Liquid Query: Multi-Domain Exploratory Search on the Web

Alessandro Bozzon, Marco Brambilla, Stefano Ceri, Piero Fraternali
Politecnico di Milano Dipartimento di Elettronica ed Informazione, Via Ponzio 34/5, 20133 Milano, Italy
{bozzon, mbrambil, ceri, fraterna}@elet.polimi.it

ABSTRACT

In this paper we propose the Liquid Query paradigm, to support users in finding responses to multi-domain queries through exploratory information seeking across structured information sources (Web documents, deep Web data, and personal data repositories), wrapped by means of a uniform notion of *search service*. Liquid Query aims at filling the gap between general-purpose search engines, which are unable to find information spanning multiple topics, and domain-specific search systems, which cannot go beyond their domain limits. The Liquid Query interface consists of interaction primitives that let users pose questions and explore results spanning over multiple sources incrementally, thus getting closer and closer to the sought information. We demonstrate our approach with a prototype built upon the YQL (Yahoo! Query Language) framework.

Categories and Subject Descriptors

H.5.4 [**Information Interfaces and Presentation**]: Hypertext/ Hypermedia. D.2.2 [**Software Engineering**]: Design Tools and Techniques. H.3.3 [**Information Storage and Retrieval**]: Information Search and Retrieval – *Search process*.

General Terms

Design, Experimentation, Human Factors, Languages.

Keywords

Exploratory search, Multi-domain search, Search computing, Federated search engine, Web, Search service, Search engine.

1. INTRODUCTION

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 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 daily.

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 [36]; 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.

Information seeking as a process-intensive task has been recognized by Marchionini [25] and White [34], who introduced the notion of **exploratory search**, defined as the situation in which the user starts from a not-so-well-defined information need and progressively discovers more on his need and on the information available to address it, with a mix of look-up, browsing, analysis and exploration. Functionality for exploratory search is showcased by several last-generation search systems, using a variety of different tools: Dynamic Faceted Taxonomies [30] (as used e.g., in DBLP faceted search [13] and Clusty.com [11]), topic exploration (e.g., in the Kosmix topical search engine), community feedback and social wisdom (e.g., as in the Hunch problem solving engine [18]) are just a few examples.

In parallel to the evolution of the user's behavior, search solutions have evolved also in the data they can collect and present in response to a query. Focus has shifted from document crawling and indexing to the collection of heterogeneous data sources, whereby documents are integrated with semi-structured or structured data coming from the deep Web [3] and enriched with semantics, extracted either manually or automatically. This trend impacts both the way in which queries are formulated (e.g., the Wolfram Alpha search engine [35] accepts in input structured expressions with a domain specific syntax and semantics, like mathematical formulas and stock comparison sequences) and results are presented (e.g., in the Google Squared system [16] the user may perform a plain keyword search, and the system responds with data organized in tabular format, which can be extended both vertically, by adding further "objects", and horizontally, by inspecting more attributes of the displayed objects).

In this paper, we focus on an important dimension of exploratory search, the support of **multi-domain queries**, i.e., queries over multiple semantic fields of interest. When a query addresses a specific domain (e.g., travels, music, shows, food, movies, health, genetic diseases, etc.), domain-specific search engines provide more focused results than general-purpose ones; but their applicability is limited to a given domain. Thus, one can separately find best travel offers and interesting music shows, or conduct genetic analysis and investigate the related medical literature, but can hardly combine information from diverse yet related domains. An expert user can perform several independent searches and manually combine his findings, but the missing aspect is the ability of automatically "joining" the results of various search processes so as to build a comprehensive answer, integrating relevant pieces of information found in different domains.

From the technology viewpoint, Web services are nowadays available to provide a sound architectural framework for multi-domain information retrieval. Service interfaces are a popular way of "opening up" systems, including notorious Web applications such as Google, Yahoo, Amazon, Twitter, YouTube, MySpace, and Facebook. This offers to search, as to any other software

Copyright is held by the International World Wide Web Conference Committee (IW3C2). Distribution of these papers is limited to classroom use, and personal use by others.
WWW 2010, April 26–30, 2010, Raleigh, North Carolina, USA.
ACM 978-1-60558-799-8/10/04.

discipline, the possibility of building more ambitious systems, based upon *search service orchestration*.

In this paper we propose Liquid Query as a new paradigm allowing users to formulate, and get responses to, multi-domain queries through an exploratory information seeking approach, based upon structured information sources exposed as software services. The Liquid Query interface presents the users composite answers, obtained by aggregating search results from various domains; result composition is achieved through *join* – a classic operation of data management, by exploiting the structural information afforded by the search service interfaces that wrap the domain-specific data sources.

The Liquid Query interface visibly highlights the contribution of each search service to the composite result set, and supports the users in fine tuning their requests to the underlying search services in several ways: by expanding the query with an extra search service, by adding or dropping attributes of an object, by asking for more results from a specific service, by aggregating results, by reordering them, by adding details (drill-down) or removing them (roll-up), and by choosing the best data visualization format at the level of individual object, combination or results set. In playing with the query and its result set, users alter the schema of the result set "container", and then results dynamically flow inside the new schema, adapting to it as a liquid to its container. All the provided manipulation operations apply to a tabular data representation similar to that of Google Square, but with increased functionality derived from the knowledge that each item in the result set is actually a combination of objects coming from different search services.

Liquid Query has been developed in the context of the Search Computing project [10] (http://www.search-computing.it), which aims at the construction of a platform for multi-domain queries across search services. The project is addressing many research problems, including search service engineering and registration, efficient join methods for search services, and flexible query execution engines. The Liquid Query client application is primarily conceived to work within the Search Computing platform, but also represents a general interface for multi-domain exploratory search process, which can be applied to any federated search platform. Search Computing advocate an approach to multi-domain information seeking that is different from the generalized crawling and indexing of Web pages done by horizontal search engines. Multi-domain query processing exploits a data source selection and preparation phase performed at application configuration phase by an expert user. This splits the burden between the programmer, who must be able to wrap and register data sources, the expert user, who selects the data sources and the connections relevant to an application, and the end user, who queries such sources and explores their content. This is fully consistent with the recent advent of topical and semantic search engines, which however are limited to single-domain queries.

To demonstrate such a wide applicability, in this paper we describe how the Liquid Query client has been coupled with a general-purpose search service environment, namely the Yahoo! Query Language (YQL) [32]. As an additional contribution, we also highlight a few shortcomings of YQL with respect to multi-domain queries and result visualization. Ultimately, the reported experience shows that the Web service paradigm can be fruitfully employed by service brokers for building multi-domain exploratory search applications on top of existing Web data.

To illustrate the Liquid Query paradigm, throughout the paper we use a running example. A user plans a leisure trip and wants to search for upcoming *concerts* (described in terms of music type, e.g., Jazz, Rock, Pop, etc.) close to an attractive *location* (like a beach, lake, mountain, natural park, and so on), considering also availability of good, close-by *hotels*. Additionally, the user can expand the search to get information about available *restaurants* near the candidate concert locations, *news* associated to the event, *photos* of locations, and possible options to combine further *events* scheduled in the same days and located in a close-by place with respect to the first one.

The example shows the Liquid Query interaction style: there is an initial "seed" query (the selection of a concert of given gender in a place of given kind), which can be expanded by the user in various directions, inspecting new related objects, guided by curiosity. At any time, the user controls the progression of the search process, backtracks, and possibly turns to a totally different combination of items, which are part of the result set of the original query.

The paper is organized as follows: Section 2 discusses the related work; Section 3 presents the liquid query approach; Section 4 presents our implementation experience; and Section 5 concludes.

2. RELATED WORK

The design of Liquid Query draws from the achievements in a number of fields related to the development of interactive systems for information seeking and data visualization.

The design of novel search systems and interfaces is backed by several studies aimed at understanding how users' **search behaviour** on the Web. After the seminal work of Broder [8], other studies have investigated search behaviours by analysing search engine logs. In the first studies [29] queries were identified and classified manually by inspecting the logs, while recently several studies addressed automated classification and log mining, targeted to larger scale inference of the user search intent [20][23][24][33]. A review of approaches to search log data mining and Web search investigation is in [2] and [21].

A specific class of studies is devoted to **exploratory search**, where the user's intent is primarily to learn more on a topic of interest [25][34]. Such information seeking behaviour challenges the search engine interface, because it requires support to all the stages of information acquisition, from the initial formulation of the area of interest, to the discovery of the most relevant and authoritative sources, to the establishment of relationships among the relevant information elements [33]. An interesting distinction between complex and exploratory search is made in [1], where complex search is characterized by: multiple searches, possibly over multiple sessions and spanning multiple sources; a combination of exploration and more directed information finding activities; and the variation of the search goal during the search process. A number of techniques have been proposed to support exploratory search, and user studies have been conducted to understand the effectiveness of the various approaches (e.g., [22]).

Topic exploration is a case of complex and exploratory search, centred on the goal of collecting information on a subject matter of interest from multiple sources. The key challenge in topic exploration on the Web is the massive amount of disparate information available on each topic, which demands novel systems capable of constructing effective entry points for quickly grasping the essence of a topic and the possible directions for its exploration. Topic exploration has been traditionally served by

vertical search engines (e.g., WebMD, Mobissimo, Google News, CareerBuilder, MP3.com), which restrict the scope of the available topics to a specific domain.

Horizontal, i.e., cross-domain, topic exploration is a recent development. **Kosmix** [28] is a general-purpose topic discovery engine, which responds to keyword search by means of a topic page that summarizes the most relevant information on the subject associated to the search. Each topic page is defined manually and constructed by evaluating a set of *modules*, which wrap calls to Web services to extract information from deep Web data sources. Internally, Kosmix uses a mix of crawling and federated search. Query processing exploits a *taxonomy of topics*, comprising millions of nodes connected in a Direct Acyclic Graph, and a *Categorization Service*, which computes the nodes of the taxonomy that are most closely related to the user's query and the data sources in the system that can provide information about the query topic. When the relevant sources of information have been identified, the Kosmix engine performs the necessary data source queries and assembles the result in the topic page.

The organization of topical information is the goal of a variety of systems that employ different approaches and technologies to collect and layout the relevant information on a subject matter related to the user's search. Powerset (now incorporated into Microsoft's **Bing** [26]) specializes in extracting and organizing information from WikiPedia. A summary page is produced for each topic associated to a keyword search, which contains the essential facts and articles. **Hakia** [17] is another search engine capable of producing resume pages for topics associated with user's queries. Hakia exploits natural language processing techniques, specifically Ontological Semantics, for building a large ontology of concepts and correlations. A hybrid approach between vertical search engines and topic exploration systems is taken also by the latest versions of the mainstream, general-purpose search engines interfaces, which are enriching results lists with **extra elements derived from vertical or topical searches**. Examples of these extensions are Google Universal Search, Ask 3D, and Microsoft Live Search.

Search engine interfaces can be seen as tools for displaying and browsing a vast collection of documents, the subset of the Web corpus that is relevant to user's query. As such, they can benefit from all the techniques and design principles for **result presentation and refinement** produced by research on information visualization over the last three decades [4]. In the search domain, the standard visualization method relies on the "pan and zoom" principle: results are displayed at a low level of details, and the user can zoom on one element at a time. Improvements to result visualization rely on the taxonomical representations and on the alternative "focus and context" paradigm, whereby the user can select one element of a collection and get information on its context.

Structured object search refers to the ability of processing queries and presenting results that address entities or real world objects described in Web pages. A number of concurring factors have renovated the interest in Web information extraction also for large scale, horizontal search systems: the availability of good quality, edited content, most notably Wikipedia; the popularity of social content tagging (e.g., Flickr); and the advances in deep Web crawling and distributed data processing. Web-extracted **structure and knowledge** can be exploited mainly in two ways: by increasing the expressive power of the query language beyond keyword search, allowing queries to refer to the properties of data items; by exploiting the structure and additional knowledge available to enrich the presentation of the results, e.g., by suggesting correlations with other potentially interesting data items outside the scope of the user's query.

For example, the work on **WebTables** [9] reports on the massive extraction of structured data from the surface Web. **Google Squared** is an application from Google Labs demonstrating the interplay between structure and Web data [16]. The interaction can be started by an ordinary keyword search, but the results are collected in a table (called a *square*) featuring all the attributes relevant to the result items as columns headers. The initial square can be extended horizontally, by adding columns suggested by the system or by providing tentative column names. An existing square can be extended both vertically with new items and horizontally with new columns, for which the system tries and locates data matching the supposed semantics.

Google Fusion Table [15] allows one to upload a data table (e.g., a spreadsheet file) and join (or "fuse") the data in some column with other tables, either supplied by other uses or mined from the Web. Spreadsheet-like views of the base or joined tables can be defined, saved, shared with others, and commented collaboratively. Alternative visualizations are possible, depending of the type of data contained in a table, e.g., timelines and maps.

Faceted search is a technique allowing users to explore large object sets by filtering items based on multiple properties. Each facet corresponds to the possible value of an object's property (e.g., color, age, and so on). Facets have been used to support the exploration of document collections or query results [12] [22]. Faceted classification allows the assignment of multiple tags to an object, and the set of facets can itself be explored and organized in multiple ways (e.g., taxonomically). Dynamic Faceted Taxonomies [30] are the state of the art in the field and demonstrated good acceptance by the users [11][13].

3. THE LIQUID QUERY APPROACH

In this section, we describe the Liquid Query approach for the creation, manipulation, and extension of multi-domain queries and results, according to the "search as a process" paradigm. Among the approaches discussed in Section 2, Liquid Query adds to such systems as Google Squared and Google Fusion Tables the capacity of joining partial results from different interrelated data sources and of enabling the user to explore a multi-domain search space, by means of such commands as the selective request of more results from specific domains. Due to the search service federation approach, the elements in the tabular result set of Liquid Query are in fact *object combinations*, and each object in a combination is associated with the search service that has returned it and can be manipulated individually, thus enabling query refinement and result manipulation primitives not available in those systems that adopt flat results structures.

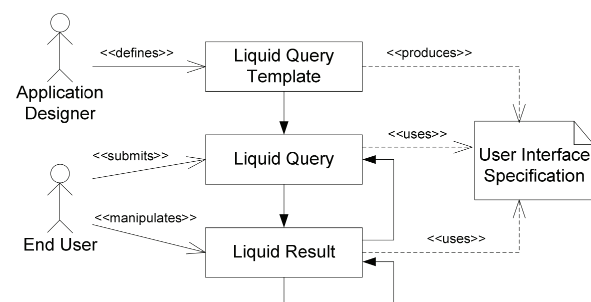

Figure 1. Overview of the Liquid Query and Result lifecycle.

3.1 Overview of the Liquid Query Lifecycle

Liquid queries act upon service registries, which hide the complexity of service implementation behind simple interfaces. Although in this paper the underlying search services are defined programmatically in YQL, liquid queries could be easily configured through high-level visual interfaces (an early example of mashup-based interface was presented in [6]).

The query life cycle, shown in Figure 1, consists of four steps:

1. **Application configuration phase:** The application designer (or expert user) defines a liquid query template for a specific application, which entails: (1) the specification of the search services that will provide results for each domain of interest; and (2) the definition of user interface aspects.
2. **Query submission phase:** Based on a query template, the end user submits an initial liquid query, which complies with the template specified by the designer.
3. **Query execution phase:** After executing the query, a liquid result is produced and delivered to the user interface. Here, results are rendered in the interface according to the user default preferences (e.g., in terms of sorting, grouping, and visualization choices).
4. **Result browsing phase:** The result set can be manipulated through appropriate interaction primitives, which update either the liquid result or the liquid query. If the interaction only involves rendering available data in different way, the query execution backend is not invoked. Conversely, some actions (e.g., the request of more results from a service or the expansion of the query with an additional service) require the interaction with the backend and the partial or total re-execution of the query.

In summary, the main objects involved in the query lifecycle are:

- **Liquid query template**: a set of input forms and controls for query submission, a set of parameters for result presentation, and a set of pre-designed query expansion targets, i.e., new services that can be invoked to extract information on other domains related to the original query.
- **Liquid query**: a concrete query upon services that is executed by the query execution backend.
- **Liquid result**: a list of tuples (combinations of objects) extracted by the query, conforming to a result schema.
- **Interaction primitive**: a specific user command that produces a side effect either on the liquid result (e.g. grouping by given attributes, selecting or re-ordering tuples) or on the liquid query (e.g. requesting more tuples, or expanding the result to new domains).

3.2 Detailed Liquid Query Specifications

In the following we provide a detailed definition of the relevant Liquid Query concepts within the running case, organized by lifecycle phase. For presentation purposes, we mainly show tabular representations of the result sets; nonetheless other, richer visualization techniques are supported (e.g., parallel coordinates [19]).

3.2.1 Application Configuration Phase

At the first stage of the application life cycle, the designer chooses the set of services required by the query and defines the structure of the Liquid Query template. A **service interface** is a high level representation of a data provisioning service which comprises input parameters, output parameters, and ranking attributes. A **liquid query template** is composed of:

- a set of service interfaces;
- a set of connection patterns for joining the involved service interfaces;
- a set of selection or join predicates;
- a set of possible ranking, grouping, and clustering attributes that can be applied on the extracted result set;
- a set of positive integer values K that represent the possible sizes for the result pages;
- a set of available query expansions, defined next.

A **query expansion** allows users to dynamically select some objects in the result set and join them with information provided by another service interface, called the *expansion target*. The join exploits connection paths, made of *join attributes*, defined at application configuration time and stored in the service repository; every object selected in the results set provides input values to the join algorithm, in correspondence to its join attributes. To compute the query expansion, each selected object is considered in turn and the target service interface is called with input parameters taken from the join attributes of the current object. Values extracted from the target service interface are then joined with the object that provided the input values and displayed alongside the results of the other queried services; in this way, a new service is added to the liquid result. Extracted objects can be used for further query expansions, thereby allowing a result to be progressively enlarged.

In the example, we assume that the initial query comprises *Concert* and *Hotel* objects, and we allow the user to subsequently expand his search to other *Events*, to related *News*, to *Restaurants*, and to *Photos* of the venues. Result ranking is by distance for *Concert* (with respect to the user-specified location) and for *Restaurant* and *Hotel* (with respect to the Event location). *News* and *Photos*, instead, are ranked by date.

3.2.2 Query Submission Phase

At runtime, the user fills in the Liquid Query template with actual parameters. A **query invocation** is a sequence of input values, provided by the user, associated with the input places of the query template. From the end user point of view, the query submission consists of an interface representing the list of input parameters to the query defined by the application designer at application configuration time. Figure 2 depicts the input interface for the case study, where the user fills in parameters for *Concert* and *Hotel*, e.g., Jazz concerts taking place in San Francisco in December, and good hotels (having an average rate of at least three out of five) within 2 miles from the venue.

3.2.3 Query Execution Phase

At query execution time, the configured Liquid Query is performed and a result set consisting of object combinations that satisfy the specified selection conditions is produced.

Figure 2. Multi-domain query submission interface

	Concert						Hotel			
	Name	Address	City	Dist.(Mi.) ▲	Start Date	❖	Name	Address	Dist.(Mi.) ▲	Ratin
☐	Jazz at the Rrazz Featuring Bruce Forman - Guitar With Mike Greensill	222 Mason St	San Francisco	0.76	12/21/2009		Hilton San Francisco	333 Ofarrell St	0.05	4.5
☐	Jazz at the Rrazz Featuring Bruce Forman - Guitar With Mike Greensill	222 Mason St	San Francisco	0.76	12/21/2009		Hotel Bijou	111 Mason St	0.07	4.5
☐	Jazz at the Rrazz Featuring Bruce Forman - Guitar With Mike Greensill	222 Mason St	San Francisco	0.76	12/21/2009		Serrano Hotel	405 Taylor St	0.12	4.5
☐	Ahmad Jamal	1330 Fillmore Street	San Francisco	0.76	12/10/2009		Metro Hotel San Francisco	319 Divisadero St	0.71	4.5
☐	Ahmad Jamal	1330 Fillmore Street	San Francisco	0.76	12/10/2009		Nob Hill Hotel	835 Hyde St	0.94	5
☐	Ahmad Jamal	1330 Fillmore Street	San Francisco	0.76	12/10/2009		Holiday Inn San Francisco-Civic Center Hotel	50 8th St	1.02	4.5
☐	Gaucho, Mitch Marcus Sessions	853 Valencia St	San Francisco	1.24	12/02/2009		Phoenix Hotel	601 Eddy St	1.67	4.5
☐	Gaucho, Mitch Marcus Sessions	853 Valencia St	San Francisco	1.24	12/02/2009		The Powell Hotel	28 Cyril Magnin St	1.88	4.5

Figure 3. Liquid Query multi-domain resultset table

The **liquid result schema** defines the format of the result set, in terms of projected attributes (i.e., shown columns), ordering attributes, clustering attributes, grouping attribute, and expansions. Notice that users can dynamically change the order within the list of selected services and within the lists of their projection attributes and ordering attributes, as well as the subsequent query expansions. A **liquid result instance** is a tuple that is extracted by the Liquid Query backend and shown as a row in the tabular representation of the result schema. In particular, each instance must comply with the schema, i.e., show all the projected attributes in the correct order. A **liquid result page** is a set of liquid result instances that are shown together in the user interface.

The result set of the running example (shown in Figure 3) consists of a table of combinations of *Concert* and *Hotel* objects.

3.3 Results Browsing

When browsing the result set, the end user is offered interaction primitives that refine or change the *shape o*f the *quer*y, the content of the result set, and the result layout. Such primitives may need to access the server-side (Remote Interaction Primitives), or could impact only the client-side version of the result set (Local Interaction Primitives). In addition, liquid queries support three classes of interactions, which use classic data visualization and personalization techniques: data calculation commands, visualization perspectives, and query management primitives. Some of the interaction primitives apply at attribute level, some at service level, and some at whole result set level.

3.3.1 Remote Interaction Primitives

Remote interaction primitives require the client to ask the server for some computation, at the purpose of producing new results. Remote interaction primitives include:

- **Expand (Target Service, Selected Tuples):** the operation expands the result schema by adding one new target service and joining it with selected tuples. The expansion causes a set of exact queries to the expanded service interface, on the values selected by the user. If the expansion requires additional inputs, a dialog box is shown to the user for submitting the needed values. As an example, let's assume that after the search performed over concerts and hotels the user is tempted by two choices, "Bruce Forman" coupled to the "Hilton San Francisco", and "Ahmad Jamal" coupled to the "Metro Hotel San Francisco". Then, the user may want to complete the vacation plan by choosing a nearby restaurant and another concert. This can be done by selecting the two concerts and performing a first expansion on "restaurants"; and then by performing a second expansion from the same concerts but using the "other events" search service. Distances of restaurants and secondary events are measured with respect to the original selected events. The expansion results, shown in Figure 4, are combined with the original objects so that the user can collectively look at a row of the liquid result as a combination of two concerts plus one restaurant plus one hotel, and still recognize and compare the two original alternatives. Adding and dropping search services to the liquid result corresponds to adding and dropping search facet groups, on which filtering, ordering, grouping, and further expansions can be applied.
- **MoreAll**: the operation loads additional tuples from all the selected services in the currently specified query (excluding extensions); this command is typically executed when the user asks for more data about her information need as a whole.
- **MoreOne (Service)**: the operation asks for additional tuples from a specific service interface in the currently defined query. This command is typically executed when the user asks for more data from a specific domain (e.g., more hotels or restaurants), leaving un-altered the others search service results. For instance, the user might be happy about the found concert, but may want more options for restaurants. The operation extracts a given number of new results for the chosen service, but the number of final answers cannot easily be estimated, as it depends on the selectivity of the join operation.

3.3.2 Local Interaction Primitives

Local interaction primitives can be performed on the client without server involvement. They include:

- **Cluster (Attribute)**: the operation changes the visual appearance of a result list by clustering adjacent tuples on a specific attribute and hiding duplicate values. The default presentation applies left-to-right clustering by attribute to the columns of all services. For instance, no clustering is applied to results in Figure 3, thus attribute values are repeated. Instead, Figure 4 shows a clustered result page, where duplicates are not repeated.
- **Uncluster (Attribute)**: the operation undoes the cluster operation; sorting and grouping are not modified. A shortcut function at service level removes all the clustering attributes for a given service.
- **Sort (Attribute)**: the operation sorts the currently displayed results w.r.t. the values of an attribute (e.g., results in Figure 3 are sorted by Distance of the concert with respect to the queried location and by Distance of hotels with respect to the concert). Notice that clusters may appear or disappear in the result list according to the new adjacencies.
- **Unsort (Attribute)**: the operation undoes a sort operation.
- **Roll-up (Attribute):** the operation removes a projection attribute, i.e., hides a currently visible attribute from the result schema. If the attribute removal introduces duplicated elements in the result table, those elements are eliminated from the list. If the attribute is used for ordering, grouping, or clustering, it is removed also from the respective sets.

Concert			Restaurant			Other Events				Hotel		
Name	Dist.	❖	Name	Distance	❖	Name	Distance	Start Date	❖	Name	Distance	❖
Jazz at the Rrazz Featuring Bruce Forman - Guitar With Mike Greensill	0.76		New Delhi Restaurant	0.14		Riverdance	0.22	12/27/2009		Hilton San Francisco	0.05	
						Tricks of Light: Silent Films	0.73					
			First Crush Restaurant Wine Bar & Lounge	0.15		Judy Butterfield Home for the Holidays Show	0.76	12/27/2009				
						The Howard Stone Show!	0.91					
Ahmad Jamal	0.76		Arang Restaurant	0.12		Bill Fontana: Spiraling Echoes	0	12/11/2009		Metro Hotel San Francisco	0.71	
						Gods and Goods: Global Art and Trade	0.21					
			Rasselas on Fillmore Jazz Club	0.14		1330 Fillmore Street	0.76	12/13/2009				
						The Edward Albee's Who's Afraid of Virginia Woolf? Presented by Actors	0.85	12/19/2009				

Figure 4. Clustered liquid result, with first two instances of Concert expanded with Restaurants and Other Events

- **Drill-down (Attribute):** the operation adds a new projection attribute to the results. The new attribute must be taken from the list of available attributes of the service not yet displayed. Note that new instances could appear in the result set, by effect of showing elements that appeared as duplicates without that attribute.
- **Filter (Condition):** the operation reduces the number of results in the result list by locally applying a filtering condition on the displayed attributes.
- **RemoveFilter (Condition)**: the operation undoes the filter operation.

3.3.3 Data Calculation Primitives

Local primitives include a set of options for applying calculations on the data, at the purpose of easing the interpretation of the information by the user. Manipulation primitives let the user add new calculated columns to the result table. Arithmetic operations on numeric values, concatenation of strings, and basic aggregated functions between attributes of the same result instance (maximum and minimum) are available. For instance, the user may want to calculate the total distance between the Event, the Restaurant, and the Hotel of each result tuple.

3.3.4 Data Visualization Primitives

Data visualization primitives introduce different visual representations of the extracted data, to get a more immediate vision on the results. Examples include the representation of:

- column data in a chart (pie, bar, and so on);
- addresses or geographical coordinates on a map;
- cloud maps of the concepts available in the result set, with an indication of the respective weights and relationships.

At this purpose, Liquid Query exploits state of the art visualization tools by adopting a mashup approach. As an example, Figure 5 shows the map of locations associated with the first concert listed in Figure 4. The large map shows the venue of the concert and some hotels, restaurants and additional events. The zoomed image shows the event details and the positions of the two closest restaurants belonging to the answer.

3.3.5 Query Management Primitives

The user can manage the query and the result set through the following query management commands:

- Export the current result set in various formats;
- Save the current query status;
- Open a previously saved query;
- Define a public permanent link to the current result set view, that can be emailed or linked from web sites;

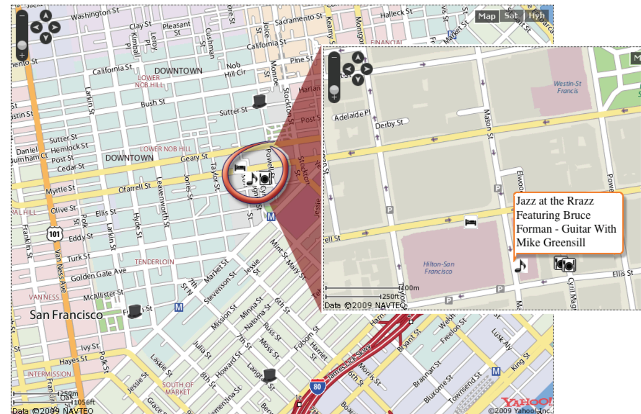

Figure 5. Example results visualization in a map

- Store the query as preferred bookmark on social bookmarking systems (delicious, digg, and others);
- Navigate the query history through the buttons Previous, Next, First, and Last.

3.3.6 Interaction Primitives in the User Interface

At the user interface level, the result table is enriched with interaction menus, at various levels:

- **At column level** (Figure 6.a), a contextual menu lets the user perform operations on the respective attribute: apply filters, grouping, and clustering; select visualization options, like charts or maps of the column data; change the sort status of the column (Asc, Desc, None); move the column in the table; roll-up on that attribute (i.e., hide the respective column).
- **At service level** (Figure 6.b), a contextual menu lets the user perform operations on the entire object, namely: change the connection path that joins the services, expand to new services; drill-down on hidden attributes; move the service in the table; apply clustering to all the columns of the service; ask for more results from the specific service; and choose the visualization options for the service results.
- **At result set level** (Figure 6.c), a set of options are available, including: creating new columns as derived values starting from the available ones within the whole result set (the new column will not belong to any service); asking for more result instances for the overall set; and choosing the visualization options for the entire result set.

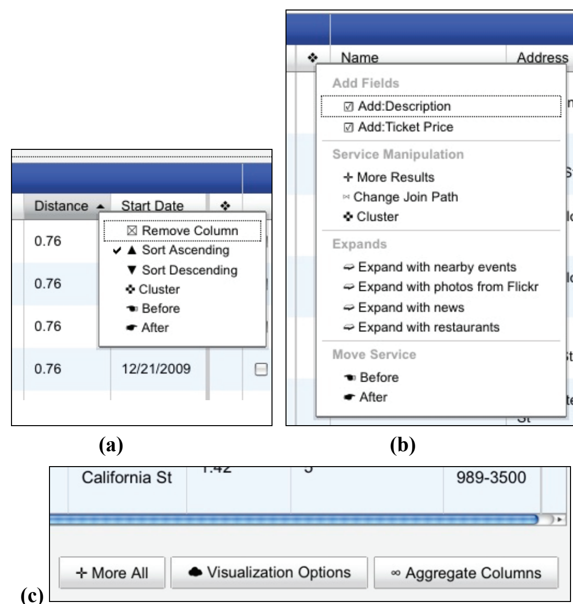

Figure 6. Column-level menu (a), Service-level menu (b) and result set commands (c)

4. IMPLEMENTATION EXPERIENCE

In this section we describe our implementation of the Liquid Query concepts upon a specific service provider and service manipulation language. Our experience consists of:

- Defining the Liquid Query primitives in terms of mappings to the available data access API and language;
- Selecting the technologies that fit best the needs of the project;
- Designing and engineering an architecture for the system that supports state of the art design frameworks, rich client interactions, and deployment over the cloud for improving scalability through distribution and replication;
- Implementing some showcase multi-domain search applications over the prototype platform.

The prototype of our system, implementing the running example presented in this paper, can be found at:

http://www.search-computing.it/LiquidQueryYQL.html

In the following subsections we shortly describe the search backend system, namely YQL; we discuss the features of the prototype and our architectural solution; we present the mapping of Liquid Query primitives to the actual system; and describe the configurable features of the system. A similar implementation could be done on top of the Google Base API [14] or of any OpenSearch-compliant system [27].

4.1 Yahoo! Query Language

Yahoo! Query Language (YQL) is a language and a platform that let Web applications query, filter, and combine data from different sources across the Internet through SQL-like statements [32]. For instance, the following YQL statement retrieves the first 100 cat photos from Flickr.

```
SELECT * FROM flickr.photos.search(0,100)¹ WHERE
text="cat"
```

[1] *(0,100)* is a YQL operator that limits the number of returned results (*limit*=100), starting from a specific offset (*offset*=0).

YQL is available as a REST Service that can be invoked through HTTP GET, passing the YQL statement as a URL parameter. For instance:

```
http://query.yahooapis.com/v1/public/yql?q= YQL query
```

When it processes a query, the YQL service accesses a data source on the Internet according to a given access API, transforms its data, and returns the results in either XML or JSON format. YQL can access several types of data sources, including Web services, REST API and Web content in formats such as HTML, XML, and RSS, wrapped as Open Data Tables[2].

4.2 System Architecture

The implemented prototype consists of a fully functional architecture that provides Liquid Query features upon the YQL backend search interface.

Figure 7 depicts the internal architecture of our system, designed as a three-tier distributed Rich Internet Application (RIA). RIAs bring to Web interfaces fluid user interactions, which enhance the user experience by enabling client-side processing, local data storage and manipulation, off-line functioning and asynchronous server communications; in addition, a rich client-side interface allows the definition of a light-weighted, scalable REST server architecture, which supports node distribution and replication.

The LQ-Server exposes a set of *data service* interaction methods, such as *query, extend, more*, etc.: each method implements the business logic required to mediate the interaction between the LQ-Client and the data back-end. Moreover, the LQ-Server provides several *application utility* methods, which provides the functionalities required to instantiate and configure a LQ application, and support its execution life-cycle. To this purpose, a *service repository* component interacts with a persistent repository (the distributed, document-based CouchDB database) in order to retrieve and manage the application configuration files, such as the service interfaces and the Liquid Query template configurations. For instance, since our example leverages the YQL APIs, for each YQL table (*local.search*, *upcoming.events*, *flickr.photos*, and *search.news*) the service repository contains both the native table descriptor and the associated *service interface*[3], as described below.

```
Local.search(businessType[I]⁴, category[I],
    minimumRating[I], location[I], radius[I],
    title[O], address[O], city[O], state[O], phone[O],
    latitude[O], longitude[O], averageRating[O][R],
    totalRatings[O], distance[O][R], url [O],
    categories*{category}[O])

Upcoming.events(location[I], radius[I], minDate[I],
    maxDate[I], eventCategory[I]⁵, locationType[I],
    searchText[I], name[O], description[O], startDate
    [O][R], endDate[O][R], latitude[O], longitude[O],
    distance[O][R], ticketPrice[O][R], url[O], address
    [O], venueCity[O], venueZIP[O])

Search.News(query[I], age[I], title[O][R], date[O][R],
    source[O][R], url[O], abstract[O])

Flickr.Photo(query[I], minDate[I], maxDate[I],url[O],
    title[O], description[O], date[O][R])
```

[2] http://datatables.org

[3] Notation: [I] input, [O] output, and [R] ranking attributes.

[4] `businessType` can be set to "hotel" or "restaurant".

[5] `eventCategory` can be set to "music" (as in the initial query of the running example) or other categories. The complete taxonomy is available at: `http://query.yahooapis.com/v1/public/yql?q=select%20*%20from%20upcoming.category`

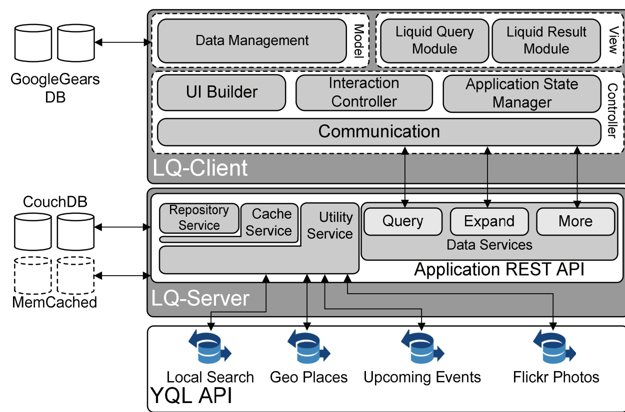

Figure 7. Liquid Query-YUI-YQL Architecture

It is then up to the *application utility* module to lift query requests coming from the client to YQL queries and, likewise, to lower YQL response to the common client data format. All methods are exposed as REST APIs, where the state of interaction is maintained both by the LQ-Client and by a distributed memory object caching system (MemCached); the latter has a twofold purpose: to store information about user sessions and to cache the results of remote query executions and joins.

The LQ-Client component layer, instead, runs inside a Web browser as a JavaScript application written according to the Model-View-Controller design pattern and exploiting the libraries and functionalities offered by the Yahoo! User Interface (YUI) libraries[6] and by Google Gears[7]. The client *view* comprises the graphical objects and presentation properties specific of the LQ application, like the *Liquid Query* and the *Liquid Result* modules, responsible of rendering the liquid query, its results, and its available interaction commands.

The client-side *controller,* is composed of (1) an *interaction controller* module (devoted to the orchestration of the client-side components upon application initialization or user interaction); (2) a *communication* module (handling the communication with the LQ-Server); (3) an *ui-builder* module (responsible for the assembling of the client *View* module according to the downloaded application configurations); and (4) an *application-state-manager*, which keeps track of the interaction state, eventually restoring it on demand, thus supporting features like navigation history and bookmarking [7].

The client *model* is consists of a *data-manager* component, which modifies client data after each interaction, possibly leveraging a client data store such as Google Gears DB. The *data-manager,* for instance, performs operations like local filtering and sorting, aggregation of result combinations, synchronization with a client persistent storage to enable off-line usage, etc.

4.3 Configurability of the Platform

The LQ system architecture has been designed to ease the development of LQ applications, shifting complexity from application development to application configuration. Configuration covers both server and client aspects, which revolve around the description of the application's service interfaces and joins attributes.

At server side, the data service methods are provided with information about the concrete data back-end: each YQL table is therefore described by its signature – the list of required inputs and produced outputs, and by its endpoint URI; in addition, the configuration contains information about the available join attributes and query expansions, so to enable their dynamic execution upon client command.

At client side, instead, the main focus is on the specification of the Liquid Query and Liquid Result modules; the client configuration contains: (1) the schema of the initial query, including its fields, mandatory and default values, rankings, and the list of enabled join attributes and selection predicates; (2) the schema of the Liquid Result interface, which comprises the list of displayed (and displayable) attributes, the set of allowed query expansions, and the set of the enabled local and remote interaction, manipulation, and visualization operations; and (3) details about the client look and feel, such as the applied style-sheets and visual identity.

4.4 Mapping Liquid Query to the Platform

The mapping of Liquid Query features onto YQL faced a few mismatching issues between the expressive power of the two languages. Mappings dealt with the four class of Liquid Query primitives discussed in Section 3.

Remote Query Interaction Primitives have been mapped to YQL commands, with the addition of locally computed join operations. *Local Resultset Interaction* and *Local Manipulation Primitives* have been managed at client side, by leveraging the GoogleGears DB. *Data Visualization Primitives* in the current prototype exploit results mashups with visualization libraries (Yahoo Maps and PlaceMaker API[8], Google Visualization API[9], and others) and applications (like GeoMaker[10]). *Query Management Primitives* have been delegated to the Application State Manager component of the architecture that keeps track of the user interaction states and retrieves old ones at need.

A short discussion of the mapping of each Remote Query Interaction primitive follows. The other primitives are less crucial with respect to the YQL implementation, and therefore we skip their discussions due to space limitations.

4.4.1 Initial query

Considering that the main objective of the platform is to join search service results, two main issues arise: (1) the main drawback is the lack of the join (and Cartesian product) operator in YQL. This is a critical point since joins are the fundamental operator for multi-domain queries. Therefore, an ad hoc strategy for implementing joins has been implemented. (2) YQL limits the maximum number of returned results and applies sorting and filtering operations to the independent result sets returned by the original services (and not to the joined results, as Liquid Query would need). Given that it is not possible to preventively download all the population of the involved data services, the problem arises of deciding how and when to ask for new data to the services and how to implement ranking.

The solution to the two abovementioned problems consists in devising a join strategy for deciding the sequence of YQL service invocations and for performing the joins between the extracted

[6] http://developer.yahoo.com/yui/

[7] http://gears.google.com/

[8] http://developer.yahoo.com/maps/, http://developer.yahoo.com/geo/placemaker/

[9] http://code.google.com/apis/visualization/

[10] http://icant.co.uk/geomaker/

result sets (plus subsequent filtering, grouping, and sorting). Three possible join strategies could be envisioned:

(1) Independent extraction of elements from each service and implementation of a complete join algorithm.

(2) Extraction of elements that match each other (exploiting the *where ... in ...*YQL primitive). For instance, the initial query that finds information about hotels located near Jazz concerts taking place in San Francisco in December is built starting from the following YQL statement:

```
SELECT title, city, Rating.AverageRating,
  latitude, longitude
FROM Local.Search (0,100)
WHERE (latitude,longitude) IN
   (SELECT latitude, longitude
    FROM Upcoming.Events
    WHERE category_id="1" AND
        min_date="2009-12-01" AND
        max_date="2009-12-31" AND
        location =  "San Francisco" AND
        search_text="Jazz
   ) AND query="Hotel" AND radius = "0.5"
     AND Rating.AverageRating > 2
     AND Rating.TotalRatings > 30
```

This extracts the hotels located close to the Jazz concerts satisfying the query conditions. Then, the concerts themselves are retrieved and joined with the hotels based on the respective latitude and longitude coordinates. This strategy would require a customized join and ranking function based on the distance between locations.

(3) Implementation of a pipe join algorithm [5]. In details, we first perform a YQL query on the most selective service (e.g., the Upcoming.Events service), thus returning its population. Then, for each tuple in the result set (Event[i]), we perform an invocation to the joined service (e.g., the Hotel service), using a filtering condition on the defined join attribute (location of the event close to location of the hotel). Each extracted tuple is therefore extended with a unique identifier and joined elements are matched through a foreign key attribute, referring to the associated tuple.

Solution (1) is obviously inefficient, since it needs to extract large amounts of data from each service, and moreover requires high computational loads to build the joined tables and the distances between restaurants and events. Solution (2) and solution (3) grant a twofold benefit: on one hand, they retrieve less information, thus speeding up the data retrieval; on the other hand, the calculations to be performed *a posteriori* on the data are less intensive. However, solution (2) is not of general applicability since it only supports exact matching between elements, which represents a very rare option on the web. Therefore, in our prototype we implemented pipe joins, i.e. solution (3). Notice however that none of the three methods can provide any guarantee on the number of actually matching elements in the join; therefore, when the initial interaction produces a number of answers which is insufficient to fill up the result pages as expected by the query, further user interactions with the services are needed.

4.4.2 More All

The *More All* primitive is implemented as follows: for each service involved in the join, we increment the *limit* and *offset* options on all the services involved in the initial query, thus increasing the overall size of the result sets returned by each service. Then, we apply the original join operation again. For instance, the query on the upcoming event service is modified as follows:

```
SELECT venue_city, venue_address,
  start_date,end_date, latitude, longitude
FROM Upcoming.Events (100,200)
WHERE category_id="1" AND min_date="2009-12-01" AND
  max_date="2009-12-31" AND search_text= "Jazz" AND
  location =  "San Francisco"
```

Since there is no guarantee on the global ranking optimality of the initial result set, new high-ranked combinations could be generated also by joining new results with old ones.

4.4.3 More One

The *More One* operation consists in increasing the result set of the selected service. The implementation of this command depends on the position of the target service of the *More one* operation with respect to the structure of the original query. In case of just two services, if the target service is the last in the pipe join (e.g., the Hotel service), the *More one* command simply consists in repeating the loop on the first service (e.g., UpcomingEvents), and invoking the last service with incremented *limit* and *offset* options; for instance, the LocalSearch service query of the running example would be modified as follows:

```
SELECT title, city, Rating.AverageRating,
  latitude, longitude
FROM Local.Search (0,200)
WHERE (latitude,longitude) IN
   (SELECT latitude, longitude
    FROM Upcoming.Events
    WHERE category_id="1" AND min_date="2009-12-01"
       AND max_date="2009-12-31"
       AND  location = "San Francisco"
       AND  search_text="Jazz
   )AND query="Hotel" AND radius = "0.5"
    AND Rating.AverageRating > 2
    AND Rating.TotalRatings > 30
```

If the target service of the *More one* operation is the first in the pipe join (e.g., the UpcomingEvent service), the *More one* command consists in: (1) invoking again the first service with incremented *limit* and *offset* options; (2) looping on the new instances and extracting the matching tuples for the second service (e.g., the Restaurant service); and (3) keeping only the second service results (e.g., Restaurants) that already existed in the old result set.

4.4.4 Expand

For all the tuples selected by the user (through a checkbox widget), the *Expand* operation invokes the target service with a pipe join approach. Actually, a left join operation is applied, with the expanded service results constituting the right-side table, so that the existing result set are preserved and extended with the new tuples, when available. The final effect is that no expansion tuples are available for non-selected result object and for those selected objects for which no matching tuples are found. For instance, the initial query can be expanded to include information about restaurants located near the concert's location as follows:

```
SELECT title, city, Rating.AverageRating,
  latitude, longitude
FROM Local.Search (100,200)
WHERE latitude = $LAT_CON AND longitude =  $LON_CON
  AND radius = "0.5" AND Rating.AverageRating > 2
  AND Rating.TotalRatings > 30 AND query="Restaurant"
```

where *$LAT_CON* and $*LON_CON* are the coordinates of the selected concert.

5. CONCLUSIONS

In this paper we described Liquid Query, a paradigm that exploits the power of underlying search services and provides the user with a multi-domain exploratory search environment. Liquid Query combines search interfaces and data visualization facilities

to improve the user experience and the productivity in information seeking. The approach helps users in performing exploratory search tasks, without the need of different online tools for composing heterogeneous information.

We demonstrated the feasibility of the approach with a prototype implemented over the YQL framework. Future work include thorough user evaluation studies, which will allow us to validate the effectiveness and acceptance of the approach, the design of additional interaction primitives, the implementation of visual design tools supporting creation of Liquid Query templates, the integration of advanced data visualization components, and the porting to different backend search systems (including the Search Computing backend engine and the Google Base platform). In particular, the integration with the Search Computing engine, currently under development, will enable the Liquid Query interface to exploit sophisticated top-k query optimization and execution algorithms, currently under implementation in the platform.

6. ACKNOWLEDGEMENTS

This research is part of the *Search Computing* (SeCo) project, funded by *European Research Council*, under the *IDEAS Advanced Grants* program. We thank all the project contributors, the advisory board members, and Michele Follo, Chiara Pasini, Andrea Vaccarella, and Riccardo Volonterio for their effort in the implementation of the prototype system.

7. REFERENCES

[1] Aula, A., and Russell, D.M. Complex and Exploratory Web Search. ISSS: Information Seeking Support Systems Workshop (Chapel Hill, NC, USA, June 2008), 23-24.

[2] Baeza-Yates, R.A. Applications of Web Query Mining. ECIR: European Conference on Information Retrieval, 2005, Springer LNCS 3408, 7-22.

[3] Barbosa, L., and Freire, J. Siphoning hidden-web data through keyword-based interfaces. SBBD, 19th Brazilian symposium on databases, 2004, 309-321.

[4] Bederson, B.B., and Shneiderman, B. *The Craft of Information Visualization: Readings and Reflections*. Morgan Kaufmann, 2003.

[5] Braga, D., Campi, A., Ceri, S., Raffio, A. Joining the results of heterogeneous search engines, Information Systems, Vol. 33, Issues 7-8, 2008, Pages 658-680.

[6] Braga, D., Ceri, S., Daniel, F., Martinenghi, D. Mashing Up Search Services. IEEE Internet Comp. 12(5) (2008), 16-23.

[7] Brambilla, M., Cabot, J., Grossniklaus, M. Modelling safe interface interactions in web applications. In Conceptual Modeling - ER. Springer LNCS Vol. 5829, 387-400. (2009)

[8] Broder, A. A taxonomy of web search. SIGIR Forum, 36(2):3–10, 2002.

[9] Cafarella, M. J., Halevy, A., Zhang, Y., Wang, D. Z., and Wu, E. WebTables: Exploring the Power of Tables on the Web. In VLDB (Auckland, NZ, 2008), 538-549.

[10] Ceri, S., Brambilla, M. (eds.). Search Computing Challenges and Directions. Springer LNCS vol. 5950, March 2010.

[11] Clusty. http:// www.clusty.com/.

[12] Dash, D., Rao, J., Megiddo, N., Ailamaki, A., and Lohman, G. 2008. Dynamic faceted search for discovery-driven analysis. 17th ACM Conference on information and Knowledge Management, CIKM '08. (Napa Valley, California, USA, October 26 - 30, 2008). ACM, 3-12.

[13] DBLP Faceted Search. http://dblp.l3s.de/.

[14] Google Base API. http://code.google.com/apis/base/.

[15] Google Fusion Tables. http://tables.googlelabs.com/.

[16] Google Squared. http://www.google.com/squared.

[17] Hakia. http://hakia.com/.

[18] Hunch. http://www.hunch.com/.

[19] Inselberg, A. The Plane with Parallel Coordinates. Visual Computer 1 (4). Springer: 69–91. (1985)

[20] Jansen, B.J., Booth, D.L., and Spink, A. Determining the user intent of web search engine queries. WWW Conf. 2007 (Banff, Canada): 1149-1150.

[21] Jansen, B.J., Pooch, U.W. A review of Web searching studies and a framework for future research. JASIST (J. of Am. Soc. Inf. Science and Techn.) 52(3): 235-246 (2001)

[22] Kules, B., Capra, R., Banta, M., and Sierra, T. What do exploratory searchers look at in a faceted search interface? JCDL, Joint Conference on Digital Libraries(2009). 313-322.

[23] Kumar, R., and Tomkins, A. A Characterization of Online Search Behavior. Data Engineering Bullettin, June 2009, 32(2), 3-11.

[24] Lee, U., Liu, Z., and Cho, J. Automatic identification of user goals in Web search. WWW 2005 (Chiba, Japan): 391-400.

[25] Marchionini, G. Exploratory search: from finding to understanding. Communications ACM 49(4): 41-46 (2006).

[26] Microsoft Bing. http://www.bing.com/.

[27] OpenSearch. http://www.opensearch.org/.

[28] Rajaraman, A. Kosmix: High Performance Topic Exploration using the Deep Web, VLDB 2009 (Lyon, France, 2009), Proceedings of VLDB 2(2): 1524-1529.

[29] Rose, D.E., and Levinson, D. Understanding user goals in Web search. 13th WWW Conf. (New York, 2004), 13-19.

[30] Sacco, G. M., and Tzitzikas, Y. *Dynamic Taxonomies and Faceted Search: Theory, Practice, and Experience*. Series: The Information Retrieval Series, Vol. 25, Springer 2009.

[31] Shafer, J.C., Agrawal, R., and Lauw, H.W. Symphony: Enabling Search-Driven Applications, USETIM (Using Search Engine Technology for Information Management) Workshop, VLDB (Lyon, Aug. 2009).

[32] Yahoo! Query Language. http://developer.yahoo.com/yql/

[33] White, R. W., and Drucker, S. M. Investigating behavioral variability in web search. 16th WWW Conf. (Banff, Canada, 2007), 21–30.

[34] White, R.W., Roth, R.A. *Exploratory Search. Beyond the Query–Response Paradigm.* Synthesis Lectures on Information Concepts, Retrieval, and Services Series, Gary Marchionini (ed.), vol. 3. Morgan & Claypool, 2009.

[35] Wolfram Alpha. http://www.wolframalpha.com/.

[36] Wu, W., Yu, C., Doan, A., and Meng, W. An interactive clustering-based approach to integrating source query interfaces on the deep Web. ACM SIGMOD (Paris, France, 2004), 95 – 106, ISBN:1-58113-859-8.