language over service interfaces, sketching how queries can be directly expressed over service marts and how these can be translated over service interfaces. The fundamental operation discussed in this chapter is the binary join between two sources, which is influenced by the type (search vs. exact) of services and by the management (parallel vs. sequential) of service calls. Then, this chapter presents an optimization framework for queries over several service interfaces, which considers several cost metrics for mapping queries into query plans, consisting of specific operations over services, and includes a branch and bound approach to the exploration of the combinatorial search space of all possible query plans.

1 Introduction

This chapter delves into the issues of formulating and optimizing multi-domain queries over several services, focusing on the specific problems that arise due to the presence of search services in the queries. The distinguishing feature of a search service is to return answers in relevance order. In general, although the answers produced by a search service can be very numerous, users are only concerned with the answers provided within the first pages of results. Thus, a query strategy that retrieves all the answers from a search service is rarely appropriate. On the other hand, only the user can correctly evaluate the relevance of answers produced by search engines. Therefore, if a query involves several searches, all answers produced by the involved search engines should be composed in the query output and presented to the user for a correct evaluation. Moreover, the user expects answers in ranking order. Thus, while composing results from multiple services, answers should be presented according to a global ranking that is obtained either as an exact composition function of the rankings (then, we can talk about “top-k” results) or an approximation of that function.

In this chapter, we define a **formal model** for the optimization and the execution of multi-domain queries over services which expose heterogeneous information sources. Such model serves as a unifying perspective for several diverse possible application

Chapter 11:

Rank-join Algorithms for Search Computing

Ihab F. Ilyas¹, Davide Martinenghi², Marco Tagliasacchi²

¹University of Waterloo, Canada

²Politecnico di Milano

ilyas@uwaterloo.ca, martinenghi@elet.polimi.it, tagliacchi@elet.polimi.it

Abstract. Joins represent the basic functional operations of complex query plans in a Search Computing system, as discussed in the previous chapter. In this chapter we provide further insight on this matter, by focusing on algorithms that deal with joining ranked results produced by search services. We cast this problem as a generalization of the traditional rank aggregation problem, i.e., combining several ranked lists of objects to produce a single consensus ranking. Rank-join algorithms, also called top-k join algorithms, aim at determining the best overall results without accessing all the objects. The rank-join problem has been dealt with in the literature by extending rank aggregation algorithms to the case of join in the setting of relational databases. However, previous approaches to top-k queries did not consider some of the distinctive features of search engines on the Web. Indeed, as pointed out in the previous chapter, joins in this context differ from the traditional relational setting for a number of aspects: services can be accessed according to limited patterns, i.e. some inputs need to be provided; accessing services is costly, since they are typically remote; the output is returned in pages of results and typically according to some ranking criterion; multiple search services can be used to answer the same query; users can interact with the system in order to refine their search criteria. This chapter analyzes the challenges that need to be tackled in the design of rank-join algorithms within the context of Search Computing.

Keywords: rank-join, top-k, query optimization

1 Introduction

Information systems of different types use various techniques to rank query answers. In many application domains, end-users are more interested in the most important (top-k) query answers in the potentially huge answer space. Different emerging applications warrant efficient support for top-k queries. For instance, in the context of the Web, the effectiveness and efficiency of meta-search engines are highly related to efficient methods for rank aggregation. The latter is the problem of combining several ranked lists of items in a robust way to produce a single consensus ranking of the items. Most of these applications execute queries that involve joining and aggregating multiple inputs to provide the top-k results.

Chapter 12:

Panta Rhei: Flexible Execution Engine for Search Computing Queries

Daniele Braga¹, Stefano Ceri¹, Francesco Corcoglioniti¹ and Michael Grossniklaus¹

¹Dipartimento di Elettronica e Informazione, Politecnico di Milano,
Piazza Leonardo da Vinci 32, 20133 Milano, Italy
{braga, ceri, corcoglioniti, grossniklaus}@polimi.it

Abstract. The efficient execution of data-intensive computations over services is a challenging task: data are retrieved from remote sources and therefore are not available in the query engine until after the execution of these calls, but the system must be inherently efficient thereafter, by guaranteeing that data is immediately cached and processed efficiently, according to the best query plan. In this chapter, we present a flexible execution model for search computing queries, named Panta Rhei. The proposed execution engine paradigm adopts the producer/consumer model and supports both data-driven and event-driven synchronization, and their interplay. Query plans are modeled as directed graphs, whose nodes are processing units and whose edges are either control or data flows. While control flows synchronize service calls and unit execution, data flows transfer data between units that process data flows to produce query results. We present the specification of Panta Rhei by formally defining the units for data production, consumption, manipulation, and caching, as well as the control and data flows. Finally, we discuss how a query plan is expressed in terms of a query execution plan.

1 Introduction

Query execution in Search Computing is a data-intensive process. The computations required for answering a query, although performed upon the data resulting from service calls, are very similar to those performed by database management systems working on physically optimized tables. Therefore, a query execution engine supporting Search Computing must be able to efficiently support dynamic data extraction, storage and caching, as well as efficiently route data flows between special-purpose computational units, whose design has been optimized so as to guarantee the fast production of query results.

Due to the very nature of many of these tasks and their embedding within Web-based contexts, which are subject to continuous change, performances of data-intensive service interactions are very hard to predict. Moreover, the execution engine must be strongly connected to the query user interface, so as to adapt to user requests that dynamically alter the query requirements, either by specializing current requests or by adding new requirements. For these reasons, the design of the query execution

Chapter 13:

Liquid Queries and Liquid Results in Search Computing

Alessandro Bozzon¹, Marco Brambilla¹, Stefano Ceri¹, Piero Fraternali¹,
and Ioana Manolescu²

¹Dipartimento di Elettronica e Informazione, Politecnico di Milano,
Piazza Leonardo da Vinci 32, 20133 Milano, Italy
{alessandro.bozzon,marco.brambilla,stefano.ceri,piero.fraternali}@polimi.it
²INRIA, Saclay-Ile-de-France and LRI, Université de Paris Sud-11, France
ioana.manolescu@inria.fr

Abstract. Liquid queries are a flexible tool for information seeking, based on the progressive exploration of the search space; they produce “fluid” results which dynamically adapt to the shape of the query, as a liquid adapts to its container. The liquid query paradigm relies on the SeCo service mart and multi-domain query execution concepts: an expert user selects a priori the service marts relevant to the information seeking task at hand and the connections necessary to join them, and publishes such a definition in the SeCo back-end. The Liquid Query client-side interface consumes the application definition created by the expert and dynamically builds a query interface for the end-user. Such interface allows one to supply keywords to query the pre-configured service marts and offers controls for exploring the combinations computed by the SeCo execution engine. The interaction commands are based on a tabular representation of results and comprise: reordering, clustering, addition or deletion of attributes, addition of extra service marts to the query for specific items in the result set or for the entire result set, request of more results from all services or from selected ones, expansion of details on selected items, and more. The Liquid Query is equipped with multiple data visualization options suited to render multi-domain results and can be instrumented with indicators showing the quality of the result set.

Keywords: user interfaces, exploratory search, search computing.

1 Introduction

As users get more and more acquainted with the use of the Web for addressing their information needs, the role played by search engines in both professional and everyday life grows. A recent study by Yahoo confirms the centrality of search in Web usage: one out of every five page views of the analyzed data set is related to some search task [21]. Initially, search engine interfaces were primarily exploited for locating specific documents. The “Google-style” user interface is the perfect example

Chapter 14:

Building Search Computing Applications

Alessandro Bozzon, Marco Brambilla, Stefano Ceri, Francesco Corcoglioniti,
Nicola Gatti

Politecnico di Milano, Dipartimento di Elettronica ed Informazione,
V. Ponzio 34/5, 20133 Milano, Italy
{alessandro.bozzon | marco.brambilla | stefano.ceri | nicola.gatti} @ polimi.it ,
francesco.corcoglioniti@gmail.com

Abstract. Search Computing aims at opening the Web to a new class of search applications, by offering enhanced expressive and computational power. The success of Search Computing, as of any technical advance, will be measured by its impact upon the search industry and market, and this in turn will be highly influenced by reactions of Web users and developers. It is too early to anticipate such reactions – as the technology is still “under construction” – but this chapter attempts a first identification of the possible future players in the development of Search Computing applications, by grossly identifying the roles of “data source publishers” and of “application developers”, and by discussing how classical advertising-based models may support the new applications. This chapter also describes the high-level design of the prototyping environment that is currently under development and how the design will support the deployment upon high performance architectures. Finally, we describe advertising as the prevalent business model of the search engines industry, and briefly discuss the options for the evolution of such model in the context of Search Computing.

Keywords: Search Computing, software engineering, development process, advertising models, cloud computing, software architectures.

1 Introduction

The distinguishing feature of Search Computing is the ability of combining, at query execution time, knowledge extracted from various domain-expert Web sources, thus yielding to knowledge that is more accurate and complete than the knowledge available to general-purpose search systems. Such expertise (about cultural events, medical specializations, popular rock songs, and so on) is contributed through either social processes (e.g., rating, tagging, commenting) or a long and careful knowledge construction process by experts. At the current state of the art, multi-domain queries over such engines can be answered only by patient and expert users, whose strategy is to interact with specialized engines one at a time, and feed the result of one search in input to another one, reconstructing answers in their mind.

With the advent of service computing and the growing interest for the Web as the predominant interface for any human activity, we expect such knowledge to become

Chapter 15:

Search Computing and the Life Sciences

Marco Masseroli¹, Norman W Paton², Irena Spasić²

¹ Dipartimento di Elettronica e Informazione, Politecnico di Milano,
Piazza Leonardo da Vinci 32, 20133 Milano, Italy.
masseroli@elet.polimi.it

² School of Computer Science, University of Manchester,
Oxford Road, Manchester M13 9PL, UK.
{npaton, i.spasic}@manchester.ac.uk

Abstract. Search Computing has been proposed to support the integration of the results of search engines with other data and computational resources. A key feature of the resulting integration platform is direct support for multi-domain ordered data, reflecting the fact that search engines produce ranked outputs, which should be taken into account when the results of several requests are combined. In the life sciences, there are many different types of ranked data. For example, ranked data may represent many different phenomena, including physical ordering within a genome, algorithmically assigned scores that represent levels of sequence similarity, and experimentally measured values such as expression levels. This chapter explores the extent to which the search computing functionalities designed for use with search engine results may be applicable for different forms of ranked data that are encountered when carrying out data integration in the life sciences. This is done by classifying different types of ranked data in the life sciences, providing examples of different types of ranking and ranking integration needs in the life sciences, identifying issues in the integration of such ranked data, and discussing techniques for drawing conclusions from diverse rankings.

Keywords: search computing, bioinformatics, data integration, ranked data

1 Introduction and Motivation

Experimental studies in the life sciences give rise to **large quantities of diverse, complex data**. Taking *genomics* as an example, initial sequencing work gives rise to a raw genome sequence, which in turn is annotated with the predicted locations of genes. In essence, every cell in an organism contains the same genome sequence, but biological processes cause the products described by the genome to be created differently in different cells. For example, different cells contain different collections of proteins, and over time the quantities of different proteins within a cell vary. As a result, to understand the dynamic behavior of a cell, it is necessary to measure quantities of different types of molecules within the cell. *Functional genomics* encompasses a collection of experimental techniques, including transcriptomics,

Search Computing Dictionary

1. Service Framework

Service mart:

Data abstraction modeling \rightarrow **data sources** referring to the same kind of “Web object” (e.g., hotels, movies, etc.). Service marts are described at three different levels: a higher level describing the \rightarrow **service mart signature** and \rightarrow **connection patterns**, an intermediate level describing all the \rightarrow **service interfaces** available for the \rightarrow **service mart signature**, and a lower level describing the \rightarrow **service implementation** of each **service interface**.

Data source:

A \diamond **data repository** that can be accessed programmatically, such as a \diamond **database**, an \diamond **application program**, a \diamond **Web Service**, a \diamond **wrapper** extracting data from a \diamond **Web site**, etc.

Service mart signature:

A characterization of a \rightarrow **service mart** in \diamond **relational** terms. It consists of the name of the service mart and the set of \rightarrow **attributes of the service mart**.

Attributes of a service mart:

Attributes characterizing the different \diamond **fields** that are handled by the \rightarrow **service mart** to describe properties of a Web object. Each attribute is associated with a \rightarrow **built-in type** and possibly with a set of \rightarrow **keywords**. The set of attributes of a service mart can be either single-valued or multi-valued. Multi-valued attributes can be put together to form a \rightarrow **repeating group of attributes**.

Repeating group of attributes:

A set of multi-valued attributes of a \rightarrow **service mart** that collectively define one \diamond **property** of a Web object. Within the same Web object, the attributes of the same repeating group all have the same number of values.

Built-in type:

Any concrete type of a programming language supporting Search Computing, such as int, string, float, double, date, ...

Keyword:

A term characterizing the semantics of an attribute. Keywords are normally taken from vocabularies, e.g. \diamond **WordNet**.

Service interface:

A characterization of one of the possible \rightarrow **service implementations** of a \rightarrow **service mart** given by one \diamond **URI**, one set of \rightarrow **Service Parameters**, and one \rightarrow **access pattern**. The same \rightarrow **access pattern** is shared by possibly more than one service interface. Every service interface has exactly one service implementation.

Adornment:

A pair consisting of an \rightarrow **access pattern** and a \rightarrow **scoring function** associated to a \rightarrow **service interface**. The same adornment is shared by possibly more than one service interface.

Adorned service mart:

The association of a \rightarrow **service mart** with an \rightarrow **adornment** existing in at least one \rightarrow **service interface** of the service mart.

Service implementation:

A concrete \diamond **implementation** of one \rightarrow **service interface**, thus providing operations for accessing data available at one or more \rightarrow **data sources** according to the \rightarrow **access pattern** of the service interface.

Access pattern:

An labeling of a \rightarrow **service mart signature** that determines which attributes of the \rightarrow **service mart** are \diamond **output** and which ones are \diamond **input** attributes. Among \diamond **output** attributes, some may be labeled as ranked and associated with a \rightarrow **score**, and in that case the service implementation returns its results in an order which depends on them. There can be several \rightarrow **service interfaces** for the same \rightarrow **access pattern**.

Scoring function:

Function mapping each tuple output by a \rightarrow **service implementation** into a \rightarrow **score**. The mapping can be determined based on the \rightarrow **attributes of the service mart** of the \rightarrow **service interface** associated with the scoring function, or the position of the tuple in the output, and depends on the \rightarrow **ranking type**. Note that if the \rightarrow **ranking type** is unranked, then the scoring function is a fixed constant (e.g., 1).

Connection pattern:

Specification of the join between two \rightarrow **service marts**, expressed as a conjunctive expression of join predicates using the \rightarrow **attributes of the service marts**. Every join predicate is built by a pair of \rightarrow **service mart attributes**, orderly taken from the first and second \rightarrow **service mart**, and an arbitrary \diamond **comparison operator**.

Service mart design:

Process of defining the signature of a \rightarrow **service mart**, then its \rightarrow **access patterns**, then the \diamond **data sources** that can support queries expressed with those \rightarrow **access patterns**, each of which is registered as a \rightarrow **service interface**. The process is completed by defining \rightarrow **connection patterns** between pairs of \rightarrow **service marts**.

Service mart implementation:

Production of a \rightarrow **service implementation** for a given \rightarrow **service interface**. The process may use components and tools for data materialization, extraction, conversion, and translation.

Service registration:

Addition of a \rightarrow **service implementation** and its \rightarrow **service interface** to a \rightarrow **service mart**. With the service registration, the service implementation is made available at the \diamond **URI** specified in the service interface.

Ranking type:

A set of properties regarding the ranking, i.e., the order in which the results of consecutive \rightarrow **fetches** performed on a \rightarrow **service implementation** are returned. The ranking type defines:

- whether a \rightarrow **service implementation** is \diamond **ranked** or \diamond **unranked**,
- whether it is opaque or visible (i.e., the \rightarrow **attributes of the service mart** include a \rightarrow **score**),
- the set of output attributes the ranking depends on (the set is empty if it does not depend on output attributes)
- the \rightarrow **scoring function** mapping the set of output attributes the ranking depends on into a \rightarrow **score**, and
- whether the ranking is \diamond **ascending** or \diamond **descending**.

Search Service:

Service implementation that returns \rightarrow **chunks** of ranked results that are possibly \diamond **unbounded** in number. The corresponding ranking type is \diamond **ranked**.

Exact Service:

Service implementation whose corresponding ranking type is \diamond **unranked**. The results returned by such a service implementation are \diamond **finite** and not \rightarrow **chunked**, except for technical reasons.

Service parameter:

Meta data describing the \rightarrow **service implementation** of a \rightarrow **service mart**, being one of \rightarrow **ranking type**, \rightarrow **cacheable**, \rightarrow **cache time-to-live**, \rightarrow **isChunked**, \rightarrow **chunk size**, \rightarrow **ERSPI**, \rightarrow **decay factor**, \rightarrow **response time**, and \rightarrow **cost**.

Fetch:

Request for the next \rightarrow **chunk** of results from a \rightarrow **service implementation**.

Chunked:

Property of a service implementation whose results are organized in \rightarrow **chunks**.

Chunk:

Any page of results in the form of \diamond **tuples** returned by a \rightarrow **service implementation**.

Cacheable:

Parameter of a \rightarrow **service interface**. True if the results returned by a \rightarrow **service implementation** can be stored in a \diamond **cache**; false otherwise.

Cache time-to-live:

Parameter of a \rightarrow **service interface**. Duration of the validity of \diamond **cached data**.

isChunked:

Parameter of a \rightarrow **service interface**. True if the \rightarrow **service implementation** is \rightarrow **chunked**; false otherwise.

Chunk size:

Parameter of a \rightarrow **service interface**. The number of \diamond **tuples** in a \rightarrow **chunk**.

ERSPI (expected result size per invocation):

Parameter of a \rightarrow **service interface**. \diamond **Positive real number** expressing the expected size of the result produced by a \rightarrow **service implementation** when called with arbitrary input values; in \diamond relational terms, expresses the \diamond **average cardinality** of the result of a query. The ERSPI is not very significant with search services, as the complete result may consist of a very high number of \diamond **tuples**, which however are not completely retrieved by fetch operations.

Selectivity:

\diamond **Positive real number** expressing the ratio between \rightarrow **ERSPI** of a \rightarrow **service implementation** and the total number of tuples that can be returned by the \rightarrow **service implementation**.

Decay function:

Parameter of a \rightarrow **service interface**. A function that models the decay of the \rightarrow **scores** of the results returned by consecutive \rightarrow **fetches**.

Response time:

Parameter of a \rightarrow **service interface**. Average \diamond **response time** of a \rightarrow **fetch**.

Cost:

Parameter of a \rightarrow **service interface**. \diamond **Monetary cost** of a \rightarrow **fetch**.

Score:

Value in the $[0,1]$ interval indicating the quality (1=highest, 0=lowest) of a \diamond **tuple** according to a predefined criterion as specified by a \rightarrow **scoring function**.

2. Query Expression and Optimization

Query on service marts:

A query is a \diamond **graph** whose nodes are labeled with \rightarrow **service marts** and whose edges are labeled with \rightarrow **connection patterns**. Nodes are additionally labeled with \diamond **selection predicates** over the \rightarrow **service mart attributes** and \diamond **projected attributes** forming the result. The same \rightarrow **service mart** can appear multiple times in the same query. Every query is interpreted as a \diamond **conjunctive query**, whose \diamond **join predicates** are built from \rightarrow **connection patterns**.

Query on adorned service marts:

A query is a \diamond **graph** whose nodes are labeled with \rightarrow **adorned service marts** and whose edges are labeled with \rightarrow **connection patterns**. Additional labeling is as for \rightarrow **queries on service marts**. A \rightarrow **query on service marts** can be turned into a query on adorned service marts simply by substituting to every \rightarrow **service mart** labeling a query node one of the corresponding \rightarrow **adorned service marts**.

Feasible query:

A \rightarrow **query on adorned service marts** is feasible if the \rightarrow **access patterns** used in the query satisfy given well-formedness conditions guaranteeing that it is possible to compute the results as if access patterns were not present.

Query on service interfaces:

A \rightarrow **query on adorned service marts** can be turned into a query on service interfaces simply by substituting to every \rightarrow **adorned service mart** labeling a query node one of its \rightarrow **service interfaces**. Recall that every \rightarrow **service interface** is also associated with a \rightarrow **service implementation**.

Query formulation:

The process of specifying \rightarrow **feasible queries** on \rightarrow **service interfaces** and possibly a \rightarrow **ranking function** associated with the query. Suitable interfaces assist users in formulating only \rightarrow **feasible queries**.

Query processing:

The process of executing \rightarrow **feasible queries** on \rightarrow **service interfaces**, producing a \rightarrow **query result**. The process consists of (1) query installation (2) query optimization (3) query execution (4) user interaction for interpreting results, possibly leading to further steps of query processing.

Query result:

A \diamond **list** of \rightarrow **combinations**.

Combination:

A \diamond **tuple** of the \rightarrow **query result**, built by the matching \diamond **tuples** produced during \rightarrow **query execution**, where a query result \diamond **tuple** includes exactly one \diamond **tuple** from the results produced by each \rightarrow **service implementations** involved in the query execution. Combinations in the query result are ordered according to a \rightarrow **ranking function** associated with the query.

Ranking function:

A \diamond **weighted sum** of the individual \rightarrow **scores** of the \diamond **tuples** forming a \rightarrow **combination**, which returns the ranking of the \rightarrow **combination** according to the users' preferences.

Query optimization:

The process of building query plans for \rightarrow **correct queries**, and then selecting the \rightarrow **optimal query plan** which is then executed.

Query plan:

A \diamond **directed acyclic graph** made of \rightarrow **nodes of a query plan** and \rightarrow **edges of a query plan**, representing an operational specification of the order in which \rightarrow **service interfaces** are to be invoked in order to retrieve a specified number of \rightarrow **chunks**, and of how \rightarrow **joins** are to be performed between \rightarrow **chunks** according to specific \rightarrow **join strategies**.

Node of a query plan:

Any node in a \rightarrow **query plan**, representing the \rightarrow **initial node**, or the \rightarrow **final node**, or an \rightarrow **invocation node**, or a \rightarrow **selection node**, or a \rightarrow **join node**.

Edge of a query plan:

A directed edge in a \rightarrow **query plan**, representing the data transfer between two \rightarrow **nodes of a query plan**

Initial node:

A \rightarrow **node of a query plan** with no incoming \rightarrow **edges** and one or more outgoing \rightarrow

edges, used to denote the user input. It is the only node in the \rightarrow **query plan** without incoming edges, and every \rightarrow **query plan** must have exactly one initial node.

Final node:

A \rightarrow **node of a query plan** with no outgoing \rightarrow **edges** and one or more incoming \rightarrow **edges**, used to denote the production of the query result. It is the only node in the \rightarrow **query plan** without outgoing edges, and every \rightarrow **query plan** must have exactly one final node.

Invocation node:

A \rightarrow **node of a query plan** that represents the invocation of a \rightarrow **service interface**.

Selection node:

A \rightarrow **node of a query plan** associated with a \diamond **selection operation**.

Join:

An operation between two \rightarrow **service interfaces**, called join operands, that uses \rightarrow **connection patterns** in order to form \rightarrow **combinations** of the \diamond **tuples** returned as results of service invocations. Joins are classified as \rightarrow **pipe joins** or as \rightarrow **parallel joins**, depending on the relationships between the \rightarrow **access patterns** associated with the join operands.

Pipe Join:

A configuration of two \rightarrow **invocation nodes** A and B in the \rightarrow **query plan** such that A and B are connected by a directed path from A to B, and one or more output attributes of A contain values which are used as input in B (i.e., attributes are labeled as input in the \rightarrow **access pattern** of B).

Pipe node:

The destination node in the directed path of a \rightarrow **pipe join** configuration.

Parallel Join:

A configuration of one \rightarrow **join node** and two antecedent nodes A and B in the \rightarrow **query plan** such that A and B correspond to service interfaces and neither the output attributes of A contain values that are used as input in B nor the output attributes of B contain values that are used as input in A.

Join node:

A \rightarrow **node of a query plan** with exactly two incoming \rightarrow **edges of a query plan** and one or more outgoing \rightarrow **edges of a query plan**, used to denote a \rightarrow **parallel join**.

Join strategy:

Defines the order in which \rightarrow **chunks**, received at a join or pipe node, are to be

considered in order to produce → **combinations** (i.e. join results). Different join strategies correspond to different orders in which the points in the → **join search space** are to be considered. Join execution strategies can use the → **nested loop** or → **merge scan** strategies and perform a → **rectangular exploration** or a → **triangular exploration**.

Nested Loop:

A → **join strategy** in which all → **chunks** from one service are retrieved (and possibly cached) before proceeding to retrieve the next → **chunk** from the other service.

Merge Scan:

A → **join strategy** in which → **chunks** from the two services being joined are alternatively retrieved in a fixed alternated order.

Rectangular Exploration:

An exploration of join combinations in which all available candidate combinations are considered as soon as new chunks are made available. The rectangular strategy can be applied to both → **merge scan** and → **nested loop**.

Triangular Exploration:

An exploration of join combinations in which only the one half of the available candidate yielding to better rankings are considered as soon as new chunks are made available. The triangular strategy can be applied to both → **merge scan** and → **nested loop**.

Cost model:

A selection of the relevant → **service parameters** and mathematical relationships thereof for assessing the execution cost of a query, representing a specific objective to guide the optimization process.

Cost function:

A mathematical function of some chosen → **service parameters** whose result is a measure of the → **cost of a query plan**.

Cost of a query plan:

The result of applying a specific → **cost function** with a specific → **cost model** to a specific → **query plan**.

Optimal query plan:

The → **query plan** whose → **cost** is minimal among all possible plans for a given query.

3. Query Execution

Execution plan:

Directed graph consisting of nodes in the form of → **scheduler units** and edges in the form of either → **data flow edges** and → **control flow edges**. An execution plan represents the physical evaluation of a → **query plan**.

Scheduler unit:

A node of the → **execution plan** that has a well defined semantics in terms of an operation that it performs within the execution plan. → **producer Unit** and → **consumer Unit**.

Producer unit:

A scheduler unit that produces output data, i.e. a publisher. → **service Invocation unit**, → **(parallel) join unit**, → **ranker unit**, → **chunker unit**, → **selection unit**, and → **cache unit**.

Consumer unit:

A scheduler unit that consumes input data, i.e. a subscriber. → **service invocation unit**, → **(parallel) join unit**, → **ranker unit**, → **chunker unit**, → **selection unit**, and → **cache unit**.

Control flow edge:

An edge of the execution plan that transmits a control signal from one unit to another. Signals are of three kinds: → **pulse signals**, → **suspend / resume signals**.

Data flow edge:

An edge of the execution plan that transports data, → **chunks** of tuples, from a → **producer unit** to a → **consumer unit**.

Clock unit:

Controls $n > 0$ → **service invocation units** using → **pulse signals** which are produced at each → **clock cycle**. A clock unit is controlled by a → **clock function** and adjusted by → **clock modifiers**.

Pulse signal:

Pulse signals are emitted by the clock and trigger → **service invocation units** to fetch data. They contain the specification of the number of → **fetches** to be performed in each → **clock cycle**. A pulse signal can as well be produced by a data → **producer unit** when it first produces a tuple in output.

Clock frequency:

Parameter of the \rightarrow **clock unit** that describes how many \rightarrow **clock cycles** per time unit a clock has. Example: 1/50 s means 50 clock cycles per second.

Clock cycle:

The period during which the clock triggers the services according to current n -tuple of the \rightarrow **clock function**. The number of clock cycles per time unit is determined by the \rightarrow **clock frequency**.

Clock function:

A clock function is a regular expression that describes, for each \rightarrow **service invocation unit** controlled by the clock and for each \rightarrow **clock cycle**, the number of \rightarrow **fetches** to be performed by the Service Invocation Unit. A regular expression for a clock function maps into a sequence of clock values given as n -tuples of \diamond **integer numbers**, where n is the number of Service Invocation Units controlled by the \rightarrow **clock unit**. As an example, $(1,1)(2,2)^*$ represents a sequence in which two services are invoked once in the first clock cycle, and twice in any subsequent clock cycle. Each clock cycle of a clock function is expected to control the \rightarrow **join execution strategy** in terms of \rightarrow **chunks** produced.

Clock modifier:

A clock modifier for a \rightarrow **clock unit** is an \diamond n -tuple of \diamond **positive real numbers**, where n is the number of \rightarrow **service invocation units** controlled by that clock unit. During query execution, clock modifiers are used to adjust the number of \rightarrow **fetches** for each Service Invocation Unit controlled by the clock unit.

Suspend/resume signals:

A suspend signal is sent by the \rightarrow **parallel join unit** to the \rightarrow **clock unit** once the \rightarrow **skew factor** of the join unit has been reached; it is also sent when a given number k of results have been produced. On receipt of a suspend signal, the clock suspends execution only at the end of a clock cycle. As soon as the \rightarrow **join execution strategy** realigns, a resume signal is sent to the clock to continue query execution. Suspend and resume signals can as well be produced by \rightarrow **end users**, the former occur when they want to suspend the execution of an execution plan, e.g. because they are temporarily satisfied with the current results, the latter when they want to resume execution of an \rightarrow **execution plan**.

Chunker unit:

A \rightarrow **scheduler unit** that, based on the specification of a \rightarrow **chunking function**, constructs new \rightarrow **chunks** with the tuples it gets as input. Internally, the chunker unit breaks up the chunks in input and produces new chunks in output.

Chunking function:

A specification, given in the form of a regular expression, of how many tuples of the input are to be combined into every chunk in output. Example: $4^3, 3^*$ means: first produce 3 chunks with 4 tuples, then continue with chunks of 3 tuples.

Service invocation unit:

A service invocation unit fetches data by using a specific → **access pattern** of a given → **service mart**. Service invocation units can either invoke → **search services** or → **exact services**. Service invocation units are triggered by a → **pulse signal** or by the availability of input data. When a service invocation follows one or more service invocations in the execution plan, it implicitly computes a → **pipe join**.

Parallel Join Unit:

It joins the results of two search service calls; executing a join causes the production of result tuples, called → **combinations**, according to a → **join execution strategy**. Joins are associated with a → **Skew** value and can emit → **suspend/resume Signals** to the clock that controls the → **service invocation units** generating the data for the join unit.

Skew:

In the case of runtime service behavior diverging from the expected one, → **parallel join units** are allowed to deviate from their predefined → **join execution strategy**. The maximum permitted deviation is specified by the skew value. It states how many → **chunks** on the x or y axis can be processed, in addition to the planned ones, before the join unit sends a → **pause signal** to its clock. The signal stops the production of new chunks and permits the join unit to wait for already fetched chunks to propagate through the → **query execution plan**.

Cache unit:

Cache units store the output generated by → **producer units** and make it available to → **Consumer Units**. Therefore they serve as a common point of synchronization in the producer/consumer (or publish/subscribe) model. Inserts of data that is already cached have no effect. Caches may trigger their consumers whenever new data is available.

Ranker unit:

Ranks or re-ranks tuples in → **chunks** according to a → **ranking function**. It has a blocking behavior, which is controlled by the specification of the ranking function. More precisely, the unit re-ranks tuples up to a certain size or time limit, after which it outputs the ranked tuples and resets itself. It may therefore work as a synchronization point when receiving tuples from more than one unit.

Selection unit:

Performs selections (consisting of \diamond **Boolean expressions** of \diamond **selection predicates**) over the dataflow \diamond **tuples** in input; only \diamond **tuples** satisfying the selection are produced in output.

D. Liquid Queries**Resource graph:**

Graph showing to \rightarrow **expert users** nodes labeled with \rightarrow **service marts** or \rightarrow **service interfaces** and arcs labeled with \rightarrow **connection patterns**; \rightarrow **expert users** build queries by means of graphic interfaces, with a \diamond **mash-up** style. Query formulation tools include facilities for selecting nodes and arcs, selecting result attributes, describing the rank function, and describing possible query augmentations.

Expert user:

Knows about resources and their semantics, assembles queries for personal use but also for saving them and making them available to \rightarrow **end users**.

End user:

Interacts with query by submitting values into query forms, possibly within slots which are well-defined in terms of semantics and with expected data types (for matching with given \rightarrow **attributes of service marts**).

Liquid query:

Query whose formulation is flexible and fluid, with the purpose of discovering the user intent while submitting queries and reading results, in a more precise manner, through a step-by-step approach that allows the user to get closer and closer to the desired information. A liquid query consists of an initial \rightarrow **query interface**, may support several \rightarrow **query augmentations**, and produces a \rightarrow **liquid result**.

Query augmentation:

Possibility of adding to a query a simple sub-query, consisting of joining one of the \rightarrow **service interfaces** already in the query to another \rightarrow **service interface**, connected to it through a \rightarrow **connection pattern**. The process is recursive and can be applied to the \rightarrow **result page** or to a subset of the \rightarrow **result combinations** shown in the \rightarrow **result page**.

Liquid result:

a set of results of a liquid query, which in turn is flexible thanks to a set of \rightarrow **result interaction primitives** that allow the end-user to modify the \rightarrow **result structure** and set of \rightarrow **result instances**, possibly shown partitioned in \rightarrow **result pages**.

Query interface:

A **user interface** that is offered for formulating the initial query. It consists of a set of → **search service forms**, which in turn are composed by → **search fields**.

Result instance:

Result tuple of a → **liquid query**, that complies with the corresponding → **result structure**.

Result page:

Set of → **result instances** that are shown together within the → **liquid result interface**.

Result structure:

The **schema** of the result of a liquid query. It consists of a set of typed attributes. See also → **Attributes of a Service Mart**.

Search fields:

Input fields that compose a → **search service form**.

Search service form:

Web form that allows user to submit parameters required by a search service in a → **query interface**.

Result interaction primitive:

User command enabled within → **liquid results** to refine, modify, extend the result itself (either in terms of → **result structure** or → **result instances**). Result interaction primitives are listed at the end of this dictionary and are marked with the symbol §.

§ Cluster:

Clustering adjacent tuples on a specific attribute and hiding duplicate valuesDefining a new grouping on the results.

§ Uncluster:

Removing a clustering on the results.

§ Sort:

Sorting the results currently being displayed according to one or more different attributes with respect to the ones initially defined. Sorting can be ascending or descending.

§ Unsort:

Removing a sorting on the results.

§ Drill-down:

Visualization of (already available but currently hidden) additional attributes .

§ Roll-up:

Hiding some attributes that are currently visible (and accordingly remove duplicates).

§ Filter:

Reducing the number of shown results based on a simple condition on one attribute.

§ DeleteInstance:

Locally deleting one instance from the currently displayed items. This can be seen as a particular case of filter operation.

§ RemoveFilter:

Canceling a filter currently applied.

§ More (all):

Asking for more results from the same query, by extracting results from every involved → **service mart**.

§ More of one service:

Asking for more results on a limited set of → **service marts** involved in the query.

§ Expand:

Asking for additional results (coming from other → **service marts**, connected through predefined → **connection patterns**) on a limited set of items listed in the current result set of the query.

§ Query re-ranking:

Ranking the results of the query according to a different ranking function wrt the one initially defined. This might produce a different set of results than those displayed before re-ranking.

§ Re-joining:

Redefining the join conditions between the services (among the acceptable predefined → **connection patterns**).

§ Refinement:

Submitting new query fragments for refining the search criteria, according to some paradigms to be defined (e.g. faceted search).

§ Pivoting:

Clustering together all instances with the same multivalued attribute value or repeating group value, thereby rendering the other attributes as repeating groups.

§ ChangeProvider:

changing the provider of a specific → **service interface**.