iMeMex: From Search to Information Integration and Back

Jens Dittrich
Saarland University

Marcos Antonio Vaz Salles
Cornell University

Lukas Blunschi
ETH Zurich

Abstract

In this paper, we report on lessons learned while building iMeMex*, the first incarnation of a Personal Dataspace Management System (PDSMS). In contrast to traditional keyword search engines, users may not only search their collections with* iMeMex *but also semantically integrate them over time. As a consequence, the system may improve on precision and recall of queries in a pay-as-you-go fashion. We discuss the impact of several conceptual and architectural decisions made in* iMeMex *and outline open research challenges in combining search and information integration in order to build personal dataspace management systems.*

1 Bringing Information Integration to Search in Personal Information

Personal Information Management (PIM) is the process performed daily by users to collect, store, organize, and access their collections of digital objects [3]. Personal information is composed of a heterogeneous data mix, including files, folders, emails, office documents, contacts, images, music, calendar items, videos, among others. In addition to heterogeneity, an added dimension regarding personal information is that it is also distributed, being stored in a wide variety of data sources such as file systems, network shares, external drives, iPods, cell phones, email servers, and on the Web.

There have been two extreme solutions to offer a unified query service over a set of heterogeneous and distributed data sources. At one extreme, traditional search engines allow users to pose simple keyword queries over unstructured sources. XML-IR search engines go one step further and allow users to also pose path queries over semi-structured sources. The semantics of those keyword or path queries are not always precise, meaning that the quality of query answers is evaluated in terms of precision and recall. As a consequence, search engines have difficulties in dealing with queries on structured data, such as grasping the meaning of complex folder hierarchies or finding relationships among documents based on their modification times, authors, or lineage. At the other extreme, an information-integration system allows users to pose complex structural queries over semantically integrated structured or even semi-structured data sources. These systems offer precise query semantics, defined in terms of data models and schemas. However, defining schemas, and mappings among different schemas, is known to be a manual and costly process, which requires the involvement of specialized professionals. As a consequence, information-integration systems are inapplicable to situations in which the number of schemas is too large or users are not specialized, such as in personal information spaces.

In the iMeMex project at ETH Zurich, we have explored a new hybrid data management architecture in-between the two extremes of search engines and traditional information-integration systems. We term systems

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

built following this new architecture *Personal Dataspace Management System (PDSMS)* [2]. The key idea of this new breed of systems is to allow users to smoothly transition from search to information integration. The system should not require any investments in semantic integration before querying services on the data are provided, i.e., at the beginning our system pretty much behaves as a standard search engine. In contrast to search engines, however, our system can be gradually enhanced over time by defining relationships among the data. In that vein, users are able to reach the idiosyncratic level of integration they are most comfortable with for their personal information. The system always strives to offer a unified view of all of the information, but the quality of the integration among items in that view is determined by the amount of integration effort that users are willing to invest over time.

In this paper, we discuss the lessons we have learned while building `iMeMex`, the first incarnation of a PDSMS [1, 2, 4, 5, 6, 13, 22, 23]. We begin by comparing PDSMSs to other common data management architectures in Section 2. In order to illustrate how `iMeMex` mixes search and information integration, we present an example in Section 3. Section 4 proceeds by discussing our choice of data and query model for `iMeMex`. After that, Section 5 discusses advantages and disadvantages of the novel information integration techniques developed to mix search with information integration in `iMeMex`. We discuss system engineering issues in Section 6 and conclude in Section 7.

2 Comparison of PDSMSs to other Systems

In this section, we compare PDSMSs to other data management architectures.

Information-Integration System. Information-integration systems are an all-or-nothing solution: Either the schema mappings are available and the sources can be queried by the system or the sources are simply not reachable [10, 11, 17, 20, 27]. This limitation hinders their applicability to scenarios in which the number of schemas to integrate is large and the users of the system are not specialized enough to provide schema mappings, as may be the case with personal and social information spaces. In contrast, PDSMSs allow users to query all data sources from the start, albeit with lower precision and recall than if the system knew the relationships among data items. These relationships may be then provided to the PDSMS over time, allowing users to reach the level of result quality and integration most cost-effective for their personal dataspace.

DBMS. With respect to managing personal information, DBMSs have two major limitations: they require a schema to be defined and they take full control of the data [26]. The former limitation is similar to the one faced by information-integration systems. DBMSs are also schema-first systems that present an all-or-nothing solution while PDSMSs allow pay-as-you-go integration of the dataspace. The latter limitation implies that all data must be imported into the DBMS so that the system will provide services over it. As personal dataspaces are a highly heterogeneous and distributed mix, a solution that does not require full control, such as a PDSMS, is more tenable.

Schema-based Dataspace System. Franklin, Halevy, and Maier wrote a vision paper calling for a data co-existence approach to information integration [9]. They termed their envisioned system abstraction a Dataspace Support Platform (DSSP). Although the vision for DSSPs is compelling, little has been shown by Franklin et al. [9] on how to actually achieve it. There have been so far two main research approaches to building DSSPs. The first approach, which we call *schema-based dataspace systems*, takes a traditional information-integration architecture and attempts to relax it in order to reduce manual intervention in the integration process. As an example, probabilistic schema mappings [7] and probabilistic mediated schemas [24] propose that both mappings and mediated schemas be automatically created from the schemas of the data sources. In order to get enough quality, however, mappings must be extracted from a common domain [7, 24]. In addition, this approach still fundamentally relies on declaring schemas and schema mappings before any data source can be integrated into the system. In contrast, PDSMSs offer an alternative approach to building DSSPs. As discussed in the Introduction, PDSMSs are bootstrapped as a state-of-the-art search engine, but allow users to increase the level of integration of their dataspace over time, in a pay-as-you-go fashion. In summary, schema-based dataspace systems relax an information-integration architecture to bring it closer to search technology, while PDSMSs power up a search architecture to bring it closer to information-integration technology.

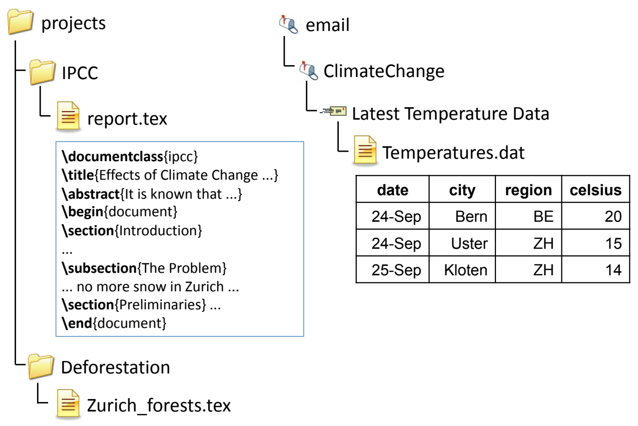
Search Engine. When a PDSMS is bootstrapped, it resembles a state-of-the-art search engine, providing a ranked querying service over a set of heterogeneous sources without any need to declare schemas or mappings. In contrast to a search engine, however, over time, integration hints are added to the PDSMS in order to improve the quality of query results. These hints are *not* full-blown schema mappings, but rather small indications about how the items in the dataspace are related to one another. The system takes advantage of these hints to raise the integration level of the dataspace in a pay-as-you-go fashion.

Cloud Computing Platform. Both PDSMSs and cloud computing platforms are pay-as-you-go solutions, but at different conceptual levels. While a cloud computing platform enables its users to add more hardware and scale incrementally, a PDSMS enables its users to add more relationships and integrate incrementally. There are scenarios in which combining the two characteristics into a common platform is desirable, e.g., for creating an integration system for the whole Web. Further research is necessary to understand how to best bring together these two system abstractions.

Social System. Social systems allow users to establish links to each other and, more importantly, to share information [15]. PDSMSs differ from social systems in that they must enable users to integrate and not only share the information in their dataspace. As such, there must be mechanisms in a PDSMS that allow users to resolve heterogeneity across their data sets and to create arbitrary connections among data items declaratively.

3 Searching Personal Information with `iMeMex`: an Example

As stated earlier, when a PDSMS is bootstrapped, it provides a search service over all data in the sources. Over time, however, users may increase the level of integration of their dataspaces by adding relationships in a pay-as-you-go fashion. In this section, we show an example of how this process works in `iMeMex`.



```
\documentclass{ipcc}
\title{Effects of Climate Change ...}
\abstract{It is known that ...}
\begin{document}
\section{Introduction}
...
\subsection{The Problem}
... no more snow in Zurich ...
\section{Preliminaries} ...
\end{document}
```

date	city	region	celsius
24-Sep	Bern	BE	20
24-Sep	Uster	ZH	15
25-Sep	Kloten	ZH	14

Figure 1: Two data sources in an example personal dataspace.

Suppose a climate-change researcher wishes to search her dataspace for information on the impact of global warming in Zurich. She could start her exploration by posing the keyword query:

```
global warming zurich
```

In the example data sources shown in Figure 1, that simple keyword query returns the files `report.tex` and `Zurich_forests.tex`, appropriately ranked according to term occurrences. There is, however, rich structured data about temperature variations in the region of Zurich hidden in the user's email server. Unfortunately, this data is not returned because the search baseline has no way to translate the keyword query into an appropriate structured query to that data source.

In iMeMex, users may remedy this situation by specifying declarative relationships among portions of their dataspace. These declarative relationships are called *trails*. Users may specify two types of trails: *semantic trails*, which establish relationships among sets of elements, and *association trails*, which establish relationships among individual elements in the dataspace. For example, the user may specify the following semantic trails:

$$T_1 : \texttt{global warming} \longrightarrow \texttt{//projects/IPCC//*}$$
$$T_2 : \texttt{//projects/IPCC} \longleftrightarrow \texttt{//email/ClimateChange}$$
$$T_3 : \texttt{zurich} \longrightarrow \texttt{region = ``ZH''}$$

T_1 directs the search engine to look for (and rank higher) items under the IPCC filesystem folder whenever the user queries for global warming. T_2 establishes that any search in the context of the IPCC filesystem folder should also consider the information inside the ClimateChange folder in the email server. Finally, T_3 translates keyword searches for zurich into a structural query on the region attribute. Taken together, these *integration hints* enable iMeMex to return the appropriate rows inside the Temperatures.dat file that fulfill the user's information need. In order to achieve that, iMeMex employs query rewrite algorithms that exploit the integration semantics given in the trails [22].

The trails in this example resolve several sources of *heterogeneity* in the dataspace. First, T_1 addresses the gap between the vocabulary employed by users on keyword searches and the hierarchies that users create to organize information on their dataspace. Second, T_2 maps between multiple hierarchies in the dataspace that relate to the same underlying information. Third, T_3 transforms between keywords and specific attribute encodings used in structured sources.

In addition to semantic trails, users may also specify association trails in iMeMex. For example, the user may provide the following association trails to the system:

$$T_4 : \texttt{class=file} \overset{\theta_1(l,r)}{\Longleftrightarrow} \texttt{class=file},\ \theta_1(l,r) := (\text{r.date}-1 \le \text{l.date} \le \text{r.date}+1).$$
$$T_5 : \texttt{class=file} \overset{\theta_2(l,r)}{\Longleftrightarrow} \texttt{class=email},\ \theta_2(l,r) := (\text{l.author} = \text{r.from}).$$

T_4 states that any two files touched about the same time are related. Thus, when users obtain as a result the report.tex file, for example, the system will also provide as context other files that she was working on when the report was last modified. T_5 relates all files authored by a given person with emails received from that same person. Given this association trails, iMeMex may provide to the user all emails that are related to the report.tex file when it is returned as a query result. Association trails allow users to declaratively specify the *context* in which query results should be placed. Processing association trails requires specific query rewriting and indexing techniques, which we have implemented in iMeMex [23].

4 Data Model and Query Language

We have introduced a new data model to represent personal information in iMeMex called iDM [5]. In a nutshell, iDM is a graph-based model in which nodes contain both attribute-value pairs and unstructured content. The unstructured content and the connections among nodes may be infinite, a feature useful to represent content and data streams. One important aspect of iDM is that it is lazily computed: all nodes and connections in the graph are merely a logical representation of the underlying data in the dataspace. These nodes and connections may be computed dynamically as the graph is navigated or be precomputed for performance reasons. In order to query iDM graphs, we have designed a simple query language, called iQL (iMeMex Query Language) [5, 23]. The core of iQL is composed of keyword and path expressions, in a manner similar in spirit to NEXI [28] and XPath

Full Text [30]. In contrast to those languages, however, iQL semantics are consistent with a graph data model and its syntax is simplified in order to make it more approachable to end users.

The Motivation. A major motivation for developing iDM was the confusion of data model, data, and data representation found in existing models. For instance, in the XML world, a concrete XML snippet like <a><b>hugo </b></a> is actually just a textual data *representation* of some tree-structured *data* conforming to one of the existing XML *data models*. However, from our experience we had seen that most developers assume data representation and data to be equal, i.e. the string "<a><b>hugo</b></a>" is *"XML data"* — which we consider to be incorrect. This confusion has had impact on several unfortunate system design decisions like storing XML data representations, sending XML data representations over the wire, etc. This, however, does not make sense in our view as the data representation is independent of the actual data. This is also one of the reasons that all XML data models are in fact very similar to object models (with some additional syntactic sugar added here and there). The major differentiator in XML is the SGML-like *data representation* of objects.

The Upside. iDM has a crystal clear separation of data and data representation. The flexibility of iDM has been key in allowing us to incorporate all of the heterogeneous data items present in the dataspace into a common model. iDM graphs can uniformly represent data source items such as email messages, files, folders, documents on the Web, social graphs, and database tuples. This also allows us to represent hierarchies available in different formats inside the same logical hierarchy, e.g., the tree-structured content inside a file is just a twig in the larger tree-structured file system. In addition, extracted data is represented in the same model. We have experimented with several different content converters for extraction. Some of them were more general, such as for LaTeX, XML, image metadata, RSS/ATOM streams, or ZIP files, while others were more specific, such as search engine or REST service parsed results.

The Downside. The flexibility that iDM bought us came at the cost of performance and code simplicity. The system had to constantly assume that a general graph was being processed, meaning that clever optimizations developed for tree-structured data were not available to us. In addition, our data model assumed flexible schemas for different nodes in the graph and thus we could not exploit schema information as aggressively as relational implementations. We have tried to reduce the performance hit by having our data source access code export some information about structural properties of the graph being processed. For example, it is useful to detect bridges in the graph as they may be used to reduce memory allocation in graph traversal. While such optimizations brought competitive performance to `iMeMex`, `iMeMex`'s performance remained inferior to what could be obtained by an optimized implementation over simpler data models.

Some dataspace approaches have chosen data models similar to RDF to represent the data in the dataspace [12]. These representations are conceptually simpler than the one introduced by iDM and there has been recent work targeted at providing efficient indexing support [18]. Nevertheless, there are still challenges for a truly efficient implementation when triples represent arbitrary graph patterns and may contain information as disparate as binary file contents and simple integers. In addition, there is little leveraging of schema information, when it is available. We speculate that a more interesting approach for future systems may be to abandon the idea of having a single common data model altogether. This enables the system to use specialized implementations for each specific data model in the dataspace. At the same time, the system must deal with the challenge of providing integration of data in these disparate models without relying on a single common data model but rather using the most appropriate "translation model" at hand.

5 Pay-as-you-go Information Integration

We have developed novel pay-as-you-go information integration techniques that represent the bulk of `iMeMex`'s query planning [22, 23]. As shown in Section 3, the search baseline provided by `iMeMex` may be enriched over time by adding integration hints, called *trails*. We have explored two major classes of trails in `iMeMex`: semantic trails [22] and association trails [23]. Semantic trails model relationships among *sets* of items in a dataspace.

A semantic trail can express, for example, that when a user queries for items in the path `//projects/PIM`, then items in the path `//mike/research/PIM` should also be considered. Association trails, on the other hand, model fine-grained relationships among individual *items* in a dataspace. For example, an association trail can express that any document is related to all emails authored by the same person who wrote the document. In other words, semantic trails enrich query results by obtaining sets of semantically equivalent items, while association trails enhance the dataspace by defining connections among individual items.

The Motivation. The major motivation for developing trails was to have a technique that is far easier to use than full-blown information integration, but provides much better query results than standard search engines.

The Upside. In contrast to a full-blown schema mapping, a trail is a simple integration hint that relates two query definitions. There is no need to specify explicit schemas for the sources in the dataspace before a trail can be defined. The queries related by a trail can be any keyword or path query expressed in iQL. As a consequence, trails are at the same time simple enough to enable users to specify them and general enough to capture interesting integration semantics. For example, semantic trails are able to model keyword-to-keyword equivalences as found in thesauri, keyword-to-query bookmarks, or attribute-to-attribute matches as special cases [22]. Association trails define declaratively a graph of connections among instances in the dataspace, being able to relate elements in timelines, through similar content or versions, or by any metadata attached to them [23]. Trails can be shared among different users and cover specialized domains. In addition, we have built in support to attach probability values to trails and to exploit these values to control query rewriting when thousands of uncertain trails are defined in the system.

The Downside. It is clear that candidate trails can be obtained from thesauri [29] and schema matching techniques [21]. It is also clear that relevance feedback can be leveraged in order to ask for confirmations of only the most promising relationships [14]. In spite of that, mining high-quality trails automatically from a user's dataspace remains an elusive goal. We have performed some initial work in that direction for keyword-to-keyword trails [25]. However, a deeper and more interesting question is how to link unstructured, keyword queries to structured, path queries based on the contents of a dataspace and on its usage patterns. This link would allow users to obtain high-quality structured data while only having to interact with a simple keyword query interface.

Another interesting issue we have been faced with is related to the expressiveness of the query language used in the system. Currently, iQL is used both for user queries and for trail definitions. For user queries, a language based on keywords and paths is a natural choice in a personal dataspace scenario. It unifies functionality found in search engines as well as in operating system navigators. However, scripting capabilities such as the ones found in operating system shells could be an interesting addition. For trail definitions, more expressiveness would allow us to define more complex relationships. A negative consequence, however, might be that the query rewriting process will become more complex. As learned from classic information integration, there is a delicate balance to be achieved when deciding on expressiveness, given that fully general queries and relationships may make the query rewriting problem undecidable [16].

6 System Engineering Issues

In order to maintain a system as large as `iMeMex`, the system needs to be separated into components. When we started to implement `iMeMex`, we had a component model in mind and used Java interfaces to connect the different components. Later on we decided to split the code apart into OSGi[1] bundles in order to gain even more flexibility to add and remove components — even at run time. Splitting a large piece of code into OSGi bundles is no easy task and consequently we also only managed to do it partially: The core functionality remained in a central main bundle, the code which was already structured using interfaces was put into smaller bundles. Over

[1]OSGi implementations (e.g. Equinox [8] or Oscar [19]) are platforms for service oriented architectures (SOA). Each service is packaged into a so called OSGi bundle.

time, we chopped away more and more of the core bundle, but still, a part remains.

Motivation. A system such as `iMeMex` offers many opportunities for extensions. More plugins can be added to access different data sources, more data formats may be supported, different indexing mechanisms are useful and query planning and processing can be indefinitely enhanced with additional features. Employing a modular system architecture allows us to add, replace and remove extensions at any point in time.

The Upside. We believe that modularity is key to develop a large system. Working with modules forces developers to follow certain guidelines in terms of what other parts of the system they are allowed to use. Doing so clarifies the system architecture over time and it also allows us to try out new things and still be able to simply get back to an old configuration when we do not observe the intended effect. OSGi naturally enforces such modularity. Modules can also be used as libraries which are shared and distributed among different projects.

The Downside. While OSGi brings modularity, this comes with a cost. Several students complained that adding or changing bundles was too complicated. The reason was that they not only had to write their code, but also had to write a script to package their bundle and configure the system to make use of it. In addition, the strict partitioning into bundles disabled "quick-hacks", i.e., adding of new functionality violating bundle boundaries or interfaces. In order to add new functionality, bundle dependencies had to be respected. In the end we had about 70 OSGI bundles. This also shows that we tried too include a lot of functionality into the system. However, this had the cost that some of this functionality was not maintained over time and got 'lost' due to disabled tests.

Another downside is that too many levels of abstraction introduce very high function call stacks and it becomes more and more complicated not only to understand what happens, but also to determine where performance goes. One such example that we experienced was the lazy computation of iDM graphs. Accessing a simple tuple attribute, could have led to fetching an RSS feed, parsing the XML document and converting it to a large graph of resource nodes — all hidden behind an innocuous `getLastModified()` call.

In the end we took a path in-between modularity and flexibility: The code is still developed in terms of OSGi bundles, but now we also offer a monolithic mode of execution which can be used for testing, debugging and integrating `iMeMex` as a library into other projects.

7 Conclusions

This paper has reviewed several key design choices we took while building `iMeMex`, the first incarnation of a PDSMS. The approach taken in `iMeMex` is to mix search with information integration into a single hybrid platform. We have discussed consequences of our data model and query language, techniques for pay-as-you-go information integration and query processing, as well as system engineering decisions. We believe this experience report may help future designers build the next generation of PDSMSs.

Acknowledgments We would like to thank ETH Zurich, SNF, and in particular Donald Kossmann for making this project possible. We would like to thank all students that contributed to `iMeMex` over the past four years: Tobias Abt, Aarno Aukia, Sandro Blum, Urs Blum, Sibylle Dürr, Markus Färber, Steven Fluck, Pascal Gamper, Olivier Girard, Stefan Hildenbrand, Julia Imhof, Roger Jäggi, Shant Kirakos Karakashian, André Schmidt, Stefan Stalder, Marco Steybe, Philip Stutz, Christian Tarnutzer, and Georg Troxler.

References

[1] L. Blunschi. Cost-based Query Optimization in iMeMex. Master's thesis, ETH Zurich, 2007.

[2] L. Blunschi, J.-P. Dittrich, O. R. Girard, S. K. Karakashian, and M. A. V. Salles. A Dataspace Odyssey: The iMeMex Personal Dataspace Management System. In *CIDR*, 2007. Demo Paper.

[3] R. Boardman. *Improving Tool Support for Personal Information Management*. PhD thesis, Imperial College London, 2004.

[4] J.-P. Dittrich. iMeMex: A Platform for Personal Dataspace Management. In *SIGIR PIM Workshop*, 2006.

[5] J.-P. Dittrich and M. A. V. Salles. iDM: A Unified and Versatile Data Model for Personal Dataspace Management. In *VLDB*, 2006.

[6] J.-P. Dittrich, M. A. V. Salles, D. Kossmann, and L. Blunschi. iMeMex: Escapes from the Personal Information Jungle. In *VLDB*, 2005. Demo Paper.

[7] X. Dong, A. Halevy, and C. Yu. Data Integration with Uncertainty. In *VLDB*, 2007.

[8] Equinox: Eclipse OSGI implementation. `http://www.eclipse.org/equinox/`.

[9] M. Franklin, A. Halevy, and D. Maier. From Databases to Dataspaces: A New Abstraction for Information Management. *SIGMOD Record*, 34(4):27–33, 2005.

[10] M. Friedman, A. Levy, and T. Millstein. Navigational Plans For Data Integration. In *AAAI*, 1999.

[11] L. Haas, D. Kossmann, E. Wimmers, and J. Yang. Optimizing Queries across Diverse Data Sources. In *VLDB*, 1997.

[12] B. Howe, D. Maier, and L. Bright. Smoothing the ROI Curve for Scientific Data Management Applications. In *CIDR*, 2007.

[13] iMeMex project web-site. `http://www.imemex.org`.

[14] S. Jeffery, M. Franklin, and A. Halevy. Pay-as-you-go User Feedback for Dataspace Systems. In *SIGMOD*, 2008.

[15] G. Koutrika, B. Bercovitz, R. Ikeda, F. Kaliszan, H. Liou, Z. Zadeh, and H. Garcia-Molina. Social Systems: Can We Do More Than Just Poke Friends? In *CIDR*, 2009.

[16] M. Lenzerini. Data Integration: A Theoretical Perspective. In *PODS*, 2002.

[17] A. Levy, A. Rajaraman, and J. Ordille. Querying Heterogeneous Information Sources Using Source Descriptions. In *VLDB*, 1996.

[18] T. Neumann and G. Weikum. RDF-3X: a RISC-style Engine for RDF. *PVLDB*, 1, 2008.

[19] Oscar: OSGi implementation. `http://oscar.objectweb.org/`.

[20] Y. Papakonstantinou, H. Garcia-Molina, and J. Widom. Object Exchange Across Heterogeneous Information Sources. In *ICDE*, 1995.

[21] E. Rahm and P. Bernstein. A Survey of Approaches to Automatic Schema Matching. *The VLDB Journal*, 10(4), 2001.

[22] M. A. V. Salles, J.-P. Dittrich, S. K. Karakashian, O. R. Girard, and L. Blunschi. iTrails: Pay-as-you-go Information Integration in Dataspaces. In *VLDB*, 2007.

[23] M. V. Salles. *Pay-as-you-go Information Integration in Personal and Social Dataspaces*. PhD thesis, ETH Zurich, 2008.

[24] A. D. Sarma, X. Dong, and A. Halevy. Bootstrapping Pay-As-You-Go Data Integration Systems. In *SIGMOD*, 2008.

[25] A. Schmidt. Semi-Automatic Trail Creation in iMeMex. Master's thesis, ETH Zurich, 2008.

[26] K. A. Shoens, A. Luniewski, P. M. Schwarz, J. W. Stamos, and J. T. II. The Rufus System: Information Organization for Semi-Structured Data. In *VLDB*, 1993.

[27] I. Tatarinov and A. Halevy. Efficient Query Reformulation in Peer Data Management Systems. In *SIGMOD*, 2004.

[28] A. Trotman and B. Sigurbjörnsson. Narrowed Extended XPath I (NEXI). In *INEX Workshop*, 2004.

[29] WordNet. `http://wordnet.princeton.edu/`.

[30] XQuery and XPath Full Text 1.0, 2008. `http://www.w3.org/TR/xpath-full-text-10/`.