DISS. ETH NO. 18079

Pay-as-you-go Information Integration in Personal and Social Dataspaces

A dissertation submitted to
ETH ZURICH

for the degree of
Doctor of Sciences

presented by

MARCOS ANTONIO VAZ SALLES

Master in Informatics, Catholic University of Rio de Janeiro (PUC-Rio)
born 25th of April, 1977
citizen of Brazil

accepted on the recommendation of

Prof. Donald Kossmann, examiner
Prof. Jens Dittrich, co-examiner
Prof. Thomas Gross, co-examiner
Prof. David Maier, co-examiner

2008

Acknowledgements

Jens Dittrich has been my mentor and friend for the past three and a half years. His insightful comments, instigating thoughts, and relentless dedication have deeply influenced both this dissertation and my professional posture. I have learned from Jens to always strive for more, to settle for nothing less than a scientifically sound work, and to never set the bar on what people are able to produce, but to help them excel in their own way.

My profound admiration also extends to Donald Kossmann. Donald has given me and Jens complete freedom to work together and to pursue our ideas. Yet, Donald has never shied away from actively providing me with feedback at key moments during the development of this dissertation, always helping me see the forest through the trees.

I owe heartfelt thanks to both Donald and Jens for believing and investing in me and for giving me the opportunity to work in what is simply one of the best systems research groups in the world.

I am grateful for the work and availability of the other members of my committee. Prof. David Maier has provided me with several interesting comments and has reviewed my dissertation in detail. Prof. Thomas Gross has raised pertinent questions that helped me improved my work. I thank Prof. Joachim Buhmann for being available to chair my examination committee and for his questions and comments.

This work would not have been possible without the tireless efforts of the many people who worked on the `iMeMex` project. Lukas Blunschi, Olivier Girard, and Shant Karakashian deserve special recognition.

Several people have helped me navigate all administrative aspects of life in Switzerland and at ETH, in particular Denise Spicher, Heidy Schümperlin, and Simonetta Zsysset.

The Swiss National Science Foundation (SNF) has partially supported the research that led to this dissertation under contract 200021-112115. The remaining financial support for this research has come from ETH Zurich.

To my friends and colleagues in Zurich, Cristian, Tim and Karin, Shant, Sara and Jörg, Rokas, Lukas, Ghislain, Tea, Heidy and Christoph, Peter and Anita, Alex, Simonetta, Irina and Virgiliu, thanks for showing me around the place and filling my mind and my heart with good memories.

My special appreciation to my parents, for all the support and love they have given me throughout my life and especially during these past three and a half years.

To God, for all the light in every step of my way.

Contents

1 Introduction **1**

1.1 Motivation 1

1.2 Overview of State-of-the-Art Information-Integration Architectures 2

1.2.1 Search Engines 2

1.2.2 Traditional Information-Integration Systems 4

1.3 Dataspace Management Systems 5

1.4 Contributions of this Dissertation 6

1.4.1 iMeMex Data Model (iDM) 6

1.4.2 iTrails 7

1.4.3 Association Trails 8

1.4.4 iMeMex PDSMS Architecture 8

1.4.5 Publications 9

1.5 Structure of this Dissertation 10

2 iDM: A Unified and Versatile Data Model for Personal Dataspace Management **11**

2.1 Introduction 11

2.1.1 Motivation 11

2.1.2 The Problem 12

2.1.3 Contributions 13

2.2 **iMeMex** Data Model 14

2.2.1 Overview 14

2.2.2 Resource Views 15

2.2.3 Examples 16

2.3 Instantiating Specialized Data Models . . . 19
2.3.1 Resource-View Classes . . . 19
2.3.2 Files and Folders . . . 20
2.3.3 XML . . . 21
2.3.4 Data Streams . . . 22
2.4 Computing the iDM Graph . . . 23
2.4.1 Lazy Resource Views . . . 23
2.4.2 Extensional Components . . . 24
2.4.3 Intensional Components . . . 24
2.4.4 Infinite Components . . . 25
2.5 Searching with the **iMeMex** Query Language . . . 27
2.6 Evaluation . . . 29
2.6.1 Setup . . . 29
2.6.2 Results . . . 30
2.7 Related Work . . . 33
2.8 Conclusions . . . 35

3 iTrails: Pay-as-you-go Information Integration in Dataspaces 37
3.1 Introduction . . . 37
3.1.1 Motivating Example . . . 38
3.1.2 Contributions . . . 41
3.2 Data and Query Model . . . 41
3.2.1 Data Model . . . 41
3.2.2 Query Model . . . 42
3.2.3 Query Algebra . . . 43
3.3 iTrails . . . 43
3.3.1 Basic Form of a Trail . . . 44
3.3.2 Trail Use Cases . . . 45
3.3.3 Where Do Trails Come From? . . . 46
3.3.4 Probabilistic Trails . . . 47
3.3.5 Scored Trails . . . 47

3.4 iTrails Query Processing 48
3.4.1 Overview 48
3.4.2 Matching 48
3.4.3 Transformation 49
3.4.4 Merging 51
3.4.5 Multiple Trails 51
3.4.6 Trail Rewrite Analysis 53
3.4.7 Trail Indexing Techniques 55
3.5 Pruning Trail Rewrites 55
3.5.1 Trail Ranking 55
3.5.2 Pruning Strategies 56
3.5.3 Rewrite Quality 57
3.6 Experiments 58
3.6.1 Data and Workload 58
3.6.2 Quality and Completeness of Results 61
3.6.3 Query Performance 61
3.6.4 Rewrite Scalability 62
3.6.5 Rewrite Quality 65
3.7 Related Work 66
3.8 Conclusions 67

4 Association Trails: Modeling and Indexing Intensional Associations in Dataspaces 69
4.1 Introduction 69
4.1.1 Motivating Example 71
4.1.2 Contributions 73
4.2 Data and Query Model 74
4.2.1 Data Model 74
4.2.2 Query Model 75
4.2.3 Basic Index Structures 76
4.3 Association Trails 76
4.3.1 Basic Form of an Association Trail 77

4.3.2 Association Trail Use Cases 77
4.3.3 Differences Between Association Trails and Semantic Trails 79
4.3.4 Where do Association Trails Come From? 80
4.3.5 Probabilistic Association Trails 81
4.4 Association-Trails Query Processing 81
4.4.1 Neighborhood Queries 82
4.4.2 Canonical Plan 83
4.4.3 N-Semi-Joins 85
4.5 Association-Trails Indexing 87
4.5.1 Materialize $A_*^{L.Q}$ and $A_*^{R.Q}$ 88
4.5.2 Materialize $A_1^{L.Q} \bowtie_{\theta_1} A_1^{R.Q} \cup \ldots \cup A_n^{L.Q} \bowtie_{\theta_n} A_n^{R.Q}$ 89
4.5.3 Grouping-compressed Index 91
4.5.4 Query Materialization and Hybrid Approach 95
4.5.5 Extensions 97
4.6 Ranking Association Trail Results 98
4.7 Experiments 100
4.7.1 Setup and Datasets 101
4.7.2 Association-Trails Query Processing 102
4.7.3 Association-Trails Indexing 106
4.7.4 Hybrid Approach 111
4.8 Related Work 116
4.9 Conclusions 118

5 A Dataspace Odyssey: The iMeMex Personal Dataspace Management System 121

5.1 Introduction 121
5.1.1 Contributions 122
5.2 The iMeMex Vision 123
5.3 **iMeMex** Core Architecture 124
5.3.1 Logical Layers 124
5.3.2 PHIL Services 125
5.3.3 LIL Services 126

5.4 Implementation of Logical Layers . . . 127
5.4.1 Query Processor . . . 127
5.4.2 Resource View Manager . . . 128
5.5 System Features . . . 129
5.5.1 Current Features . . . 129
5.5.2 Upcoming Features . . . 130
5.6 Use Case . . . 131
5.6.1 iMeMex Dataspace Navigator . . . 131
5.6.2 Querying and Navigating the Dataspace . . . 131
5.7 Related Work . . . 133
5.8 Conclusions . . . 135

6 Conclusion 137

6.1 Summary . . . 137
6.2 Ongoing and Future Work . . . 139

Abstract

A personal and social dataspace is the set of all information pertaining to a given user. It includes a heterogeneous mix of files, folders, email, contacts, music, calendar items, images, among others, distributed among a set of data sources such as filesystems, email servers, network shares, databases, and web servers. In addition, it includes all connections this user has to other users in a number of online services such as social networking web sites. In spite of a personal and social dataspace being richly heterogeneous and distributed, only very limited tools are available to aid users manage their information and have a unified view over it. At one extreme, search engines allow users to pose simple keyword and path searches over all of their data sources. These systems, however, return only ranked lists of best-effort results and provide limited or no means for users to increase the quality of query results returned by the system over time. At the other extreme, systems built on top of classic database technology, such as traditional information-integration systems, provide precise query semantics for queries over a set of data sources. Although the quality of query results returned by these systems is high, they are typically restricted to a subset of the personal information of a user, given the need to specify complex schema mappings to integrate the data. As a consequence, these systems have limited coverage and provide equally limited support for non-expert users to refine their view of their personal information over time.

This thesis investigates a new breed of information-integration architecture that stands in-between the two extremes of search engines and traditional information-integration systems. We term this new type of system a *Personal Dataspace Management System (PDSMS)*. Like a search engine, when a PDSMS is bootstrapped, it provides a simple search service over all of the user's dataspace. In contrast to search engines, however, the PDSMS represents data not at the coarse-grained level of files (or text documents), but rather using a fine-grained graph-based data model. In addition, a PDSMS provides means for a user to increase the level of integration of her dataspace gradually, in a pay-as-you-go fashion. That is done by enabling users to provide simple integration "hints" that allow the PDSMS to improve the quality of query results. In contrast to traditional information-integration systems, however, at no point does the PDSMS require users to specify a global mediated schema for their information.

We make four main contributions to the design of PDSMSs. First, we propose the `iMeMex` Data Model (iDM), a simple, yet powerful, graph-based data model able to represent the heterogeneous data mix found in a personal and social dataspace. Our data model enables query capabilities on top of the user's dataspace not commonly found in state-of-the-art tools.

Second, we introduce iTrails, a technique for pay-as-you-go information integration in dataspaces. This technique enables users to increase the level of integration of their personal dataspaces by providing integration “hints”, called trails, to the system. Trails specify relationships between arbitrary subsets of items in a user dataspace.

Third, we propose *association trails*, a technique to declaratively model fine-grained relationships among individual instances in a dataspace. With association trails, instances in a dataspace are connected in a graph intensionally. This graph may be used to explore the dataspace and find related instances according to different contexts (e.g. time, similar content, or same metadata).

Fourth, we integrate all of the previous contributions into the architecture of `iMeMex`, the first PDSMS implementation. The `iMeMex` PDSMS follows a layered and extensible architecture. The various layers of `iMeMex` provide increasingly higher levels of abstraction over a personal and social dataspace, bringing physical and logical data independence to this heterogeneous environment.

Zusammenfassung

Ein persönlicher und sozialer Datenraum umfasst die Menge aller Informationen, die einem bestimmten Benutzer zugeordnet sind. Hierzu gehören eine heterogene Mischung aus Dateien, Ordnern, E-Mails, Kontakten, Musik, Kalender-Einträgen, Bildern und anderem. Diese sind verteilt auf verschiedene Datenquellen wie Dateisysteme, E-Mail-Server, Netzwerk-Ablagen, Datenbanken und Webserver. Darüber hinaus umfasst der Datenraum alle Verbindungen, welche dieser Benutzer zu anderen Benutzern in Online-Portalen für Soziale Netzwerke hat. Obwohl ein persönlicher und sozialer Datenraum also sehr heterogen und auf verschiedene Datenquellen verteilt ist, gibt es doch nur eine sehr geringe Anzahl von Werkzeugen, welche einem Benutzer helfen, seine Informationen zu verwalten und eine einheitliche Sicht darüber zu erhalten. Auf der einen Seite ermöglichen Suchmaschinen einfache Schlüsselwortabfragen beziehungsweise Pfadabfragen zu machen. Diese Systeme liefern jedoch nur eine geordnete Liste von Ergebnissen und bieten dem Benutzer keine oder nur wenig Möglichkeiten, die Qualität der Suchergebnisse nach und nach zu verbessern. Auf der anderen Seite ist es möglich, mit Systemen, die auf klassischer Datenbanktechnologie aufbauen, wie zum Beispiel traditionellen Information-Integrationssystemen, eine präzise Abfrage-Semantik über verschiedenen Datenquellen zu ermöglichen. Die Qualität der Suchergebnisse dieser Systeme ist zwar hoch, aber üblicherweise ist die Suche auf eine Teilmenge aller Informationen beschränkt, da für jede Datenquelle komplexe Schema-Abbildungsregeln spezifiziert werden müssen. Als Folge davon bieten diese Systeme nur eine begrenzte Abdeckung der Daten und bieten auch nur eingeschränkte Unterstützung für Nicht-Experten, ihre Sicht über ihre persönlichen Informationen im Laufe der Zeit zu verfeinern.

Diese Arbeit geht einen neuen Weg. Wir untersuchen eine neue Art von Information-Integrationsarchitektur, welche zwischen den beiden Extremen von Suchmaschinen und traditionellen Information-Integrationssystemen steht. Wir nennen diese neue Art von System *Persönliches Datenraum-Management-System (PDSMS)*. Vergleichbar mit einer Suchmaschine bietet ein PDSMS anfänglich einen einfachen Suchdienst über alle Informationen eines Benutzers an. Im Gegensatz zu Suchmaschinen repräsentiert ein PDSMS die Daten jedoch nicht auf einer grobkörnigen Ebene wie Dateien (oder Textdokumente), sondern mit Hilfe eines feinkörnigen, grafischen Datenmodells. Zusätzlich bietet ein PDSMS die Möglichkeit, das Niveau der Datenintegration kontinuierlich zu erhöhen. Dies wird erreicht, indem ein Benutzer dem System einfache Integrations-"Tipps" gibt, welche dem System helfen, die Qualität der Suchergebnisse zu verbessern. Ungleich einem traditionellen Information-Integrationssystem, verlangt ein PDSMS kein kompletes Regelset, welches alle Datenquellen miteinander verbindet.

In dieser Arbeit machen wir vier wichtige Beiträge zur Gestaltung von PDSMS. Erstens präsentieren wir das iMeMex Daten-Modell (iDM), ein einfaches aber trotzdem mächtiges, grafisches Datenmodell, welches den heterogenen Datenmix in einem persönlichen und sozialen Datenraum darstellen kann. Unser Datenmodell ermöglicht Abfragen über den Datenraum eines Benutzers, welche in anderen modernen Werzeugen nicht möglich sind.

Zweitens stellen wir iTrails, eine Technik für Pay-as-you-go-Informationsintegration in Datenräumen vor. Diese Technik ermöglicht Benutzern, den Grad der Integration von Daten Schritt für Schritt durch Integrations-"Tipps" — so genannten Trails — zu erhöhen. Trails geben Hinweise auf Beziehungen zwischen beliebigen Teilmengen von Elementen im Datenraum eines Benutzers an.

Drittens schlagen wir *Assoziationstrails* vor, eine Technik, um deklarativ feinkörnige Beziehungen zwischen einzelnen Elementen in einem Datenraum zu modellieren. Mit Assoziationstrails sind Instanzen in einem Datenraum über die Auswertung eines Prädikates verbunden. Hierdurch entsteht ein assoziativer Overlay-Graph. Dieser Graph wird ausgenutzt, um einen Datenraum nach ähnlichen Kontexten zu durchforschen, zum Beispiel Zeit, ähnlichem Inhalt, gleichen Metadaten oder ähnlichen Interessen von Benutzern.

Viertens verbinden wir alle oben genannten Beiträge zu einem konkreten System: iMeMex. Dieses System ist eine der ersten Implementierungen eines PDSMS. Zudem ist iMeMex erweiterbar und besteht aus verschiedenen Systemebenen. Jede weitere Ebene in iMeMex führt zu höherer Abstraktion in einem persönlichen und sozialen Datenraum und ermöglicht damit physische und logische Daten-Unabhängigkeit, sowie Pay-as-you-go-Informationsintegration.

Chapter 1

Introduction

1.1 Motivation

Personal Information Management (PIM) is the process performed daily by users to collect, store, organize, and access their collections of digital objects [20]. 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 filesystems, network shares, external drives, email servers, and on the Web.

As interconnected devices become increasingly tied to the lifestyles of millions of people, more and more of our activities are going digital. Even social activities are now performed on-line using community web sites, which list millions of users organized into complex social graphs [7]. This trend imposes increasing strain on traditional tools used to manage personal and social information. In fact, most current tools and applications focus on specific types of information, such as email, social contacts, or particular file types. As a result, users are offered a fragmented view of their personal and social information spaces, which they must manually reconcile through significant effort.

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 precise, meaning that the quality of query answers is evaluated in terms of precision and recall. As a consequence, search engines have difficulties dealing with queries on structured data, such as grasping the meaning of complex folder hierarchies in personal information or understanding relationships among people in social spaces. At the other extreme, an information-integration system allows users to pose complex structural queries over semi-structured or structured 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 and users are not specialized, such as in personal and social information spaces.

This thesis is targeted at building next-generation information-integration architectures that provide a new point in the design space in-between the two extremes of search engines and traditional information-integration systems. The key idea of this new breed of systems is to offer a data-coexistence approach. The system should not require any investments in semantic integration before querying services on the data are provided. Rather, the 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 and social information. The system always offers 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.

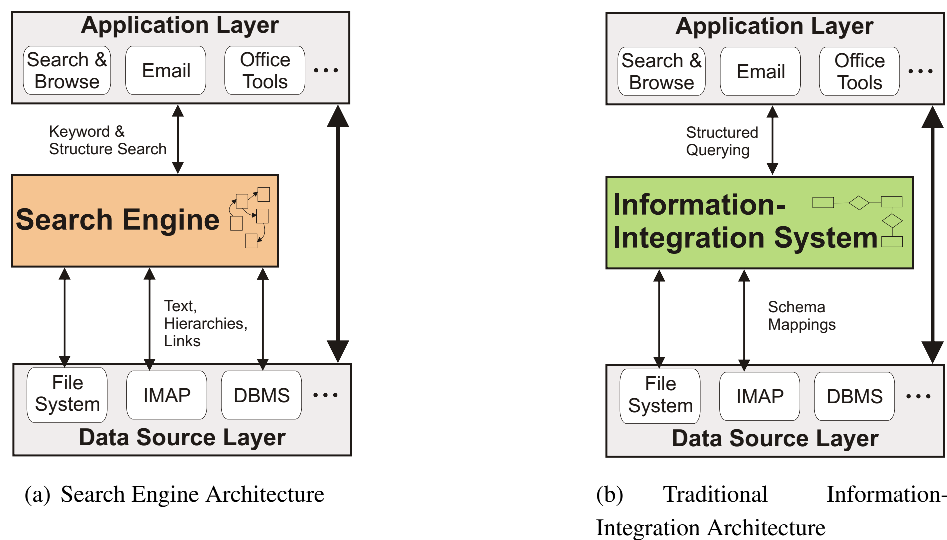
Franklin, Halevy, and Maier [65] wrote a vision paper calling for this new type of information-integration architectures and terming them dataspace systems. Although the vision for dataspace systems is compelling, little has been shown by Franklin et al. [65] on how to actually achieve it. In the following, we will first take a closer look at current information-integration architectures in Section 1.2. We will then review the dataspace vision of [65] in Section 1.3 and proceed in Section 1.4 by stating the contributions that this thesis brings towards the design of dataspace systems.

1.2 Overview of State-of-the-Art Information-Integration Architectures

In this section, we review some related work around the two extreme solutions of search engines and traditional information-integration systems to querying a set of distributed and heterogeneous data sources. This section does not intend to be an exhaustive survey, but rather to set the context in which the contributions of this thesis are developed. Related work specific to each technique introduced by this thesis is discussed in the appropriate subsequent chapters.

1.2.1 Search Engines

Figure 1.1(a) presents a high-level architecture followed by typical search engines, such as Google [25]. Search engines follow a *no-schema* approach, that is, all information from the sources is converted to a simple data model, either the bag-of-words model or a semi-structured XML model. That is done without the need for the search engine to take full control of the data, that is, direct access to the underlying data sources is always possible. The engine then offers a simple keyword search service that allows users to find candidate matches for the queries from the items in the sources. As search engines do not perform any semantic integration on the data, the query model offered by them gives query answers that are not precise. Therefore, a key aspect of the design of a search engine lies in developing an effective ranking method. In the case of web search engines, high-quality ranking methods such as PageRank [25] or



(a) Search Engine Architecture

(b) Traditional Information-Integration Architecture

Figure 1.1: State-of-the-art Information-Integration Architectures

HITS [105] are known. Finding good ranking methods for complex scenarios such as personal and social spaces, however, is still the subject of investigation [58].

XML-IR search engines [9, 29, 37, 156] take the traditional keyword search querying model one step further and provide queries combining structure and content over collections of XML documents. As with classic web search engines, one important concern in these systems is ranking. XSearch [37] ranks results based on text metrics and also whether they are interconnected through common ancestors in the XML document. TopX [156], in contrast, adapts the Okapi BM25 [140] text-scoring metric to XML. The authors of XML Fragments [29] evaluate a number of different ranking alternatives based on structural proximity. What is missing in all of these approaches is the capability to take into consideration integration semantics and to exploit those integration semantics to improve the quality of query results. The same holds for systems that use relevance feedback to improve search results, as for example the work of Schenkel and Theobald [145]. The focus of these approaches is to improve search quality in terms of a single data set rather than using the feedback to provide integration semantics to integrate multiple data sets.

As we will see in this thesis, it is possible to provide integration semantics on top of a search engine in a way that is orthogonal to the underlying scoring scheme used (Chapter 3). We will also see that using a richer data model (Chapter 2) and considering fine-grained integration relationships among data items (Chapter 4) in personal and social dataspaces leads to querying capabilities not available in current search engines. In short, we argue that while current search engines have the strong advantage of providing a query service over the data right from the start, they lack in understanding of the integration semantics of the data items in the sources. Furthermore, there is little support for improving that understanding over time, in a pay-as-you-go fashion. Thus, search engines have limited ability to provide high-quality query results in scenarios such as personal and social dataspaces.

1.2.2 Traditional Information-Integration Systems

Information-integration systems (IIS) aim to provide a global, semantically integrated view of data. The general architecture of a traditional information-integration system is shown in Figure 1.1(b). These systems work with a mediated schema over which structured queries with precise semantics may be posed. The price to be paid for that query service is the need to provide schema mappings between the schemas of the data sources and the mediated schema of the information-integration system. When a schema mapping is not provided (as for the DBMS source in Figure 1.1(b)), then the data source simply cannot be reached by the system. While the query model of IISs is significantly different from the one of search engines, both share some architectural similarities. First, both systems offer a common query service over the data sources, though differing in source coverage and answer quality. Second, both systems may be bypassed and direct access to the data sources is possible at all times.

Classical systems either express the mediated schema as a view on the sources (GAV, e.g., TSIMMIS [131]), source schemas as views on the mediated schema (LAV, e.g., Information Manifold [109]), or a mix of these approaches (GLAV [69]). The differences between LAV, GAV, and GLAV are further discussed by Lenzerini [108]. An extension to traditional IIS is quality-driven information integration [124]. It enriches query processing in a mediator system by associating quality criteria to data sources. One deficiency of these data integration solutions is the need for high up-front effort to semantically integrate all source schemas and provide a global mediated schema. Only after this startup cost can queries be answered using reformulation algorithms [85]. Peer-to-peer data management [94, 127, 154] tries to alleviate the problem of providing schema mappings by not requiring semantic integration to be performed against all peer schemas. Nevertheless, the data on a new peer is only available after schema integration is performed with respect to at least one of the other peers in the network.

Query optimization is also a concern for integration systems. In Multibase [43], several tactics are developed to push down projections and minimize data movement when queries are distributed to the sources. The Garlic optimizer [84] extends traditional query optimization to include rules that describe the query capabilities of the heterogeneous sources accessed by the information-integration system.

Providing schema mappings, however, is the main bottleneck faced by information-integration systems, given the notorious difficulty of this task. Active research is targeted at providing tools to aid humans in the tasks of schema matching and, ultimately, schema mapping [172]. We point the reader to the survey by Rahm and Bernstein [136] for a review of semi-automatic schema-matching methods and also to some more recent papers [21, 47, 101]. Discussions of other aspects related to information integration may be found elsewhere [22, 86, 89].

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. 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 this thesis, we will argue for an approach that allows all of the data in the sources to be queryable right from the start, while at the same time enabling users to specify relationships

that provide increasingly more accurate results. We will show how to represent the data in the sources to make it available for querying with a unified data model (Chapter 2) and describe different categories of relationships that may be added over time to improve the quality of query results (Chapters 3 and 4).

1.3 Dataspace Management Systems

In this section, we review the dataspace vision of Franklin, Halevy, and Maier [65] and provide an overview of the novel information-integration architecture that we will move towards in this thesis. Franklin, Halevy, and Maier advocate in [65] that a new system abstraction is necessary to coordinate research efforts in the data management field. They identify the core challenge for information management today as the one "from organizations (e.g., enterprises, government agencies, libraries, 'smart' homes) relying on a large number of diverse, interrelated data sources, but having no way to manage their *dataspaces* in a convenient, integrated, or principled fashion" [65]. Dataspaces are defined as a set of sources and relationships among them, and the systems used to manage dataspaces are termed Dataspace Support Platforms (DSSPs). In contrast to traditional information-integration systems, DSSPs are a *data co-existence approach*: They require no up-front effort to semantically integrate the sources before base services on the data are provided. For example, services such as keyword search should be available on all sources right from the start. If a higher level of semantic integration is required, then users of the DSSP should be able to add semantic relationships gradually, in a *pay-as-you-go* fashion.

The four key requirements for a DSSP are [65]: (1) to support *all* the data in the dataspace; (2) *not* to require full control of the data; (3) to provide best-effort query answers; (4) to enable users to increase the level of integration of the data in the dataspace if needed be. These requirements are in sharp contrast to current DBMS architectures. DBMSs focus on the subset of the data of an organization that may be cast into a fixed set of schemas; they require full control in order to offer services on the data; like traditional information-integration systems, they are all-or-nothing architectures, either providing exact query answers with respect to the schema or not being able to represent the data at all; they require up-front investment to define a schema for the data and offer little support to evolve the understanding users have of the data over time.

In this thesis, we focus on personal and social dataspaces. As we have argued previously, current systems are ill-equipped to handle the heterogeneity present in such environments. That is especially true as we consider that users are typically not data-integration experts and, therefore, are only willing to invest time in semantic integration if it can be done intuitively and if the benefits outweight the costs.

Inspired by the dataspace vision [65], this thesis investigates a novel information architecture that we call a *Personal Dataspace Management System (PDSMS)*. A PDSMS is a hybrid architecture in-between search engines and traditional information-integration systems. The high-level architecture of a PDSMS is shown in Figure 1.2. When a PDSMS is bootstrapped, its workings resemble the ones of a search engine: It represents *all* data in the sources in a common data model and provides a basic querying service over it. As we do not expect our users to be experts, this querying service understands an intuitive

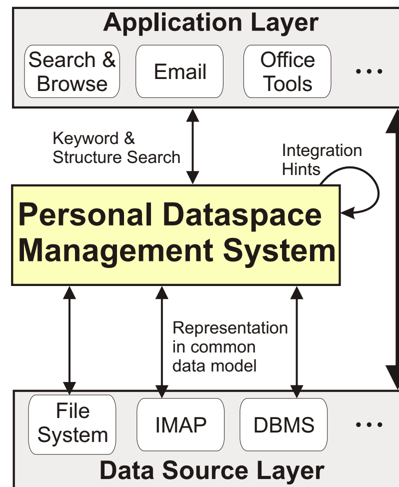


Figure 1.2: Architectural overview of a Personal Dataspace Management System (PDSMS).

query language for desktop users, with its core formed by keywords and paths. As with search engines, query results provided at this point are not precise. Moreover, direct access to the data sources is possible at all times. In contrast to search engines, however, over time the personal dataspace can be semantically enriched as the user provides integration "hints" to the system. These hints are *not* full-blown schema mappings, but rather small indications about how the items in the dataspace are related to one another. As hints are lightweight, they may be specified or shared by users and even suggested by the system by mining the contents of the dataspace. The PDSMS makes use of those hints to improve the quality of query results. This process is dynamic: As users perceive that query results are of unsatisfactory quality, they may provide more hints to the system, thus reaching a level of integration that is good enough for their particular scenarios.

1.4 Contributions of this Dissertation

The ultimate goal and contribution of this dissertation is to explore the design space for Personal Dataspace Management Systems, a novel breed of information-integration architecture that stands in-between search engines and traditional information-integration systems. In order to achieve this goal, we have concentrated our work around four major building blocks, schematically depicted in Figure 1.3. We outline the main contribution brought by each of those building blocks below.

1.4.1 `iMeMex` Data Model (iDM)

The first research challenge we face when dealing with the heterogeneous data mix found in personal and social dataspaces is to represent *all* data available in various data sources and formats in a common model. Our goal is to provide unified representation and unified querying on all the information available on the dataspace. Unlike in traditional information-integration systems, this unified query service should not

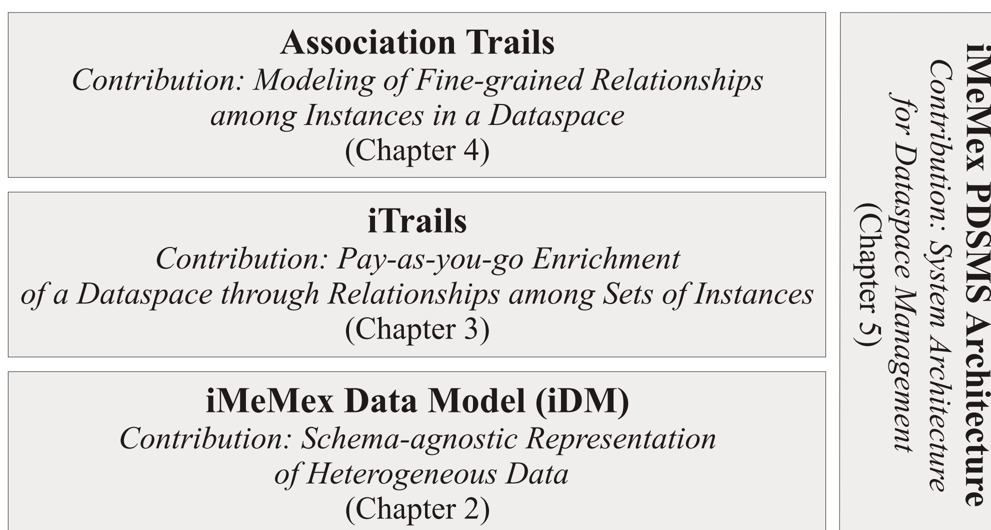


Figure 1.3: Schematic depiction of the main contributions of this dissertation.

depend upon semantic integration of the schemas of the sources; unlike in search engines, however, this unified query service should be able to take advantage of structural information available in the sources. For example, in personal desktops, it is currently not possible to pose queries that exploit structural information outside files in the folder hierarchy and at the same time structural information inside files. We address that challenge in this dissertation by representing all the heterogeneous information in the data sources into a single, powerful, yet simple graph-based data model, the iMeMex Data Model (iDM). In iDM, we favor a clear separation between the logical data model and the physical representation of data. In particular, data available in the user's personal dataspace need not be explicitly imported into iDM, but rather only represented into it. In line with the general PDSMS architecture discussed in the previous section, this logical data representation implies that a system built based on iDM does not need to take full control of the data, being able to provide additional querying services on data that remains in its original data sources. Of course, the system may still maintain indexes and replicas that speed up query processing. In fact, iDM simplifies the building of such a system, given that the logical graph defined in iDM is lazily computed. In order to query iDM, users may employ a simple search-and-query language that combines keyword search with structural restrictions. As iDM effectively removes the boundary between outside and inside of files, queries may be posed that combine structural information across that boundary as well as across data sources and formats.

1.4.2 iTrails

Representing all the information in the data sources in a common data model does not address the issue of semantic heterogeneity among sets of items in the sources. In fact, users may write search queries that return results across sources and formats, but these queries will have answers that are not precise. What we would like to offer to users is the possibility to gradually increase the level of integration of their personal and social dataspace. iTrails is a novel technique that targets this research challenge. The

core idea of iTrails is to enable users to provide integration "hints", called *trails*, to the system. These trails are exploited by the system to rewrite user queries and improve the precision and recall of search results. In contrast to traditional information-integration architectures, at no point do we require that a global mediated schema or complex schema mappings be created. Trails are small bits of integration information, given to the system in a pay-as-you-go fashion. When users specify trails, they relate arbitrary *subsets* of their dataspace. For example, a user could define a trail that relates all items in the path `//projects/PIM` to items in the path `//mike/research/PIM`. During query processing, queries that reference `//projects/PIM` would be rewritten by the trail to obtain also elements that lie in path `//mike/research/PIM`, unifying these distinct hierarchies. In personal and social dataspaces, we do not expect all trails to be directly input by the end-user. Trails could be automatically mined from the contents of the dataspace or shared among different user communities. Thus, we explore how to deal with uncertainty in the trails and how to scale the rewrite procedure to thousands of trail definitions.

1.4.3 Association Trails

While iTrails is an effective technique to model relationships among sets of items in a dataspace, it is often the case in personal and social dataspaces that we would like to model finer-grained relationships among individual instances. For example, we might want to connect each file in the user's desktop to the corresponding emails that were received around the same time that file was created. In a social dataspace, we might want to define that all people that attend the same yoga course are connected in this context. In addition, unlike in current systems, we would like to specify such relationships declaratively, without having to add edges to the dataspace graph connecting all affected instances explicitly. To address this research challenge, we propose *association trails*. Association trails enable users to declare fine-grained, instance-level relationships in a dataspace through general join semantics. A core advantage of association trails is that they model this graph of connections among instances intensionally. When processing exploratory queries on this graph, this declarative definition allows us to investigate several query processing techniques, including combining selective computation of portions of the intensional graph with query processing itself. As a result, processing costs may be over an order of magnitude lower than the naive strategy of materializing the full intensional graph and then processing queries over it. In addition, the intensional graph of connections among instances may be used as additional evidence for ranking query results as well as their neighborhoods. This enables us to provide to users the *context* in which query results appear on the dataspace, while at the same time being able to evolve the very definition of context declaratively.

1.4.4 iMeMex PDSMS Architecture

The final research challenge addressed in this thesis is the one of how to architect and design a Personal Dataspace Management System. This challenge involves specifying the inner workings of a general PDSMS as depicted in Figure 1.2. In particular, two design goals should be met. First, we would like to combine all of the contributions above into a consistent system architecture. Second, we would like to

design a highly extensible system, given that a PDSMS must interact with a multitude of heterogeneous data sources and formats. Moreover, the PDSMS should admit working in a number of configurations, such as when accessing sources that only offer a mediation query interface (query-shipping mode) or sources that allow the PDSMS to operate in warehousing (data-shipping mode) [66]. During the course of this dissertation, we have worked on the design and implementation of the iMeMex Personal Dataspace Management System. The iMeMex PDSMS follows a layered architecture with two main logical blocks: the Physical Independence Layer (PHIL) and the Logical Independence Layer (LIL). PHIL is responsible for abstracting from physical data sources, devices, and formats, allowing iMeMex to access a mix of file systems, email servers, network shares, RSS feeds, relational databases, web services, among others. The core idea of PHIL is to represent all of the heterogeneous information available in these sources in a common data model, the iMeMex Data Model (iDM). In that way, PHIL is able to hide representation details tied to each specific source and, in fact, may index and replicate data in the sources to speed up query processing. LIL is responsible for providing complex query and pay-as-you-go information-integration capabilities on top of PHIL. LIL achieves that goal by incorporating a query processor for a simple search-and-query language that interacts with iTrails and with association trails. All queries posed to iMeMex are thus rewritten by LIL to incorporate pay-as-you-go information-integration semantics. In order to achieve extensibility, both PHIL and LIL are componentized and implemented using the OSGi framework [129], a service-based platform for Java. Developers may plug in handlers for different sources and formats, as well as provide new index structures for iMeMex. Our system prototype is open-source under an Apache 2.0 License and available for download at http://www.imemex.org.

1.4.5 Publications

Previous versions of some chapters of this thesis have been published originally in international conferences. We list below these chapters along with the their respective previous publications:

- **Chapter 2 — iMeMex Data Model (iDM) [50]**: Jens-Peter Dittrich and Marcos Antonio Vaz Salles. iDM: A Unified and Versatile Data Model for Personal Dataspace Management. In *VLDB*, 2006.

- **Chapter 3 — iTrails [143]**: Marcos Antonio Vaz Salles, Jens-Peter Dittrich, Shant Kirakos Karakashian, Olivier René Girard, and Lukas Blunschi. iTrails: Pay-as-you-go Information Integration in Dataspaces. In *VLDB*, 2007.

- **Chapter 5 — iMeMex Architecture [19]**: Lukas Blunschi, Jens-Peter Dittrich, Olivier René Girard, Shant Kirakos Karakashian, and Marcos Antonio Vaz Salles. A Dataspace Odyssey: The iMeMex Personal Dataspace Management System. In *CIDR*, 2007. Demo Paper.

1.5 Structure of this Dissertation

The organization of this dissertation follows the contributions outlined above. In Chapter 2, we introduce the iMeMex Data Model (iDM) and discuss how it enables a schema-agnostic representation of the heterogeneous data mix typically found in personal dataspaces. As a natural next step, we build the iTrails framework on top of iDM. In Chapter 3, we present how iTrails allow users to gradually enrich a loosely integrated dataspace by adding set-level relationships. Fine-grained, instance-level relationships are the subject of Chapter 4, in which we introduce association trails. Chapter 5 reviews the architecture of the iMeMex Personal Dataspace Management System and discusses how our contributions can be integrated into a consistent system design. Finally, we present closing remarks and outline directions for future research in Chapter 6.

Chapter 2

iDM: A Unified and Versatile Data Model for Personal Dataspace Management

Personal Information Management Systems require a powerful and versatile data model that is able to represent a highly heterogeneous mix of data such as relational data, XML, file content, folder hierarchies, emails and email attachments, data streams, RSS feeds and dynamically computed documents, e.g. ActiveXML [4]. Interestingly, until now no approach was proposed that is able to represent all of the data above in a single, powerful, yet simple data model. This chapter fills this gap. We present the `iMeMex` Data Model (iDM) for personal information management. With iDM, we are able to represent unstructured, semi-structured, and structured data inside a single model. Moreover, iDM is powerful enough to represent graph-structured data, intensional data, as well as infinite data streams. Furthermore, our model enables users to represent the structural information available inside files. As a consequence, the artifical boundary between inside and outside a file is removed to enable a new class of queries. As iDM allows for the representation of the whole personal dataspace [65] of a user in a single model, it is the foundation of the `iMeMex` Personal Dataspace Management System (PDSMS), presented in Chapter 5. This chapter also presents results of an evaluation of an initial iDM implementation in `iMeMex` that show that iDM can be efficiently supported in a real PDSMS.

2.1 Introduction

2.1.1 Motivation

Personal information consists of a highly heterogeneous data mix of emails, XML, LaTeX and word documents, pictures, music, address book entries, and so on. Personal information is typically stored in files scattered among multiple file systems (local or network), multiple machines (local desktop, network share, mail server), and most of all different file formats (XML, LaTeX, Office, email formats).

Much of the data stored in those files represents information such as hierarchies and graphs that is not

exploited by current PIM tools to narrow search and query results. Moreover, there is an artificial gap between the structural information *inside* files and the *outside* structural information established by the directory hierarchies employed by the user to classify her files.

This chapter proposes an elegant solution to close these gaps. Key to our approach is to map all available data to a single graph data model — no matter whether it belongs *inside* or *outside* a file. For instance, our model may be applied to represent at the same time data pertaining to the file system (folder hierarchies) and data pertaining to the contents inside a file (structural graph). Consequently, the artifical boundary between inside and outside a file is removed. At the same time, we also remove the boundary between different subsystems such as file systems and IMAP email servers as we map that information to the same graph model. This enables new kinds of queries that are not supported by state-of-the-art PIM tools.

2.1.2 The Problem

EXAMPLE 2.1 [INSIDE VERSUS OUTSIDE FILES] Personal Information includes semi-structured data (XML, Word or other Office documents[1]) as well as graph-structured data (LaTeX). These documents are stored as files on the file system. Consider the following query:

> Query 1: “Show me all LaTeX ‘Introduction’ sections pertaining to project PIM that contain the phrase ‘Mike Franklin’.”

The query references information that is partly kept *outside* files on the file system, i.e., all project folders related to the PIM project. Another part of the query references information kept *inside* certain LaTeX files, i.e., all introduction sections containing the phrase ‘Mike Franklin’. With current technology, this query cannot be issued in one single request by the user as it has to bridge that *inside-outside file boundary*. The user may only search the file system using simple system tools such as grep, find, or a keyword search engine. However, these tools may return a large number of results that would have to be examined manually to determine the final result. Even when a matching file is encountered, for structured file formats like Microsoft PowerPoint, the user then typically has to conduct a second search inside the file to find the desired information [42]. In fact, current tools are very sensitive to the way information is stored by users when processing queries. In our example above, it might have been feasible to process the query with a keyword search engine if the user had carefully separated all sections of her documents in different files. In other words, if the user had manually shifted the inside-outside file boundary, more of the structural information would be exposed to current tools. To sum up, state-of-the-art operating systems do not support at all exploitation of structured information *inside* the user’s documents. The structured information inside the files is in a data cage and cannot be used to refine the query.

Desiderata: What is missing is a technology that enables users to execute queries that bridge the divide between the graph-structured information inside their files and the outside file system. □

[1] Open Office has stored documents in XML since version 1.0. MS Office 2007 also enables storage of files using zipped XML.

EXAMPLE 2.2 [FILES VERSUS EMAIL ATTACHMENTS] Email has become one of the most important applications of personal information management. Consider a simple project management scenario. When managing multiple projects, e.g., industry projects, research projects, PhD students, lectures, and so on, you may decide to store documents of big projects in a separate folder on your local hard disk. For smaller projects you may decide to keep that information as attachments to email messages you exchanged with members of the project team. Now let us consider the following query:

> Query 2: "Show me all documents pertaining to project 'OLAP' that have a figure containing the phrase 'Indexing Time' in its label."

With state-of-the-art technology this type of query cannot be computed as it requires the system to consider *structural information* available on the different folder hierachies *outside*, i.e., both the folder hierarchy on the local hard disk as well as the folder hierarchy on the IMAP server. Again, the structural constraint "all Figures containing" has to be evaluated considering *structural information* present *inside* certain files, i.e. locally available files and email attachments. In addition, emails have both an *outside* nature, as they contain connections to email attachments, and an *inside* nature, as they have content themselves. As current search engines do not abstract from the inside-outside file boundary when representing personal data, these tools make it unnatural for users to write queries that consider both the structural features of email as well as its content.

Desiderata: What is missing is a technology that abstracts from the different *outside* subsystems, such as file systems and email servers, to enable a unified approach to querying graph-structured personal information. □

2.1.3 Contributions

This chapter presents an elegant solution to the problems above. The core idea is to introduce a unified, versatile, and yet simple data model. All personal information such as semi-structured documents (LaTeX, Word, or other Office documents), relational data, file content, folder hierarchies, email, RSS feeds, and even data streams can be instantiated in that model. Since we are able to represent all data inside a single model, we are then in the position to use a single powerful query language to query all data within a single query. This chapter presents our data model. In addition, we also specify a search-oriented subset of our query language. The full specification of our query language will be presented as part of future work.

In summary, this chapter makes the following contributions:

1. We present the `iMeMex` *Data Model (iDM)* for personal information management. With iDM, we allow for a unified representation of all personal information such as XML, relational data, file content, folder hierarchies, email, data streams, and RSS feeds inside a single data model.
2. We present how to instantiate existing data such as XML, relations, files, folders, and data streams inside our model. Furthermore, we show how to instantiate iDM subgraphs based on the structural content of files such as tree structures (XML) and graph structures (LaTeX).

3. We show that all parts of the *iMeMex Data Model (iDM)* can be computed lazily. Because of this characteristic, iDM is able to support so-called *intensional data*, i.e., data that is obtained by executing a query or calling a remote service. For this reason, iDM can model approaches such as Active XML [4] as a use-case. In contrast to the latter, however, iDM is not restricted to the XML domain.
4. We have used iDM as the foundation for the *iMeMex Personal Dataspace Management System (PDSMS)*, presented in Chapter 5. In order to query iDM graphs in iMeMex, we present how to express search queries using a simple query language, the iMeMex Query Language (iQL).
5. Using an initial iDM implementation in iMeMex, we present results of experiments. The results of our experiments demonstrate that iDM can be efficiently implemented in a real PDSMS, providing both efficient indexing times and interactive query-response times.

This chapter is structured as follows. Section 2.2 presents an overview and the formal definition of the *iMeMex Data Model (iDM)* for personal information management. Section 2.3 then describes how to represent XML, files, folders, as well as data streams using iDM. After that, Section 2.4 discusses how to compute an iDM graph. Section 2.5 gives an overview on how to express search queries on iDM graphs with the query language iQL. Section 2.7 reviews related work. Section 2.6 presents experiments with our initial iDM implementation in the iMeMex Personal Dataspace Management System. Finally, Section 2.8 concludes the chapter.

2.2 iMeMex Data Model

This section is structured into an overview (2.2.1), the formal definition of our model (2.2.2), and a series of examples (2.2.3).

2.2.1 Overview

The iMeMex Data Model (iDM) uses the following important ideas to represent the heterogeneous data mix found in personal information management:

- In iDM, all personal information available on a user's desktop is exposed through a set of *resource views*. A resource view is a sequence of components that express *structured*, *semi-structured*, and *unstructured* pieces of the underlying data. Thus, all personal data items, though varied in their representation, are exposed in iDM uniformly. For example, every node in a files&folders hierarchy as well as every element in an XML document would be represented in iDM by one distinct resource view.

- Resource views in iDM are linked to each other in arbitrary directed *graph structures*. The connections from a given resource view to other resource views are given by one of its components.

- In contrast to XML approaches such as ActiveXML [4], iDM does not impose the need to convert data to a physical XML document representation before query processing may take place. Rather, we favor a clear *separation between logical and physical representation* of data. Data may be dynamically represented in our model during query processing although it remains primarily stored in its original format and location. Note that this logical representation does not preclude a system that implements iDM providing facilities such as indexing or replication of the original data to speed up query evaluation.

- We introduce *resource-view classes* to precisely define the types and representation of resource view components. Any given resource view may or may not conform to a resource-view class. We show how to use resource-view classes to constrain our model to represent data in files and folders, relations, XML, and data streams in Section 2.3.

- Resource views may be given *extensionally* (e.g., files and folders or tuples in a relational store) or computed *intensionally* (e.g., as a result to a query). Furthermore, resource views may contain components that are *finite* as well as *infinite*. Those aspects of our model are explored in Section 2.4.

2.2.2 Resource Views

In this section, we formally define a resource view and explain its constituent parts. After that, we present examples of resource-view graphs (see Section 2.2.3).

DEFINITION 2.1 (RESOURCE VIEW) *A* resource view V_i *is a 4-tuple* $(\eta_i, \tau_i, \chi_i, \gamma_i)$*, where* η_i *is a name component,* τ_i *is a tuple component,* χ_i *is a content component, and* γ_i *is a group component.*[2] *We define each component of a resource view* V_i *as follows:*

1. NAME COMPONENT*:* η_i *is a finite string that represents the name of* V_i.
2. TUPLE COMPONENT*:* τ_i *is a 2-tuple* (W,T)*, where* W *is a schema and* T *is one single tuple that conforms to* W*. The schema* $W = \langle a_j \rangle, j = 1,2,\dots,k$ *is defined as a sequence of attributes where attribute* a_j *is the name of a role played by some domain*[3] D_j *in* W*. The tuple* $T = \langle v_j \rangle, j = 1,2,\dots,k$ *is a sequence of atomic values where value* v_j *is an element of the domain* D_j *of the attribute* a_j*.*
3. CONTENT COMPONENT*:* χ_i *is a string of symbols taken from an alphabet* Σ_c*. The content* χ_i *may be finite or infinite. When* χ_i *is finite, it takes the form of a finite sequence of symbols denoted by* $\langle c_1,\dots,c_l \rangle, c_j \in \Sigma_c, j = 1,...,l$*; when* χ_i *is infinite, the respective sequence is infinite and we denote* $\chi_i = \langle c_1,\dots,c_l \rangle_{l\to\infty}, c_j \in \Sigma_c, j = 1,\dots,l,\ l \to \infty$.
4. GROUP COMPONENT*:* γ_i *is a 2-tuple* (S,Q)*, where* S *is a (possibly empty)* ***set*** *of resource views and* Q *is a (possibly empty) ordered* ***sequence*** *of resource views. Furthermore,*

[2]We use the subscript notation to denote one specific resource view and its components.

[3]A domain is considered to be a set of atomic values. In the remainder, whenever we mention attributes or domains, we refer to the definitions given above. Our definitions of domains and attributes conform to the ones given by Elmasri and Navathe [59].

(i) *The set S and the sequence Q may be finite or infinite. When S is finite, we denote* $S = \{V_1, \ldots, V_m\}$*; when it is infinite, then we denote* $S = \{V_1, \ldots, V_m\}_{m\to\infty}$*. Likewise, when Q is finite, we denote* $Q = \langle W_1, \ldots, W_n \rangle$*; when Q is infinite, then we denote* $Q = \langle W_1, \ldots, W_n \rangle_{n\to\infty}$*.*
(ii) $S \cap Q = \emptyset$*, i.e., S and Q are disjoint.*[4]
(iii) *Assume a resource view* V_i *has a non-empty* γ_i *component. If there exists a resource view* V_k *for which* $V_k \in S \cup Q$ *holds, we say that* V_k *is* ***directly related*** *to* V_i*, i.e.,* $V_i \to V_k$*. Any given resource view may be directly related to zero, one, or many other resource views.*
(iv) *If* $V_i \to V_j \to \ldots \to V_k$*, we say that resource view* V_k *is* ***indirectly related*** *to* V_i*, denoted* $V_i \rightsquigarrow V_k$*.*

If any of the components of a resource view is empty, we denote its value, where convenient, by the empty n-tuple $()$ *or by the empty sequence* $\langle\rangle$*.* □

The η_i component is a non-unique name used to refer to the resource view. It is of service for the construction of path queries, discussed in Section 2.5. The τ_i component has a similar definition as the one given for tuples in a relational data model [36, 59]. One important difference is that the schema W is defined for *each* tuple, instead of for a set of tuples. This decision of course creates a tension for representing sets of resource views with the same structural features. Schematic information is important in this setting and we introduce it in our model by defining *resource view classes* (see Section 2.3).

The χ_i component represents arbitrary unstructured content. For iDM, it is merely a sequence of atomic symbols from some alphabet, such as characters in a file's content or in an XML text node. The χ_i component may be finite or infinite. Allowing infinite content is useful to naturally represent media streams in our model.

Finally, the γ_i component creates a directed graph structure that connects related views. Note that we impose no restriction on this graph, so that we may represent trees, DAGs, and cyclic graphs. If the relative order among the connections established between resource views is of importance, we represent them in the sequence Q of γ_i. Otherwise, they are represented in the set S. Like the χ_i component, both the set and the sequence of γ_i may be finite or infinite. The infinite case is useful for representing data streams as infinite sequences of resource views in our model.

2.2.3 Examples

In Figure 2.1(a), we present a typical files-and-folders hierarchy containing information on some research projects of a user. There is one high-level folder that groups all research projects and we show two subfolders for specific projects. The 'PIM' folder is further expanded to reveal one LaTeX document for one version of the VLDB 2006 paper that corresponds to this chapter, one Microsoft Word document for a grant proposal, and one folder link to the top-level 'Projects' folder. The contents of those documents are also partially displayed. Both documents contain sections, subsections and their corresponding

[4] For the sake of simplicity, we extend notation and use the symbols ∩ and ∪ to denote intersection not only between sets, but also between a set and a sequence. The ∩ operation returns a set containing the (distinct) elements of the sequence that also belong to the intersected set. The ∪ operation returns a set containing the (distinct) elements that belong either to the set or to the sequence.

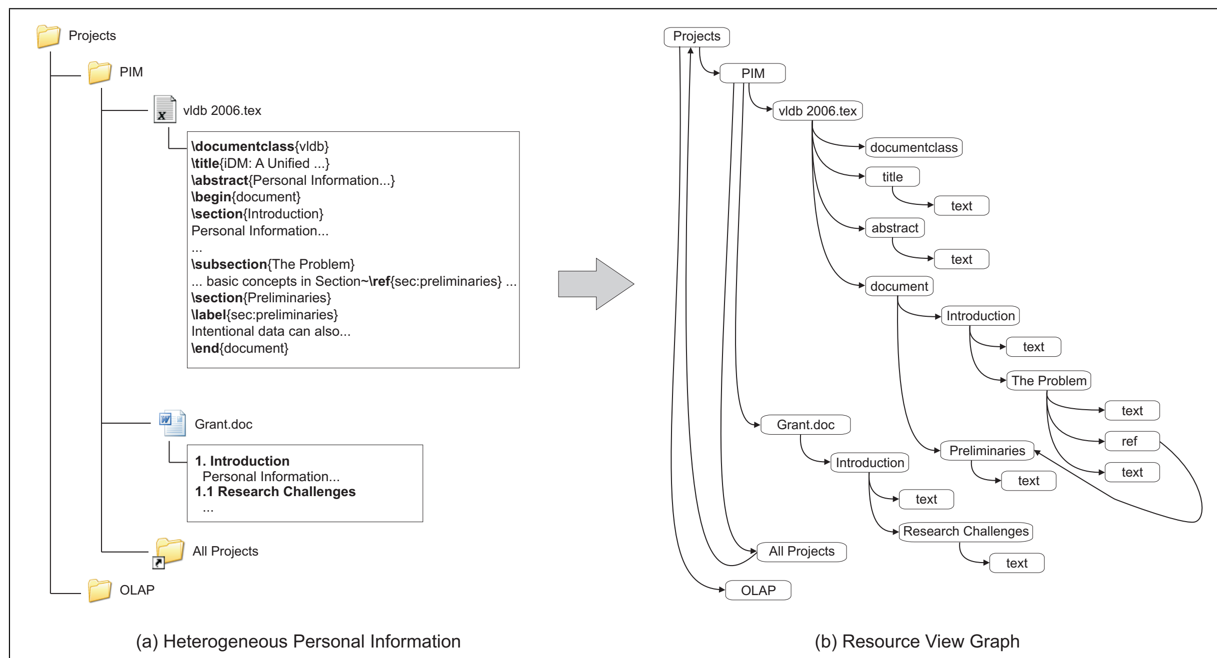


Figure 2.1: iDM represents heterogeneous personal information as a single resource view graph. The resource view graph represents the whole dataspace of a user

text. In addition, the document 'vldb2006.tex' also contains a reference to section 'Preliminaries' in the subsection 'The Problem'.

Unified Representation. As we may notice in Figure 2.1(a), there is a gap between the graph-structured information *inside* files and the hierarchies present on the *outside* file system. We show the representation of the same information in iDM in Figure 2.1(b). Nodes denote resource views and edges denote the connections induced by the resource views' group components. Each node is labeled with the resource view's name component.

In iDM, what we call files and folders are only resource views. Each file or folder is represented as one resource view in Figure 2.1(b). Furthermore, the data stored inside files is also uniformly represented as resource views. The document class, title, abstract, and document portions of the VLDB 2006 paper are some of these resource views. They are directly related to the 'vldb 2006.tex' file resource view. In addition, any structural information present in the document portion of that file is represented as resource views. Note that the same applies to the 'Grant.doc' resource view. Extracting this structural information is possible as an increasing number of office tools, such as Microsoft Office 2007, Open Office, and LaTeX, offer semi-structured or graph-structured document formats that make it feasible to write content converters that obtain good-quality resource-view graphs. In addition, structure-extraction techniques [41, 56, 117] may be used in conjunction with our approach to further increase the quality and coverage of resource view subgraphs extracted from content components.

Intensional Data. One important aspect of our model is that the resource view graph is a *logical* repre-

sentation of personal information. That logical representation does not need to be fully materialized at once and may be computed lazily. Personal information does not have to be imported to or exported from our data model, but rather represented in it. This logical approach is in sharp contrast to XML approaches which are usually tied to having a physical representation of the whole data as an XML document before querying may be carried out (see Section 2.4).

Resource Views and their Components. Let us show the detailed representation of one resource view present in Figure 2.1. Consider the 'PIM' folder. We represent it as a resource view $V_{\text{PIM}} = (\eta_{\text{PIM}}, \tau_{\text{PIM}}, \chi_{\text{PIM}}, \gamma_{\text{PIM}})$, in which:

$\eta_{\text{PIM}} =$ 'PIM';

$\tau_{\text{PIM}} =$ (W,T), where $W = \langle$creation time: date, size: integer, last modified time: date$\rangle$ and $T = \langle$'19/03/2005 11:54', 4096, '22/09/2005 16:14'$\rangle$, i.e., the τ_{PIM} component represents the filesystem properties associated with the 'PIM' folder;

$\chi_{\text{PIM}} = \langle\rangle$;

$\gamma_{\text{PIM}} =$ (S,Q), where $S = \{V_{\text{vldb 2006.tex}}, V_{\text{Grant.doc}}, V_{\text{All Projects}}\}$ and $Q = \langle\rangle$.

Note that the children of the 'PIM' folder in the filesystem are represented as resource views directly related to V_{PIM}. These resource views are $V_{\text{vldb 2006.tex}}$, $V_{\text{Grant.doc}}$, and $V_{\text{All Projects}}$. The resource views $V_{\text{vldb 2006.tex}}$ and $V_{\text{Grant.doc}}$ have their η- and τ-components defined analogously to V_{PIM}, their γ-component corresponding to the resource views in their document content, and their χ-component equal to the binary stream of each file. The $V_{\text{All Projects}}$ resource view is also represented analogously to V_{PIM}, except that its γ-component corresponds to the folder 'Projects'. Here and in the remainder, we will omit the empty components from the resource view notation as they are clear from the context. Therefore, we denote $V_{\text{PIM}} = (\text{'PIM'}, \tau_{\text{PIM}}, \gamma_{\text{PIM}})$. We also use V_{PIM} to represent a view named 'PIM' where there is no risk of confusion with other views with the same name.

Graph Structures. Some data models, such as XML or traditional files and folders, organize data in a tree structure. The extension of that structure to represent graphs is usually done through the inclusion of *second-class citizens* in the model, such as links. With iDM, we naturally represent graph structures by the connections induced by the resource views' group components. The relevance of representing and querying arbitrary graph data has been increasingly recognized in the literature [78, 100, 126, 164].

For example, the $V_{\text{Projects}} \rightarrow V_{\text{PIM}} \rightarrow V_{\text{All Projects}} \rightarrow V_{\text{Projects}}$ path of directly related resource views in Figure 2.1(b) forms a cycle in the resource view graph. Furthermore, in Figure 2.1(b), resource view $V_{\text{Preliminaries}}$ is directly related to both views V_{document} and V_{ref}.

Schemas. Personal information is trapped inside a large array of data cages. Some solutions to this problem propose to cast user's information into a rigid set of developer-defined schemas, e.g. Microsoft WinFS [165]. They assert that the relational model is the most appropriate underlying representation for desktop data. Other proposals, e.g., Apple Spotlight [152], abolish schemas and employ search engine technology to represent all data as completely unstructured.

In contrast, iDM offers a schema-agnostic graph representation of personal information. We may employ resource view classes, defined in the following section, to enrich iDM with schematic information and to constrain it to represent a wide array of data models typically used to represent personal data.

2.3 Instantiating Specialized Data Models

Resource View Class		**Resource View Components Definition**				
		η_i^C	τ_i^C	χ_i^C	γ_i^C	
Description	**Name**				S	Q
File	file	N_f	(W_{FS}, T_f)	C_f	$\varnothing$	$\langle\rangle$
Folder	folder	N_F	(W_{FS}, T_F)	$\langle\,\rangle$	$\{V_1^{\text{child}}, \ldots, V_m^{\text{child}}\}$ child $\in$ {file, folder}	$\langle\rangle$
Relational Tuple	tuple	$\langle\,\rangle$	(W_R, t_i)	$\langle\,\rangle$	$\varnothing$	$\langle\rangle$
Relation	relation	N_R	$(\,)$	$\langle\,\rangle$	$\{V_1^{\text{tuple}}, \ldots, V_m^{\text{tuple}}\}$ $V_i^{\text{tuple}} = \langle\tau_i^{\text{tuple}}\rangle, \tau_i^{\text{tuple}} = (W_R, t_i)$, $i = 1, \ldots, m$	$\langle\rangle$
Relational database	reldb	N_{DB}	$(\,)$	$\langle\,\rangle$	$\{V_1^{\text{relation}}, \ldots, V_m^{\text{relation}}\}$	$\langle\rangle$
XML text node	xmltext	$\langle\,\rangle$	$(\,)$	C_t	$\varnothing$	$\langle\rangle$
XML element	xmlelem	N_E	(W_E, T_E)	$\langle\,\rangle$	$\varnothing$	$\langle V_1^{\text{xmlnode}}, \ldots, V_n^{\text{xmlnode}}\rangle$ xmlnode $\in$ {xmltext, xmlelem}
XML document	xmldoc	$\langle\,\rangle$	$(\,)$	$\langle\,\rangle$	$\varnothing$	$\langle V_{\text{root}}^{\text{xmlelem}}\rangle$
XML File	xmlfile	N_f	(W_{FS}, T_f)	C_f	$\varnothing$	$\langle V_{\text{doc}}^{\text{xmldoc}}\rangle$
Data Stream	datstream	$\langle\,\rangle$	$(\,)$	$\langle\,\rangle$	$\varnothing$	$\langle V_1, \ldots, V_n\rangle_{n\to\infty}$
Tuple stream	tupstream	$\langle\,\rangle$	$(\,)$	$\langle\,\rangle$	$\varnothing$	$\langle V_1^{\text{tuple}}, \ldots, V_n^{\text{tuple}}\rangle_{n\to\infty}$
RSS/ATOM stream	rssatom	$\langle\,\rangle$	$(\,)$	$\langle\,\rangle$	$\varnothing$	$\langle V_1^{\text{xmldoc}}, \ldots, V_n^{\text{xmldoc}}\rangle_{n\to\infty}$ or: same as in xmldoc

Table 2.1: Important Resource View Classes to represent files and folders, relations, XML, data streams, and RSS

In the following, we show how to instantiate specialized data models such as files and folders, XML, as well as data streams using iDM. Table 2.1 summarizes the instantiations for these specialized data models. For brevity, we omit a detailed discussion on how to instantiate relational data and only present its formal specification in Table 2.1. These additional instantiations, however, will become clear to the reader based on the discussion for the other data models.

2.3.1 Resource-View Classes

For convenience, we introduce *resource-view classes*, which constrain iDM to represent a particular data model.

DEFINITION 2.2 (RESOURCE-VIEW CLASS) *A* resource-view class *C is a set of formal restrictions on the* η-, τ-, χ-, *and* γ-*components of resource views. The formal restrictions specified by a resource-view class include:*

1. EMPTINESS OF COMPONENTS: *specifies which subset of the components of resource views that obey the resource-view class must be empty and which must not.*
2. SCHEMA OF τ: *determines the schema W that the* τ *components of resource views must have.*
3. FINITENESS OF χ OR γ: *enforces content* χ *or group* γ *elements S or Q to be either finite, infinite or empty.*
4. CLASSES OF DIRECTLY RELATED RESOURCE VIEWS: *given a resource view* V_i, *determines the set of resource-view classes that are acceptable for any resource view* V_j *such that* $V_i \rightarrow V_j$.
5. OPTIONALITY OF ELEMENTS IN γ: *determines that certain elements either in the set or in the sequence of* γ *are optional.*
6. MAPPINGS OF COMPONENTS: *given an underlying data model D being represented in iDM, specifies a mapping of elements of D to resource view components.*

When a resource view V_i *conforms to the resource-view class C, we denote it* V_i^C. *Accordingly, we denote its components* η_i^C, τ_i^C, χ_i^C, *and* γ_i^C.

A given resource view may conform directly to only one class. One may, however, organize resource-view classes in generalization (or specialization) hierarchies. When a resource view conforms to a given class C, it automatically conforms to all classes that are generalizations of C. In this sense, our classes have an object-oriented flavor [30]. As with object-oriented classes, when a resource-view class C_1 specializes a resource-view class C_2, then C_1 inherits the formal restrictions of C_2. Moreover, C_1 may include additional restrictions or redefine a subset of C_2's restrictions.

Resource-view classes may be used by application developers to provide pre-defined schema information on sets of resource views. Note that a full specification of schemas, including several different categories of integrity constraints, is an interesting avenue for future work. Note, additionally, that not all resource views need to have a resource-view class attached to them. As a consequence, unlike the relational [36] or object-oriented approaches [30], which support only a schema-first data modeling strategy, our model also supports schema-later and even schema-never strategies [65].

2.3.2 Files and Folders

Traditional filesystems can be seen as trees in which internal nodes represent non-empty folders and leaves represent files or empty folders. Some filesystems also include the concept of links, which may transform the filesystem into a more general graph structure. We begin by discussing how to represent a tree of files and folders, without links, in iDM. Each node in the tree has a fixed set of properties, such as size, creation time, last-modified time, etc. The attributes that appear on each node are defined in a filesystem-level schema $W_{FS} = \{$size: int, creation time: date, last modified time: date, ...$\}$. The sequence of values that conform to that schema are expressed per node and, for a node n, we denote this sequence T_n. Each node also has a name, denoted N_n. In addition, if n is a file node, then it has an associated content C_n. In iDM we consider a "file" just one out of many possible resource views on user data. In

order to have iDM represent the files&folders data model we define a *file resource view class*, denoted file, that constrains a view instance $V_{\mathsf{i}}^{\mathsf{file}}$ to represent a file f as follows:

$$V_{\mathsf{i}}^{\mathsf{file}} = (\eta_{\mathsf{i}}^{\mathsf{file}}, \tau_{\mathsf{i}}^{\mathsf{file}}, \chi_{\mathsf{i}}^{\mathsf{file}}), \text{ where:}$$
$$\eta_{\mathsf{i}}^{\mathsf{file}} = N_f, \tau_{\mathsf{i}}^{\mathsf{file}} = (W_{FS}, T_f), \text{ and } \chi_{\mathsf{i}}^{\mathsf{file}} = C_f.$$

Based on this definition we can recursively define the concept of a 'folder'. A *folder resource view class*, denoted folder, constrains a view instance $V_{\mathsf{i}}^{\mathsf{folder}}$ to represent a folder F as follows:

$$V_{\mathsf{i}}^{\mathsf{folder}} = (\eta_{\mathsf{i}}^{\mathsf{folder}}, \tau_{\mathsf{i}}^{\mathsf{folder}}, \gamma_{\mathsf{i}}^{\mathsf{folder}}), \text{ where:}$$
$$\eta_{\mathsf{i}}^{\mathsf{folder}} = N_F, \tau_{\mathsf{i}}^{\mathsf{folder}} = (W_{FS}, T_F), \text{ and}$$
$$\gamma_{\mathsf{i}}^{\mathsf{folder}} = \left(\{V_1^{\mathsf{child}}, \dots, V_{\mathsf{m}}^{\mathsf{child}}\}, \langle\rangle\right), \ \mathsf{child} \in \{\mathsf{file}, \mathsf{folder}\}.$$

In order to represent filesystems that contain links, we may create in iDM one separate resource view for each link in the filesystem. The link name is mapped to the resource view name and the link target is represented in the resource view's group component.

Furthermore, as discussed in Section 2.2, iDM may be used to exploit semi-structured content inside files. One special case of this type of content is XML. We may thus specialize the file resource view class to define an *XML file resource view class*, denoted $\mathsf{xmlfile}$, in which the group component is non-empty and defined as:

$$\gamma_{\mathsf{i}}^{\mathsf{xmlfile}} = \left(\varnothing, \langle V_{\mathsf{doc}}^{\mathsf{xmldoc}} \rangle\right).$$

The resource view $V_{\mathsf{doc}}^{\mathsf{xmldoc}}$ pertains to resource-view class xmldoc and is the document resource view for the XML document. The xmldoc resource-view class is defined in Section 2.3.3. Note that other specializations of the file resource-view class may be defined analogously for other document formats, e.g., LaTeX.

2.3.3 XML

We assume that XML data is represented according to the definitions in the XML Information Set [171]. For brevity, we only discuss how to instantiate the core subset of the XML Information Set (document, element, attribute, character) in iDM.

In order to have iDM represent the XML data model, we first define an *XML text resource view class*, denoted $\mathsf{xmltext}$, that constrains a view instance $V_{\mathsf{i}}^{\mathsf{xmltext}}$ to represent a character information item with text content $C_t = \langle c_1, \dots, c_n \rangle$ as follows:

$$V_{\mathsf{i}}^{\mathsf{xmltext}} = (\chi_{\mathsf{i}}^{\mathsf{xmltext}}), \text{ where: } \chi_{\mathsf{i}}^{\mathsf{xmltext}} = C_t.$$

As a second step, we need to represent element information items in our model. An element information item E has a name N_E, a set of attributes with schema W_E and values T_E, and a sequence of children, each

of which is either a text or an element information item. We define an *XML element resource view class*, xmlelem, that constrains a view instance V_i^{xmlelem} to represent an element information item E as follows:

$$
\begin{aligned}
V_i^{\text{xmlelem}} = (\eta_i^{\text{xmlelem}}, \tau_i^{\text{xmlelem}}, \gamma_i^{\text{xmlelem}}),\ \text{where:} \\
\eta_i^{\text{xmlelem}} = N_E,\ \tau_i^{\text{xmlelem}} = (W_E, T_E),\ \text{and} \\
\gamma_i^{\text{xmlelem}} = (S, Q), S = \varnothing, \\
Q = \left\langle V_1^{\text{xmlnode}}, \ldots, V_n^{\text{xmlnode}} \right\rangle, \\
\text{xmlnode} \in \{\text{xmltext}, \text{xmlelem}\}.
\end{aligned}
$$

Note that the XML attribute nodes are modeled by the τ_i^{xmlelem} component of V_i^{xmlelem}. Finally, we can define an *XML document resource view class*, xmldoc, to represent a document information item as follows:

$$
\begin{aligned}
V_i^{\text{xmldoc}} = (\gamma_i^{\text{xmldoc}}),\ \text{where:} \\
\gamma_i^{\text{xmldoc}} = \left(\varnothing, \langle V_{\text{root}}^{\text{xmlelem}} \rangle\right),\ \text{and}
\end{aligned}
$$

$V_{\text{root}}^{\text{xmlelem}}$ is a view that represents the root element information item of the document. Figure 2.2 shows an

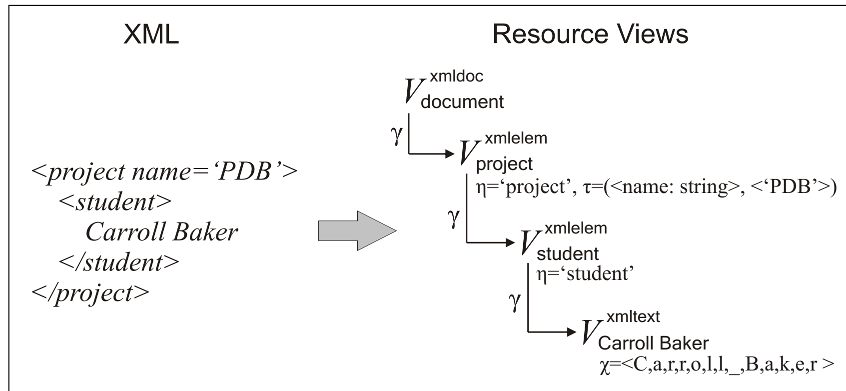


Figure 2.2: XML fragment represented as a resource view graph

example of an instantiation of an XML fragment in iDM. Each node in the XML document is represented as a resource view. We use an expanded notation in the figure to represent nodes in the resource view graph and explicitly indicate their components. The connections among views are given by the resource views' γ components.

2.3.4 Data Streams

We assume that a data stream is an infinite data source delivering data items. For the moment we do not require any restrictions on the type of data items. To constrain iDM to represent a generic data-stream model, we define a **generic data-stream** resource-view class, datstream, that restricts a resource view $V_i^{\text{datstream}}$ as follows:

$$
V_i^{\text{datstream}} = (\gamma_i^{\text{datstream}}),\ \text{where: } \gamma_i^{\text{datstream}} = \left(\varnothing, \langle V_1, \ldots, V_n \rangle_{n \to \infty}\right).
$$

Figure 2.3 shows schematically how the instantiation of a data stream occurs in iDM. Each data item delivered by that stream is represented as a resource view. Depending on the data model in which the item is represented, the corresponding resource views may be of different classes. The data stream itself is represented as a resource view whose γ component contains all corresponding data item views.

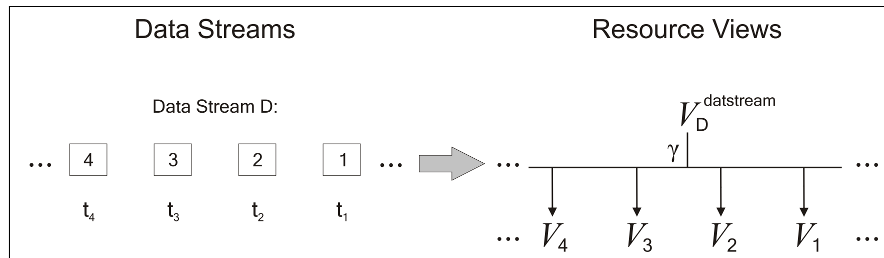


Figure 2.3: A data stream is represented as a resource view graph

Examples of data streams include streams that deliver tuples. A **tuple-stream** resource-view class, tupstream, is defined to restrict a view instance $V_{\mathrm{i}}^{\mathrm{tupstream}}$ as follows:

$$V_{\mathrm{i}}^{\mathrm{tupstream}} = (\gamma_{\mathrm{i}}^{\mathrm{tupstream}}), \text{ where:}$$
$$\gamma_{\mathrm{i}}^{\mathrm{tupstream}} = \big(\varnothing, \langle V_{1}^{\mathrm{tuple}}, \ldots, V_{\mathrm{n}}^{\mathrm{tuple}} \rangle_{n\to\infty}\big).$$

Another example of a data stream is an RSS/ATOM stream delivering XML messages. An **RSS stream** resource-view class, rssatom, is defined to restrict a resource view $V_{\mathrm{i}}^{\mathrm{rssatom}}$ as follows:

$$V_{\mathrm{i}}^{\mathrm{rssatom}} = (\gamma_{\mathrm{i}}^{\mathrm{rssatom}}),$$
$$\gamma_{\mathrm{i}}^{\mathrm{rssatom}} = \big(\varnothing, \langle V_{1}^{\mathrm{xmldoc}}, \ldots, V_{\mathrm{n}}^{\mathrm{xmldoc}} \rangle_{n\to\infty}\big).$$

Note that RSS/ATOM streams, even though they are frequently called ‘streams’, are just simple XML documents that are provided by a growing number of web servers. There is no notification mechanism for the clients interested in those documents. So, one may argue that an alternative representation for an RSS/ATOM stream is the same as given for an XML document (see also Table 2.1).

2.4 Computing the iDM Graph

This section outlines how to compute resource views and resource-view graphs. In the following, we show that all components of a resource view may be computed lazily (Section 2.4.1). After that, we discuss three paradigms to compute resource-view components: *extensional components* (Section 2.4.2), *intensional components* (Section 2.4.3), and *infinite components* (Section 2.4.4).

2.4.1 Lazy Resource Views

It is important to understand that all components of a resource view may be computed on demand (also known as lazily or intensionally). We do *not* require that any of the components of the iDM graph be

materialized beforehand. Whether the components of a resource view are materialized or not is hidden by modelling a resource view as an interface consisting of four get-methods, one for each component:

```
Interface ResourceView {
    getNameComponent(): return η
    getTupleComponent(): return τ
    getContentComponent(): return χ
    getGroupComponent() : return γ
}
```

As a consequence, each resource view hides *how*, *when*, and even *where* the different components are computed. The return values η, τ, χ, and γ in the interface above are themselves implemented as interfaces that provide access to the data encapsulated in each component. This functional style is especially useful to allow the manipulation of components that may be infinite, i.e., the χ- and γ-components (see Section 2.4.4).

As an example of the use of the interface above, in Figure 2.1 the subgraph representing the contents of LaTeX file "vldb 2006.tex" may be transformed to an iDM graph if a user requests that information, i.e., when she calls getGroupComponent() on resource view $V^{\text{file}}_{\text{vldb 2006.tex}}$. In the following sections, we discuss how to obtain the data returned by the four different get*Component-methods.

2.4.2 Extensional Components

One important class of resource-view components are *extensional components*. Extensional components return base facts. No additional query processing is required to retrieve those facts.

A prominent example of base facts is data that is stored on a hard disk drive or in main memory. For instance, the byte content of a Word file is stored on disk. That byte content is considered a base fact. A second example is a DB table that is persisted in one of the segments of a DBMS.

2.4.3 Intensional Components

In contrast to extensional components, *intensional components* require query processing to compute the component. Examples of intensional components include computing the result to a query based on locally available data or based on calling remote hosts.

For instance, a view defined on top of a set of DB tables is considered intensional data. This holds even if that view was explicitly created as a materialized view [81]. In that case that data was simply precomputed (i.e., materialized). However, on the logical iDM level, that data is still considered intensional data as it represents a query result. In our approach, any intensional components of an iDM graph may be materialized in order to speed up query processing or graph navigation. The discussion on whether to materialize a resource view or not is orthogonal to our model.

iDM Use-case: Active XML

A special case of querying a remote host is a call to a web service. For this reason, iDM can also easily represent XML-specific schemes such as ActiveXML [4]. The basic idea of ActiveXML is to enrich XML documents with calls to web services. Whenever a web service is called, the result of that call will be inserted into the XML document. For instance, consider the ActiveXML document:

```
<dep>
  <sc>web.server.com/GetDepartments()</sc>
</dep>
```

The <sc> element contains the web service call. When that web service is executed, its result is inserted into the XML document:

```
<dep>
    <sc>web.server.com/GetDepartments()</sc>
    <deplist>
       <entry>
          <name>Accounting</name>
       </entry>
       ...
    </deplist>
</dep>
```

Note that in ActiveXML the original web service call is not replaced in the document. Rather, the data corresponding to the call is added to the document contents. To instantiate Active XML documents in iDM, it suffices to define a special subclass AXML of resource view class xmlelem with

$$\gamma_i^{\mathsf{AXML}} = \left(\varnothing, \langle V_j^{\mathsf{sc}}[, V_k^{\mathsf{scresult}}] \rangle\right).$$

Here V_j^{sc} contains the call to the web service and $[, V_k^{\mathsf{scresult}}]$ is an optional entry that is only part of the group component if the web service was called.

It is important to note that iDM, in contrast to ActiveXML, also supports graph data and is not restricted to XML documents. Moreover, the publish-subscribe features of Active XML can also be instantiated in iDM. We describe how infinite components can be modeled in iDM in the next section.

2.4.4 Infinite Components

Infinite components may occur in two different components of a resource view. First, the content component χ_i may be infinite (see Definition 2.1). Prominent examples of infinite content are media streams such as audio and video streams, e.g., Real or QuickTime. These streams may be modelled as infinite sequences of symbols delivered by a content component χ_i. Second, the group component γ_i may also contain either an infinite set of resource views, an infinite sequence of resource views, or both (see Definition 2.1). Prominent examples of infinite group components are data streams (see Section 2.3.4), information-filter message notifications [49], and email. In the following, we will discuss why email may be considered infinite.

iDM Use-case: EMail

Consider the example of an email repository. An email repository subscribes to an infinite stream of messages, e.g., all messages containing `jens.dittrich@inf.ethz.ch` in at least one of the TO, CC, or BCC fields. The email server keeps a window on the incoming messages. Users may delete messages from or add messages to that window. Using iDM we have two options to model email:

Option 1: Model the State. We may model the **state** of the INBOX. That state is finite. It can be modeled as

$$\gamma_{\text{i}}^{\text{INBOX State}} = \big(\varnothing, \langle V_{\text{q}_1}^{\text{message}}, \ldots, V_{\text{q}_n}^{\text{message}} \rangle\big)$$

where $V_{\text{q}_1}^{\text{message}}, \ldots, V_{\text{q}_n}^{\text{message}}$ represent the message window currently available in the INBOX, i.e., the current state of the INBOX. Note that for this option the state of that INBOX may be retrieved multiple times.

Option 2: Model the Stream. Alternatively, we may model the **stream** of incoming messages itself bypassing the state window offered by the email server. This stream is modeled as follows:

$$\gamma_{\text{i}}^{\text{INBOX message stream}} = \big(\varnothing, \langle V_{\text{q'}_1}^{\text{message}}, \ldots, V_{\text{q'}_n}^{\text{message}} \rangle_{n\to\infty}\big),$$

Here, $V_{\text{q'}_1}^{\text{message}}, \ldots, V_{\text{q'}_n}^{\text{message}}$ represent all messages routed to address `jens.dittrich@inf.ethz.ch` during the lifetime of that address. As the stream has no state, messages delivered by the stream cannot be retrieved a second time. The API to access the infinite is based on lazy computation, using either pull or push operators [77, 125].

Both options may make sense depending on the application. For instance, Option 1 may be used in those cases in which the user employs multiple email clients to read email. In that case, Option 1 could observe the email repository of a user without removing any emails from the server. In contrast, Option 2 is useful in those cases in which iDM is used as the single point of access to the user's email repository. In that case, emails streamed to the user would be delivered to the client and removed from the server immediately. No other client would be able to see those messages.

In addition, if we are not able to obtain a real data stream, we may convert a state into a pseudo data stream using a generic polling facility. In the iDM graph that pseudo data stream is then represented as an infinite message stream.[5]

Implementing Streams: Need to Push

In order to efficiently support stream processing, any system implementing iDM graphs has to provide push-based protocols [67, 125]. Our current implementation of iDM in `iMeMex` already supports push-based operators. Our push-operators may register for changes on any of the components of a resource

[5]Interestingly, several popular email services such as POP and IMAP servers do not support the second option. The same applies to RSS/ATOM servers. RSS/ATOM is a method useful for publishing a sequence of messages. However, RSS/ATOM is implemented by simply publishing an XML document containing those messages on a web server. Clients do not get any notifications on changes of that document. For this reason, clients have to poll the server for updates regularly.

view. Incoming change events on any resource view, such as a new email message or a new tuple on a data stream, will then be passed to all subscribed push-operators. They will process those events immediately. In that manner, data-driven stream processing in the spirit of specialized data stream management systems (DSMS) [1] is enabled.

2.5 Searching with the iMeMex Query Language

This section provides an overview of the query language used to query iDM in the *iMeMex Personal Dataspace Management System (PDSMS)*. We have developed a simple query language termed *iMeMex Query Language (iQL)*, which we use to evaluate search queries on a resource view graph. Our primary design goal was to have a language that can be used by an *end-user*, i.e., a language that is at the same time *simple and powerful*. For this reason, we decided to design our language as an extension to existing IR keyword search interfaces as used by Google and other search engines. In that manner, users may benefit from our language with only a minimal learning effort. At the same time, advanced users may use the same language and interfaces to execute more advanced queries.

One may argue that XML query languages such as XPath and XQuery could be extended to work on top of iDM. The main reasons we did not do this, however, were that XPath and XQuery are too complex to be offered as an end-user language for personal information management. Neverheless, XPath 2.0 has some interesting features. Some of those we have adopted for our language iQL, such as path expressions and predicates on attributes. In that respect, our language is close in spirit to NEXI [159]. In contrast to NEXI, however, iQL will include features important for a PDSMS, such as support for updates and continuous queries. A full specification of iQL will be detailed in future work. Here, we present only the relevant subset of iQL necessary to pose search queries against the dataspace. We therefore consider only dataspaces that can be modeled by resource views with finite components.

The following definition describes the semantics of query expressions in iQL.

DEFINITION 2.3 (QUERY EXPRESSION) *Assume a finite graph of resource views $G := \{V_1, \ldots, V_n\}$. We require G to be a complete set of resource views, i.e. there must be no resource view $V_k \notin G$ that is at the same time in a group component γ_i of $V_i \in G$. In addition, we assume the content and group components of all resource views in G to be finite. A query expression Q selects a subset of nodes $R := Q(G) \subseteq G$.*

The query expression Q follows the syntax and semantics given in Tables 2.2 and 2.3. □

In order to get an idea of the expressiveness of our language the following list presents some example iQL queries:

`Donald Knuth`: returns all resource views containing the keywords “Donald” and “Knuth” in their content component.

`size > 42000 and lastmodified < yesterday()`: returns those resource views having a tuple component attribute named size greater than 42000 and a lastmodified attribute with a date value before yesterday.

```
QUERY_EXPRESSION := (PATH | KT_PREDICATE) ((UNION|INTERSECT) QUERY_EXPRESSION)*
PATH             := (LOCATION_STEP)+
LOCATION_STEP    := LS_SEP NAME_PREDICATE ('[' KT_PREDICATE ']')?
LS_SEP           := '//' | '/'
NAME_PREDICATE   := '*' | ('*')?  VALUE ('*')?
KT_PREDICATE     := (KEYWORD | TUPLE) (LOGOP KT_PREDICATE)*
KEYWORD          := '"' VALUE (WHITESPACE VALUE)* '"'
                    | VALUE (WHITESPACE KEYWORD)*
TUPLE            := ATTRIBUTE_IDENTIFIER OPERATOR VALUE
OPERATOR         := '=' | '<' | '>' | '∼'
LOGOP            := 'AND' | 'OR'
```

Table 2.2: Syntax of query expressions (Core grammar)

Query Expression	Semantics
//*	$\{V_i \mid V_i \in G\}$
a	$\{V_i \mid V_i \in G \wedge \text{`}a\text{'} \subseteq \chi_i\}$
a b	$\{V_i \mid V_i \in G \wedge \text{`}a\text{'} \subseteq \chi_i \wedge \text{`}b\text{'} \subseteq \chi_i\}$
//A	$\{V_i \mid V_i \in G \wedge \eta_i = \text{`}A\text{'}\}$
//A/B	$\{V_i \mid V_i \in G \wedge \eta_i = \text{`}B\text{'} \wedge (\exists W_j \in G : \eta_j = \text{`}A\text{'} \wedge W_j \rightarrow V_i)\}$
//A//B	$\{V_i \mid V_i \in G \wedge \eta_i = \text{`}B\text{'} \wedge (\exists W_j \in G : \eta_j = \text{`}A\text{'} \wedge W_j \rightsquigarrow V_i)\}$
b=42	$\{V_i \mid V_i \in G \wedge \exists \tau_i = (\langle \ldots, b, \ldots \rangle, \langle \ldots, 42, \ldots \rangle)\}$
b=42 a	:= b=42 ∩ a
//A/B[b=42]	:= //A/B ∩ b=42

Table 2.3: Semantics of query expressions

//Introduction[class="latex_section"]: returns resource views named "Introduction" and of resource-view class "latex_section".

//PIM//Introduction[class="latex_section"]: returns every resource view named "Introduction" of class "latex_section" that is indirectly related to a resource view named "PIM".

//PIM//Introduction[class="latex_section" and "Mike Franklin"]: returns resource views named "Introduction" of class "latex_section" that are indirectly related to a resource view named "PIM". All returned results have to contain the keywords "Mike" and "Franklin" in their content component (solves Example 2.1 from the Introduction of this chapter).

//OLAP//*[class="figure" and "Indexing time"]: first, selects resource views that are indirectly related to a resource view named "OLAP". In addition, all results have to be of resource view class "figure" and have to contain the keywords "Indexing" and "time" in their content component (solves Example 2.2 from the Introduction of this chapter).

Data Source	Total Size (MB)	# of Resource Views						
		Base Views			Derived Views			Total
		Files and Folders	Email	Total	XML	LaTeX	Total	
Filesystem	4,243	14,297	0	14,297	117,298	11,528	128,826	143,123
Email / IMAP	189	0	6,335	6,335	672	350	1,022	7,357
Total	**4,435**	**14,297**	**6,335**	**20,632**	**117,970**	**11,878**	**129,848**	**150,480**

Table 2.4: Characteristics of the personal dataset used in the evaluation

Note that to find indirectly related resource views in a general iDM graph, we must avoid cycles during processing. The simplest approach is to perform depth- or breadth-first search on the iDM graph at query time. If more efficiency is required, however, an interesting alternative is to use a reachability index structure [158].

Our implementation of iQL in `iMeMex` also supports user-defined joins by employing a special join function. See Section 2.6.2 for join examples used in our evaluation.

2.6 Evaluation

The goal of this experimental evaluation is to show that iDM can be efficiently implemented in a real PDSMS. We present results based on an initial implementation of iDM in `iMeMex`. We report indexing times and sizes for an actual personal dataset. Furthermore, we execute different classes of queries over iDM and report query-response times.

2.6.1 Setup

We performed our experiments on an Intel Pentium M 1.8 GHz PC with 1 GB of main memory and an IDE disk of 60 GB. We used MS Windows XP Professional SP2 as its operating system and a volume with NTFS. The `iMeMex` PDSMS is implemented in Java and we used Sun's hotspot JVM 1.4.2_08-b03. The JVM was configured to allocate at most 512 MB of main memory. In order to index resource views, we make use of both Apache Derby 10.1 and Apache Lucene 1.4.3 as libraries. The full-text of all text-based content and PDF content is indexed with Lucene. Furthermore, we have implemented converters that take content components in XML or in LaTeX formats and generate resource-view graphs as shown in Section 2.2.

Many authors have observed that there is a lack of benchmarks to evaluate PIM approaches [99, 163]. Thus, we have decided to use a real personal dataset to evaluate the efficiency of our iDM implementation. Our dataset consists of the personal files and emails of the author of this thesis. All files were kept on the same computer in which `iMeMex` was run; emails were kept on a remote server and were accessed via the IMAP protocol.

The characteristics of the data set used are shown in Table 2.4. We show each data source and the total number of resource views extracted from the data source. The set of all resource views may be broken into two subsets: base views and derived views. Base views are obtained by representing in iDM base items of the data sources (i.e., files and folders for the filesystem; emails, folders and attachments for email). Derived views are extracted from XML content or from LaTeX content. We report the number of resource views in each of these subsets. We also show the total storage requirements of the items stored in the data sources.

As we may observe in Table 2.4, the largest portion of the data is stored locally on the filesystem. Most of the resource views present on the filesystem data source are obtained from the content of XML and LaTeX files. These views were obtained from 47 XML documents and 282 LaTeX documents. In the email data source, the number of resource views extracted from XML or LaTeX documents was relatively smaller, as in this dataset these documents are not commonly exchanged as attachments to email messages. We have 13 XML documents and 7 LaTeX documents as attachments to email messages. When both data sources are taken together, the number of resource views derived from base items greatly surpasses the number of base items.

2.6.2 Results

We report results on two experiments using the dataset described in the previous section. In the first experiment, we show `iMeMex` index and replica creation times and sizes; in the second experiment, we evaluate query performance when exploiting iDM to run a set of queries over structured and unstructured desktop data.

Indexing. In `iMeMex`, we employ a set of index and replica implementations to speed up query processing. In the evaluation, we have used the following structures:

1. **Name Index and Replica**: an Apache Lucene full-text index that also stores the values of the resource views' name components.
2. **Tuple Index and Replica**: a replica of all resource views' tuple components. The system keeps this replica in memory and an auxiliary sorted index structure is also kept in memory. This index structure is based on vertical partitioning [40].
3. **Content Index**: an Apache Lucene full-text index on text extracted from content components, if possible. This index is not a replica, i.e., the original content is not saved in the index.
4. **Group Replica**: a replica of all resource views' group components. The system keeps this replica in memory.

For the remainder, we refer to indexes and replicas simply by the term *indexes*, as their use in this evaluation is to improve query processing time. In our initial implementation, all resource views present in the data source were registered in a **Resource-View Catalog** and their components were indexed in the structures described above. The Resource-View Catalog was used to provide unique identifiers for each resource view.

Data Source	Net Input Data Size (MB)	Index Sizes (MB)					
		Name	Tuple	Content	Group	RV Catalog	Total
Filesystem	212.3	12.5	11.5	113.0	3.3	24.4	**164.7**
Email / IMAP	43.1	0.4	1.8	5.0	0.2	0.4	**7.8**
Total	**255.4**	**12.9**	**13.3**	**118.0**	**3.5**	**24.8**	**172.5**

Table 2.5: Index sizes for the personal dataset

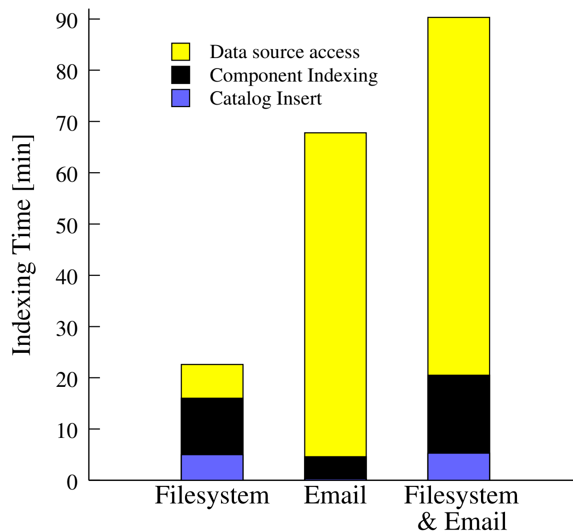


Figure 2.4: Indexing times [min]

Table 2.5 shows the resulting *index sizes* for the personal dataset. Although the original dataset occupies around 4 GB, we observe that the net input data size is of about 255 MB. The net input data size is obtained by excluding the content size of those resource views whose content could not be converted to a textual representation (e.g., image formats). That content was therefore not given as input to the content index. As we may notice on Table 2.5, the total size for all indexes is 67.5% of the net input data size. Most of that total index size is taken by the full-text index on content.

The total *indexing time* for the personal dataset described in Section 2.6.1 is shown in Figure 2.4. For each data source, we report the time necessary to register metadata in the Resource-View Catalog (Catalog Insert), the time spent inserting data in our index structures (Component Indexing), and the time spent to obtain the data from the underlying data sources (Data Source Access). For the filesystem, the total indexing time is about 22 min. Roughly half of that time is spent on inserting information on index data structures, while the remaing time is distributed among catalog maintenance and the actual scanning of the underlying filesystem. The indexing of email takes about 68 min and the time is dominated by data source access. The catalog-maintenance time during email indexing is negligible, as the email data source contains only a small number of resource views. Overall, the influence of indexing the remote email data source on the total indexing time is significant, as most of the time is spent with data source access.

Query Processing. We focus our evaluation on response times observed for a set of queries over the desktop dataset used. The queries evaluated are shown in Table 2.6, along with their number of results.

	iQL Query expression	# of Results
Q1	`database`	941
Q2	`database tuning`	39
Q3	`size > 420000 and lastmodified < 12.06.2005`	88
Q4	`//papers//*Vision/*["Franklin"]`	2
Q5	`//VLDB200?//?onclusion*/*["systems"]`	2
Q6	`//VLDB2005//*["documents"] ∪ //VLDB2006//*["documents"]`	31
Q7	`join(//VLDB2006//*[class="texref"] as A, //VLDB2006//*[class="environment"]//figure* as B, A.name=B.tuple.label)`	21
Q8	`join(//*[class = "emailmessage"]//*.tex as A, //papers//*.tex as B, A.name = B.name)`	16

Table 2.6: iQL queries used in the evaluation

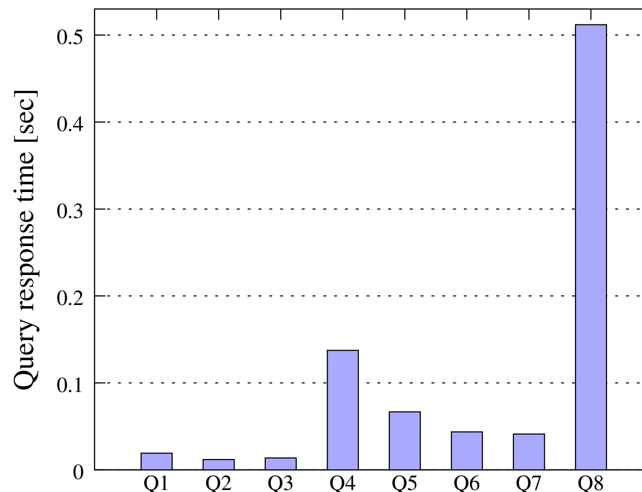


Figure 2.5: Query-response times for queries Q1–Q8 [sec]

Figure 2.5 shows the observed response times for the eight queries evaluated. We report all execution times with a warm cache, i.e., each query is executed several times until the deviation on the average execution time becomes small. As we may notice, most queries evaluated (Q1-Q7) are executed in less than 0.2 seconds. The only exception is Q8, a join query that includes information from different subsystems (email, filesystem) and takes about 0.5 seconds. The reason for this higher execution time is that after fetching the data via index accesses, our query processor obtains directly and indirectly related resource views by forward expansion, i.e., we navigate the iDM graph to find directly and indirectly related resource views by processing elements in the forward direction of group component connections. For Q8, that causes the processing of a large number of intermediate results when compared to the final result size. In order to provide even better response times in such situations, we plan to investigate alternative processing strategies such as backward or bidirectional expansion [100]. Nevertheless, in the HCI community it is argued that a response time of less than 1 second should be the goal for every computer

system [148]. Thus, we conclude that for a sample of meaningful queries over real desktop data the implementation of iDM in `iMeMex` allows for query processing with interactive response times.

2.7 Related Work

Personal Information Management is an area broad in scope and we review below several contributions related to our work. We divide them into four categories: data models, operating systems, structure extraction, and PIM systems. For each of these categories, we discuss how iDM distinguishes itself from earlier approaches.

Data Models. The relational model has been proposed as a means to provide universal access to data [36]. In the context of information integration, several works have used the relational model to provide a global, data source independent view of data [10, 109]. The Information Manifold [109], for example, is based on the idea of a global schema on top of which the data sources may be expressed as relational views. Object-oriented systems [30] augment relational approaches to represent complex data types. Shoens et al. [149] propose the Rufus system, which provides an object-based data model for desktop data. The system imports desktop data into a set of pre-defined classes and provides querying capabilities over those classes. All of these previous approaches are based on a schema-first modeling strategy that is not well suited for personal information. In contrast, our data model is in line with the goals recently presented in the dataspace vision [65], which claims that future information systems must be capable of offering a wide range of modeling strategies, such as *schema-later* or even *schema-never*.

Semi-structured approaches [131, 170] have been proposed as an alternative to self-describe data and thus eliminate the need to provide pre-defined schemas. Currently, the most prominent semi-structured model is XML [170]. XML is typically associated with a specific serialization for data exchange, which means that it becomes harder to represent data in its data model from a purely logical standpoint, as advocated by Abiteboul [2]. Furthermore, XML is not capable of representing graph, intensional, and infinite data without using add-on techniques. However, the former types of data are key in personal information management and therefore fully supported by iDM. Some approaches have tried to extend XML in order to cope with these limitations. Colorful XML [97] proposes a data model in which an XML document is extended with multiple hierarchies, distinguished by node coloring. ActiveXML [4, 119] represents intensional data in XML documents by embedding web service calls. In sharp contrast, iDM offers a logical model that represents a wide range of available heterogeneous personal information and integrates all of the above approaches into one generic framework. For instance, in Section 2.4.3 we show how ActiveXML may be modeled as a use case of iDM. Some recent works have developed keyword-search techniques considering graph data models to represent heterogeneous data [78, 100]. The search techniques developed are orthogonal to our approach and may also be used on top of iDM.

Operating Systems. Recently, modern operating systems have been amended to provide full-text search capabilities. Examples of such systems are Google Desktop [71], MS Desktop Search [162], and Apple Spotlight [152]. These systems use simple data models based on sequences of words. Unlike in iDM, the structural information available within files is not exploited for queries.

The Linux add-on file system Reiser FS [139] blurs the distinction among files and folders by letting files behave as folders in special circumstances, e.g. by displaying file attributes as files. The Windows Explorer file manager in WindowsXP [167] allows for example archive contents to be navigated, also blurring the distinction among files and folders in some special cases. Microsoft WinFS [165], now discontinued,[6] proposed to base storage of personal information in a relational database. Data was represented in WinFS in an item data model, which is a subset of the object-oriented data model. Like in previous object-oriented approaches for personal information [149], WinFS employed a schema-first data modeling strategy. This problem was addressed in WinFS in a brute-force fashion, i.e., by shipping the system with a wide array of pre-defined schemas for personal information that may then be extended by application developers. In sharp contrast to that, in iDM there is no need to import generally available data as a variety of data models may already be expressed as constrained versions of iDM (see Section 2.3). That allows a system based on iDM to be useful from the start — with investment on schemas being made only if necessary [65]. In addition, from a systems perspective, WinFS represented a platform-specific solution restricted to a vendor-specific set of devices. Our approach is totally different from that as the iDM-based `iMeMex` PDSMS already works with a larger set of operating systems including Linux and other Unixes, Windows, and Mac OS X (see Chapter 5). At the same time, it is possible to make `iMeMex` seamlessly integrate into all of the operating systems above [51].

Structure Extraction. Approaches to the problem of structure extraction are *automatic or semi-automatic*. Some techniques for *automatic* structure extraction have a pre-defined schema to which entity references are matched, e.g. reference reconciliation [56] and email classification [38, 93], while others seek to detect categories on the data, e.g. topic and social network analysis [41, 117]. These techniques are orthogonal to our approach, as they provide means to extract structure from data; derived information may be represented in a variety of data models, including iDM.

Fully automatic structure extraction is a long-term goal. Thus, *semi-automatic* extraction approaches [87] try to increase accuracy by incorporating user interference. Hubble [110] is a system that allows users to organize and categorize XML files into dynamically generated folders using complex XQuery expressions. In contrast to iDM, Hubble is restricted to extensional XML files and does not suppport infinite data. Moreover, iDM provides a logical data model to unify the XML file's elements hierarchy and the outside folder hierachy.

PIM Systems. Systems such as SEMEX [53] and Haystack [102] enable users to *browse by association*. The information in the desktop is extracted and imported into a repository that encodes pre-defined schemas about high-level entities such as persons, projects, publications, etc. The rich relationships among such personal data items call for a data model that may represent arbitrary graph data and not only trees or DAGs. Lifestreams [68] completely abandons the files-and-folders paradigm and bases the organization of information on a timeline. MyLifeBits [15] allows visualizations to be built on top of a unified store for personal information. All information, including resource content, is stored in a relational database. Queries are represented in these systems as collections whose content is calculated on

[6]The downloadable beta as well as all other preliminary information about WinFS were recently removed from its website [165].

demand. Such use-cases of intensionally defined data may also be modeled in iDM (see Section 2.4.3).

2.8 Conclusions

Personal Information Management has become a key necessity of almost everybody. Several techniques and systems have been proposed for various scenarios and operating systems. Considerable progress has been made in the PIM area in the recent past. At the same time, it has become clear that what is missing is a unified approach to PIM that is able to represent *all* personal information in a single, powerful, and yet simple data model. This chapter has proposed the iMeMex Data Model (iDM) as a unified and versatile solution. The major advantages of our approach are: (1) iDM clearly differentiates between the logical data model and its physical representation, (2) iDM is powerful enough to represent XML, relations, files, folders, and cyclic graphs in a single data model, (3) iDM is able to represent the structural contents inside files as part of the same data model, (4) iDM is powerful enough to represent extensional data (base facts), intensional data (e.g. ActiveXML), as well as infinite data (content and data streams), (5) iDM enables a new class of queries that are not available with state-of-the-art PIM tools.

iDM builds the foundation of the *iMeMex Personal Dataspace Management System* (see Chapter 5). We have demonstrated that iDM can be efficiently implemented in such type of system offering both quick indexing times and interactive query-response times.

We point out below some issues relevant to personal dataspace management that are orthogonal to our model, but which are easier to tackle once a data model like iDM is in place.

1. **Versioning.** A PDSMS keeps track of all changes made to the dataspace. As with classical versioning techniques, logically, each change creates a new version of the whole dataspace. With iDM, we represent the entire dataspace of a user in one model. Thus, the implementation of versioning is strongly simplified.
2. **Lineage.** Data lineage refers to keeping the history of all data transformations that originated a given resource view. For example, when a user copies a file into another and then modifies the new file, the system should keep provenance information for the new resource view. With a unified model such as iDM, it is possible to keep lineage information across data sources and formats.

The exploration of these issues is part of ongoing and future work in the iMeMex Personal Dataspace Management System. Future work also includes developing a full specification of iQL and exploring PIM applications such as reference reconciliation and clustering on top of the iMeMex platform.

Chapter 3

iTrails: Pay-as-you-go Information Integration in Dataspaces

Dataspace management has been recently identified as a new agenda for information management [65, 88] and information integration [89]. In sharp contrast to standard information integration architectures, a dataspace management system is a *data-coexistence approach*: It does not require *any* investments in semantic integration before querying services on the data are provided. Rather, a dataspace can be gradually enhanced over time by defining relationships among the data. Defining those integration semantics gradually is termed *pay-as-you-go* information integration [65], as time and effort (pay) are needed over time (go) to provide integration semantics. The benefits are better query results (gain). This chapter explores how to perform pay-as-you-go information integration in dataspaces. We provide a technique for declarative pay-as-you-go information integration named iTrails. The core idea of our approach is to declaratively add lightweight "hints" (trails) to a search engine thus allowing gradual enrichment of loosely integrated data sources. Our experiments confirm that iTrails can be implemented efficiently, introducing only little overhead during query execution. At the same time, iTrails strongly improves the quality of query results. Furthermore, we present rewriting and pruning techniques that allow us to scale iTrails to tens of thousands of trail definitions with minimal growth in the rewritten query size.

3.1 Introduction

Over the last ten years information integration has received considerable attention in research and industry. It plays an important role in mediator architectures [89, 109, 124, 131], data warehousing [28], enterprise information integration [86], and also P2P systems [127, 154]. There exist two opposite approaches to querying a set of heterogeneous data sources: *schema first* and *no schema*.

Schema first: The schema-first approach (SFA) requires a semantically integrated view over a set of data sources. Queries formulated over that view have clearly defined semantics and precise answers. SFA is implemented by mediator architectures and data warehouses. To create a semantically integrated view,

the implementor of the SFA must create precise mappings between every source's schema and the SFA's mediated schema. Therefore, although data sources may be added in a stepwise fashion, integrating *all* the desired data sources is a cumbersome and costly process. □

Due to the high integration cost of sources, in typical SFA deployments, only a small set of the data sources is available for querying when the system first becomes operational. This drawback causes users, developers, and administrators to choose *no schema* solutions, e.g., desktop, enterprise, or Web search engines, in many real-world scenarios.

No schema: The no-schema approach (NSA) does not require a semantically integrated view over the data sources. It considers all data from all data sources right from the start and provides basic keyword and structure queries over all the information on those sources. NSA is implemented by search engines (e.g., Google, Beagle, XML-IR engines [37, 157]). The query model is either based on a simple *bag of words model*, allowing the user to pose keyword queries, or on an *XML data model*, allowing the user to pose structural search constraints, e.g., using a NEXI-like query language (Narrowed eXtended XPath[1] [159]). □

Unfortunately, NSAs do not perform *any* information integration. Furthermore, query semantics in NSAs are imprecise, as schema information is either non-existent or only partially available. Thus, the quality of search answers produced depends heavily on the quality of the ranking scheme employed. Although effective ranking schemes are known for the Web (e.g. PageRank [25]), there is still significant work to be done to devise good ranking methods for a wider range of information integration scenarios.

In this chapter, we explore a new point in the design space inbetween the two extremes represented by SFA and NSA:

Trails: The core idea of our approach is to start with an NSA but declaratively add lightweight "hints", termed *trails*, to the NSA. The trails add integration semantics to a search engine and, therefore, allow us to gradually approach SFA in a "pay-as-you-go" fashion. We present a declarative solution for performing such pay-as-you-go information integration named iTrails. The work presented in this chapter is an important step towards realizing the vision of the new kind of information integration architecture called *dataspace management systems* [65]. □

3.1.1 Motivating Example

Figure 3.1 shows the dataspace scenario used throughout this chapter as a running example. The example consists of four data sources: (a) an email server containing emails and PDF attachments, (b) a network file server containing files and folders, (c) a DBMS containing relational data but also PDFs as blobs, and (d) a laptop containing files and folders. We assume that all data is represented using the graph model introduced in Chapter 2. For instance, in Figure 3.1(a), each structural element is represented by a separate node: The email account of user 'mike' is represented by Node 1. This node has a directed

[1]NEXI is a simplified version of XPath that also allows users to specify keyword searches. All of those queries could also be expressed using XPath 2.0 plus full-text extensions.

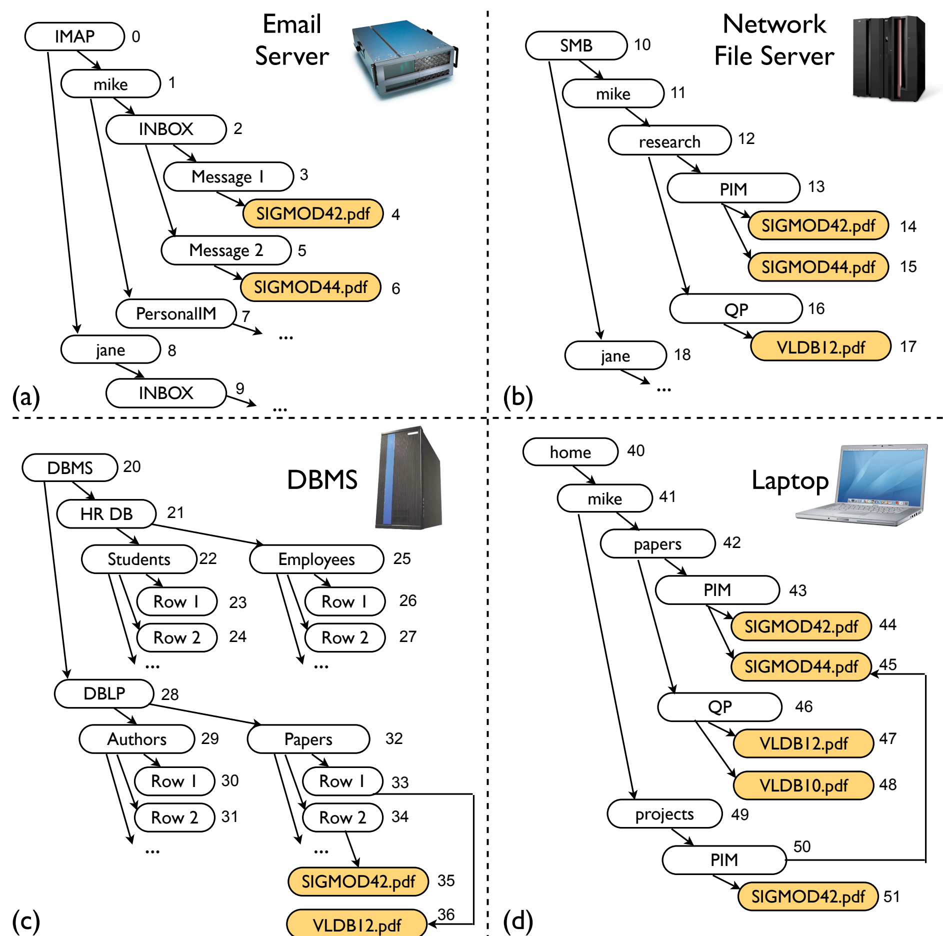


Figure 3.1: Running Example: a Dataspace consisting of four heterogeneous data sources. The data provided by the data sources is represented using the graph model of Chapter 2. The four components of the graph are disconnected.

edge to Mike's 'INBOX', which is represented by Node 2. Node 2 has directed edges to all emails, which again are represented by separate nodes. Also in Figures 3.1(b,c,d), files, folders, databases, tables, and rows are represented by separate nodes. Again as illustrated in Chapter 2, it is important to understand that we do *not* require that the data be *stored* or *materialized* as a graph. We just *represent* the original data *logically* using a graph model.

In the following, we present two example scenarios on how to query the data presented in Figure 3.1.

EXAMPLE 3.1 (PDF YESTERDAY) "Retrieve all PDF documents that were added or modified yesterday."

State-of-the-art: Send a query to the search engine returning all documents matching `*.pdf`. Depending on the data source the constraint 'yesterday' has to be evaluated differently: First, for the email server, select all PDFs that are attachments to emails that have an attribute `received` set to yesterday; second, for the DBMS, select the PDFs that are pointed to by rows that were added or changed yesterday; third,

for the network file server and the laptop, select the PDFs that have an attribute `lastmodified` set to yesterday. In any case, the user has to specify a complex query considering all of the schema knowledge above.

Our goal: To provide a method that allows us to specify the same query by simply typing the keywords `pdf yesterday`. To achieve this goal, our system exploits "hints", in this chapter called *trails*, that provide partial schema knowledge over the integrated information. Consider the system has the following hints:

1. The `date` attribute is mapped to the `modified` attribute;
2. The `date` attribute is mapped to the `received` attribute;
3. The `yesterday` keyword is mapped to a query for values of the `date` attribute equal to the date of yesterday;
4. The `pdf` keyword is mapped to a query for elements whose names end in `pdf`.

As we will see, our method allows us to specify the hints above gradually and to exploit them to rewrite the keyword search into a structural query that is aware of the partial schema information provided by the hints. □

EXAMPLE 3.2 (MULTIPLE HIERARCHIES) "Retrieve all information about the current work on project PIM."

State-of-the-art: User Mike has some knowledge of the hierarchy (schema) he has used to organize his data on his laptop. Thus, Mike would send the following path query to the search engine: `//projects/PIM`. However, that query would retrieve only part of the information about project PIM. In order to retrieve all the information, Mike must manually send other queries to the search engine that are aware of the hierarchies (schemas) used to store the data in all the sources. On the network file server, that would amount to sending another query for `//mike/research/PIM`; on the email server, `//mike/PersonalIM`. Note that not all folders containing the keyword PIM are relevant. For example, the path `//papers/PIM` in Mike's laptop might refer to already published (not current) work in the PIM project. In summary, Mike must be aware of the schemas (hierarchies) used to store the information in all the sources and must perform the integration manually.

Our goal: To provide a method for specifying the same query by simply typing the original path expression `//projects/PIM`. To achieve this goal, we could gradually provide the following hints:

1. Queries for the path `//projects/PIM` should also consider the path `//mike/research/PIM`;
2. Queries for the path `//projects/PIM` should also consider the path `//mike/PersonalIM`.

As we will see, our method allows us to exploit these hints to rewrite the original path expression to also include the additional schema knowledge encoded in the other two path expressions. □

3.1.2 Contributions

In summary, this chapter makes the following contributions:

1. We provide a powerful and generic technique named iTrails for pay-as-you-go information integration in dataspaces. We present how to gradually model semantic relationships in a dataspace through *trails*. Trails include traditional semantic attribute mappings as well as traditional keyword expansions as special cases. However, at no point does a complete global schema for the dataspace need to be specified.
2. We provide query processing strategies that exploit trails. These strategies comprise three important phases: *matching*, *transformation*, and *merging*. We explore the complexity of our query rewriting techniques and refine them by presenting several possible trail-rewrite pruning strategies. Some of these pruning strategies exploit the fact that trails are uncertain in order to tame the complexity of the rewrite process while at the same time maintaining acceptable quality for the rewrite. Furthermore, as trail-expanded query plans may become large, we discuss a materialization strategy that exploits the knowledge encoded in the trails to accelerate query-response time.
3. We perform an experimental evaluation of iTrails considering dataspaces containing highly heterogeneous data sources. All experiments are performed on top of our `iMeMex` Dataspace Management System (see Chapter 5). Our experiments confirm that iTrails can be efficiently implemented introducing only little overhead during query execution but at the same time strongly improving the quality of query results. Furthermore, using synthetic data we show that our rewriting and pruning techniques allow us to scale iTrails to tens of thousands of trail definitions with minimal growth in the rewritten query size.

This chapter is structured as follows. Section 3.2 reviews the data and query model as well as the query algebra used throughout this chapter. We will consider a simplified subset of iDM and build our technique on top of it. Section 3.3 presents the iTrails technique and discusses how to define trails. Section 3.4 presents iTrails query processing algorithms and a complexity analysis. Section 3.5 presents trail-rewrite pruning techniques. Section 3.6 presents the experimental evaluation of our approach. Section 3.7 reviews related work and its relationship to iTrails. Finally, Section 3.8 concludes this chapter.

3.2 Data and Query Model

3.2.1 Data Model

The data model used in this chapter is a simplified version of the generic iMeMex Data Model (iDM) introduced in Chapter 2. As illustrated in the Introduction of this chapter and in Figure 3.1, iDM represents every structural component of the input data as a node. Note that, using iDM, all techniques presented in this chapter can easily be adapted to relational and XML data. In order to simplify notation, in this

chapter, we refer to resource-view components by their plain names (e.g. V_i.`group`) instead of by their formal names (e.g., group component γ_i of V_i).

DEFINITION 3.1 (FINITE RESOURCE-VIEW GRAPH) *A* finite resource-view graph $G := \{V_1,..,V_n\}$ *is a finite and complete set of resource views. We use the same meaning of a complete set of resource views introduced in Definition 2.3 (Section 2.5): there must be no resource view $V_k \notin G$ that is at the same time in a group component V_i.`group` of $V_i \in G$. The content and group components of each resource view $V_i \in G$ must in addition follow the restricted definitions given below:*

Component of V_i	**Definition**
V_i.`content`	Finite byte sequence of content (e.g. text)
V_i.`group`	Finite sequence of resource views $\langle V_1', \ldots, V_k' \rangle$ related to V_i

Table 3.1: Restrictions on content and group components of a resource view V_i. □

The definition above constrains iDM to finite components. This restriction is important as in this chapter we will not be dealing with continuous queries, but only with the search subset of iQL introduced in Section 2.5. The specification of continuous queries in iQL is a subject of future work.

3.2.2 Query Model

We refer the reader to Section 2.5 for the core grammar and semantics of query expressions written in iQL. The search subset of iQL that we define in Section 2.5 is already targeted only at finite dataspace graphs such as the ones described in the previous section.

In this section, we introduce a new concept that will be important for this chapter: the notion of component projections. A component projection specializes a query expression by refering to a subset of the resource-view components of the resource views returned by the query expression. We formalize a component projection below.

DEFINITION 3.2 (COMPONENT PROJECTION) *A* component projection *C obtains a projection on the set of resource views selected by a query expression Q, i.e., if $R := Q(G)$, then*

$$R.C := \{V_i.C \mid V_i \in Q(G)\},\ \ C \in \{\texttt{name}, \texttt{tuple.<att}_i\texttt{>}, \texttt{content}\}.$$

The set R.C is composed of the distinct occurrences of name and content components or tuple attribute values that appear on the resource views in R. □

For instance, in Figure 3.1(c), if a query expression selects $R = \{33\}$, then R.`name` is the unit set of the string “Row 1”.

Operator	Name	Semantics
G	All resource views	$\{V \mid V \in G\}$
$\sigma_P(I)$	Selection	$\{V \mid V \in I \wedge P(V)\}$
$\mu(I)$	Shallow unnest	$\{W \mid V \rightarrow W \wedge V \in I\}$
$\omega(I)$	Deep unnest	$\{W \mid V \rightsquigarrow W \wedge V \in I\}$
$I_1 \cap I_2$	Intersect	$\{V \mid V \in I_1 \wedge V \in I_2\}$
$I_1 \cup I_2$	Union	$\{V \mid V \in I_1 \vee V \in I_2\}$

Table 3.2: Logical algebra for query expressions.

3.2.3 Query Algebra

Query expressions are translated into a logical algebra with the operators briefly described in Table 3.2. As the query language we use is close in spirit to NEXI [159] (and therefore, XPath), our algebra resembles a path-expression algebra [24]. In contrast to a path-expression algebra [24], however, our algebra operates not only on XML documents but on a general graph model that represents a heterogeneous and distributed dataspace (as shown in Chapter 2). We will use the algebraic representation to discuss the impact of trails on query processing. We represent every query by a *canonical form*.

DEFINITION 3.3 (CANONICAL FORM) *The canonical form $\Gamma(Q)$ of a query expression Q is obtained by decomposing Q into location step separators (LS_SEP) and predicates (P) according to the grammar in Table 2.2 in Section 2.5. We construct $\Gamma(Q)$ by the following recursion:*

$$tree := \begin{cases} G & \textit{if tree is empty} \\ \omega(tree) & \textit{if } LS_SEP = // \wedge \textit{ not first location step} \\ \mu(tree) & \textit{if } LS_SEP = / \wedge \textit{ not first location step} \\ tree \cap \sigma_P(G) & \textit{otherwise.} \end{cases}$$

Finally, $\Gamma(Q) := tree$ is returned. □

For instance, the canonical form of `//home/projects//*["Mike"]` is displayed in Figure 3.2. Each location step (`//home`, `/projects`, and `//*["Mike"]`, resp.) is connected to the next by shallow or deep unnests. The predicates of each location step are represented by separate subtrees.

3.3 iTrails

In this section, we present our iTrails technique. We first present a formal definition of trails. Then we present several use cases (Section 3.3.2). After that, we discuss how to obtain trails (Section 3.3.3). Finally, we show how to extend trails to probabilistic trails (Section 3.3.4) and scored trails (Section 3.3.5).

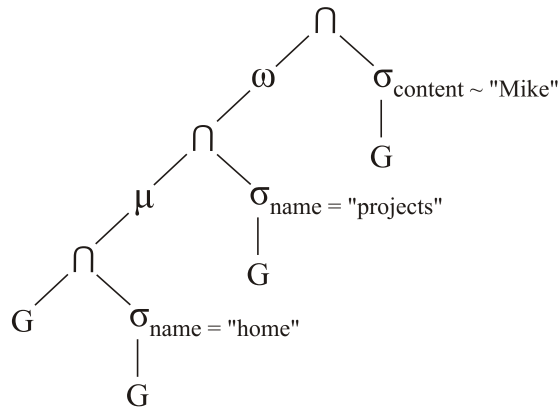


Figure 3.2: Canonical form of `//home/projects//*["Mike"]`.

3.3.1 Basic Form of a Trail

In the following, we formally define *trails*.

DEFINITION 3.4 (TRAIL) *A* unidirectional trail *is denoted either as*

$$\psi := Q_L \longrightarrow Q_R, \text{ or}$$
$$\psi := Q_L.C_L \longrightarrow Q_R.C_R.$$

The first form of a unidirectional trail means that the query on the left Q_L induces the query on the right Q_R. Likewise, the second form means that the query Q_L and component projection C_L induce the query Q_R and component projection C_R. By inducing, we mean that whenever we query for Q_L (or $Q_L.C_L$) we should also query for Q_R (or $Q_R.C_R$). A bidirectional trail *is denoted either as*

$$\psi := Q_L \longleftrightarrow Q_R, \text{ or}$$
$$\psi := Q_L.C_L \longleftrightarrow Q_R.C_R.$$

The latter also means that the query on the right Q_R (respectively, query and component projection $Q_R.C_R$) induces the query on the left Q_L (respectively, query and component projection $Q_L.C_L$). □

Before we formally define the effect of a trail on a query Q (see Section 3.4), we present some trail use cases.

3.3.2 Trail Use Cases

USE CASE 3.1 (FUNCTIONAL EQUIVALENCE) Assume the following trail definitions:

$$\psi_1 := \texttt{//*.tuple.date} \longrightarrow \texttt{//*.tuple.modified}$$
$$\psi_2 := \texttt{//*.tuple.date} \longrightarrow \texttt{//*.tuple.received}$$
$$\psi_3 := \texttt{yesterday} \longrightarrow \texttt{date=yesterday()}$$
$$\psi_4 := \texttt{pdf} \longrightarrow \texttt{//*.pdf}$$

The first two trails establish a relationship between a query for any resource view with a component projection on the `date` tuple attribute and queries for any resource view with component projections on the `modified` or the `received` tuple attributes. The third trail states that when we query for the keyword `yesterday`, we should also consider the results of the query `date=yesterday()`. Finally, the fourth trail states that a keyword query for `pdf` induces a query for all resource views with names matching `*.pdf`, i.e., resource views with names ending in `pdf`.

Now, whenever there is a keyword search for `pdf yesterday`, that keyword search should include the results of the original query `pdf yesterday` but also the results of the structural query:

```
//*.pdf[modified=yesterday() OR received=yesterday()].
```

So instead of making a complex query simpler as done in query relaxation [8], we do the opposite: We extend the original query to a more complex one. This use case of trails addresses Example 3.1. Note that in contrast to Schema-Free XQuery [111], we do not depend on heuristics to detect meaningfully related substructures. Instead, we exploit the hints, i.e., trails, that were defined by the user or administrator in advance. □

USE CASE 3.2 (TYPE RESTRICTION) Assume a trail definition

$$\psi_5 := \texttt{email} \longrightarrow \texttt{class=email}.$$

The query expression `class=email` selects all resource views of type ‘email’ in the dataspace. Then, a simple keyword query for `email` should be rewritten to union its results with all emails contained in the dataspace no matter whether they contain the keyword ‘email’ or not. So, in contrast to P2P integration mappings (such as the ones in Piazza [154]), which define equivalences only between data sources, trails define equivalences between any two sets of elements in the dataspace. Thus, our method can model equivalences both between data sources and between arbitrary subsets of the dataspace. □

USE CASE 3.3 (SEMANTIC SEARCH) Semantic search is just one special case of iTrails. Semantic search includes query expansions using dictionaries, e.g., for *language agnostic search* [12], using *thesauri* such as Wordnet [168], or *synonyms* [135]. The benefit of our technique is that it is not restricted to a specific use case. For example, a trail `car`⟶`auto` could be automatically generated from Wordnet [168].

The same could be done for the other types of dictionaries mentioned above. In contrast to the previous search-oriented approaches, however, iTrails also provides *information-integration semantics*. □

USE CASE 3.4 (HIDDEN-WEB DATABASES) If the dataspace includes mediated data sources, e.g., hidden-web databases, then trails may be used to integrate these sources. Consider the trail:

$$\psi_6 := \texttt{train home} \longrightarrow \texttt{//trainCompany//*[origin="Office str." AND dest="Home str."]}$$

The query on the right side of the trail definition returns a resource view whose content is the itinerary given by the trainCompany web source. Thus, this trail transforms a simple keyword query into a query to the mediated data source. □

3.3.3 Where Do Trails Come From?

We argue that an initial set of trails should be shipped with the initial configuration of a system. Then, a user (or a company) may extend the trail set and tailor it to her specific needs in a pay-as-you-go fashion.

We see four principal ways of obtaining a set of trail definitions: (1) Define trails using a drag-and-drop front end, (2) Create trails based on user-feedback in the spirit of relevance feedback in search engines [145], (3) Mine trails (semi-) automatically from content, (4) Obtain trail definitions from collections offered by third parties or on shared web platforms. As it is easy to envision how options (1) and (2) could be achieved, we discuss some examples of options (3) and (4). A first example of option (3) is to exploit available ontologies and thesauri such as Wordnet [168] to extract equivalences among keyword queries (see Use Case 3.3). Machine learning techniques can also be used to create keyword-to-keyword trails, as has been shown by initial work from Schmidt [146]. Furthermore, another example of (3) would be to leverage information extraction techniques [52, 80] to create trails between keyword queries and their schema-aware counterparts. In addition, trails that establish correspondences among attributes can be obtained by leveraging methods from automatic schema matching [136]. As an example of (4), sites in the style of bookmark-sharing sites such as `del.icio.us` could be used to share trails. In fact, a bookmark can be regarded as a special case of a trail, in which a set of keywords induce a specific resource in the dataspace. Sites for trail sharing could include specialized categories of trails, such as trails related to personal information sources (e.g., email or blogs), trails on web sources (e.g., translating location names to maps of these locations), or even trails about a given scientific domain (e.g., gene data). All of these aspects are beyond the scope of this chapter and are interesting avenues for future work.

However, the trail-creation process leads to two additional research challenges that have to be solved to provide a meaningful and scalable trail rewrite technique.

1. **Trail Quality.** Depending on how trails are obtained, we argue that it will make sense to collect quality information for each trail: We have to model whether a trail definition is "correct". In general,

a hand-crafted trail will have much higher quality than a trail that was automatically mined by an algorithm.

2. **Trail Scoring.** Some trails may be more important than others and therefore should have a higher impact on the scores of query results: We have to model the "relevance" of a trail. For instance, trail ψ_5 from Use Case 3.2 should boost the scores of query results obtained by `class=email` whereas a mined trail `car`$\longrightarrow$`auto` should not favor results obtained from `auto`.

In order to support the latter challenges we extend our notion of a trail by the definitions in the following subsections.

3.3.4 Probabilistic Trails

To model trail quality we relax Definition 3.4 by assigning a probability value p to each trail definition.

DEFINITION 3.5 (PROBABILISTIC TRAIL) *A* probabilisitc trail *assigns a probability value* $0 \leq p \leq 1$ *to a trail definition:*

$$\psi := Q_L \underset{p}{\longrightarrow} Q_R, \text{ or respectively}$$

$$\psi := Q_L.C_L \underset{p}{\longrightarrow} Q_R.C_R.$$

□

The intuition behind this definition is that the probability p reflects the likelihood that results obtained by trail ψ_i are correct.

Obtaining Probabilities. Note that the problem of obtaining trail probabilities is analogous to the problem of obtaining *data* probabilities [107] in probabilistic databases [16]. Therefore, we believe that techniques from this domain could be adapted to obtain trail probabilities. Furthermore, methods from automatic schema matching [136] could also be leveraged for that purpose. In Section 3.5 we will show how probabilistic trail definitions help us accelerate the trail rewrite process.

3.3.5 Scored Trails

A further extension is to assign a scoring factor for each trail:

DEFINITION 3.6 (SCORED TRAIL) *A* scored trail *assigns a scoring factor* $sf \geq 1$ *to a trail definition:*

$$\psi := Q_L \underset{sf}{\longrightarrow} Q_R, \text{ or respectively}$$

$$\psi := Q_L.C_L \underset{sf}{\longrightarrow} Q_R.C_R.$$

□

The intuition behind this definition is that the scoring factor sf reflects the *relevance* of the trail. The semantics of scoring factors are that the additional query results obtained by applying a trail should be

scored sf times higher than the results obtained without applying this trail. As a consequence, scoring factors must be always larger than 1, given that lower scoring factors would penalize the query results obtained by the trail. The effect of scored trails is that they change the original scores of results as obtained by the underlying *data* scoring model.

Obtaining Scoring Factors. If no scoring factor is available, $sf = 1$ is assumed. Scoring factors may be obtained in the same way as trail probabilities (see Section 3.3.4). In addition, when trails are given by a user, a front-end could intuitively support the user in controlling the scoring factor she wishes to give to a trail, e.g., by having a slider that determines how important the trail's right side is.

3.4 iTrails Query Processing

This section discusses how to process queries in the presence of trails. After that, Section 3.5 extends these techniques to allow pruning of trail applications.

3.4.1 Overview

Query processing with iTrails consists of three major phases: *matching*, *transformation*, and *merging*.

1. **Matching.** Matching refers to detecting whether a trail should be applied to a given query. The matching of a unidirectional trail is performed on the left side of the trail. For bidirectional trails, both sides act as a matching side. In the latter case, we consider all three iTrails processing steps successively for each side of the trail. In the following, we focus on unidirectional trails. We denote the left and right sides of a trail ψ by ψ^L and ψ^R, respectively. The corresponding query and component projections are denoted by $\psi^{L.Q}$, $\psi^{L.C}$, and $\psi^{R.Q}$, $\psi^{R.C}$, respectively. If a query Q matches a trail ψ, we denote the matching portion of Q as Q^M_ψ.
2. **Transformation.** If the left side of a trail ψ was matched by a query Q, the right side of that trail will be used to compute the transformation of Q, denoted Q^T_ψ .
3. **Merging.** Merging refers to composing the transformation Q^T_ψ with the original query Q into a new query. The new query $Q^*_{\{\psi\}}$ *extends* the semantics of the original query based on the information provided by the trail definition, i.e., the new query may return additional results as a consequence of merging the trail.

3.4.2 Matching

In order to define trail matching, we must first introduce what we understand by query containment. Our definition closely follows the discussion of Miklau and Suciu [118].

DEFINITION 3.7 (QUERY CONTAINMENT) *Given two queries Q_1 and Q_2, we say Q_1 is contained in Q_2, denoted $Q_1 \subseteq Q_2$, if it holds that $\forall G: \ Q_1(G) \subseteq Q_2(G)$, where G is a finite resource-view graph.*

For the purposes of this chapter, the query containment tests we must perform are path-query containment tests. A sound containment test algorithm based on tree pattern homomorphisms is presented by Miklau and Suciu [118] and we followed their framework in our implementation. The algorithm is polynomial in the size of the query trees being tested. Details of our implementation are documented in the work of Girard [74].

Now, we are ready to define the conditions for matching trails with and without component projections to a query.

DEFINITION 3.8 (TRAIL MATCHING) *A trail* ψ matches *a query Q whenever its left side query, $\psi^{L.Q}$, is contained in a query subtree Q_S of the canonical form $\Gamma(Q)$ (see Section 3.2.3). We denote this containment condition as $\psi^{L.Q} \subseteq Q_S$. Furthermore, Q_S must be maximal, i.e., there must be no subtree $Q^{\hat{S}}$ of Q, such that $\psi^{L.Q} \subseteq Q^{\hat{S}}$ and Q_S is a subtree of $Q^{\hat{S}}$. If ψ does not contain a component projection, we require that Q_S not contain $\psi^{R.Q}$, i.e., $\psi^{R.Q} \not\subseteq Q_S$. We then take $Q_\psi^M := Q_S$. On the other hand, i.e., if ψ contains a component projection, we require that the component projection $\psi^{L.C}$ be referenced in the query in a selection by an operator immediately after Q_S in $\Gamma(Q)$. The matching subtree Q_ψ^M is then obtained by extending Q_S by the portion of the query referencing the component projection $\psi^{L.C}$.* □

EXAMPLE 3.3 Consider query Q_1 := `//home/projects//*["Mike"]`. Its canonical form is shown in Figure 3.3(a). Suppose we have a set of four trails, $\Psi = \{\psi_7, \psi_8, \psi_9, \psi_{10}\}$ with

ψ_7 := `Mike`$\longrightarrow$`Carey`

ψ_8 := `//home/*.name`$\longrightarrow$`//calendar//*.tuple.category`

ψ_9 := `//home/projects/OLAP//*["Mike"]`$\longrightarrow$
`//imap//*["OLAP" "Mike"]`

ψ_{10} := `//home//*`$\longrightarrow$`//smb//*`.

Trails ψ_7 and ψ_8 match Q_1 as $\psi_7^{L.Q}$ and $\psi_8^{L.Q}$ are contained in query subtrees of $\Gamma(Q_1)$. Furthermore, for ψ_7, the query subtree is maximal and does not contain $\psi_7^{R.Q}$; for ψ_8, the query operator immediately after one maximal subtree is a selection on the component projection `name`, i.e., `name="projects"` in the example. We display the matching query subtrees $Q_{1\,\psi_7}^M$ and $Q_{1\,\psi_8}^M$ in Figure 3.3(a). Note that the left side of a trail does not have to occur literally in $\Gamma(Q)$, but rather be contained in some query subtree of $\Gamma(Q)$. This situation is illustrated by ψ_9, which matches Q_1 as $\psi_9^{L.Q} \subseteq Q_1 = Q_{1\,\psi_9}^M$. In contrast, trail ψ_{10} does not match Q_1 as $\psi_{10}^{L.Q}$ is not contained in any subtree Q_S of $\Gamma(Q_1)$ such that Q_S does not contain $\psi_{10}^{R.Q}$. □

3.4.3 Transformation

The transformation of a trail is obtained by taking the right side of the trail and rewriting it based on the matched part of the query.

DEFINITION 3.9 (TRAIL TRANSFORMATION) *Given a query expression Q and a trail ψ_i without component projections, we compute the* transformation *$Q_{\psi_i}^T$ by setting $Q_{\psi_i}^T := \psi_i^{R.Q}$. For a trail ψ_j with com-*

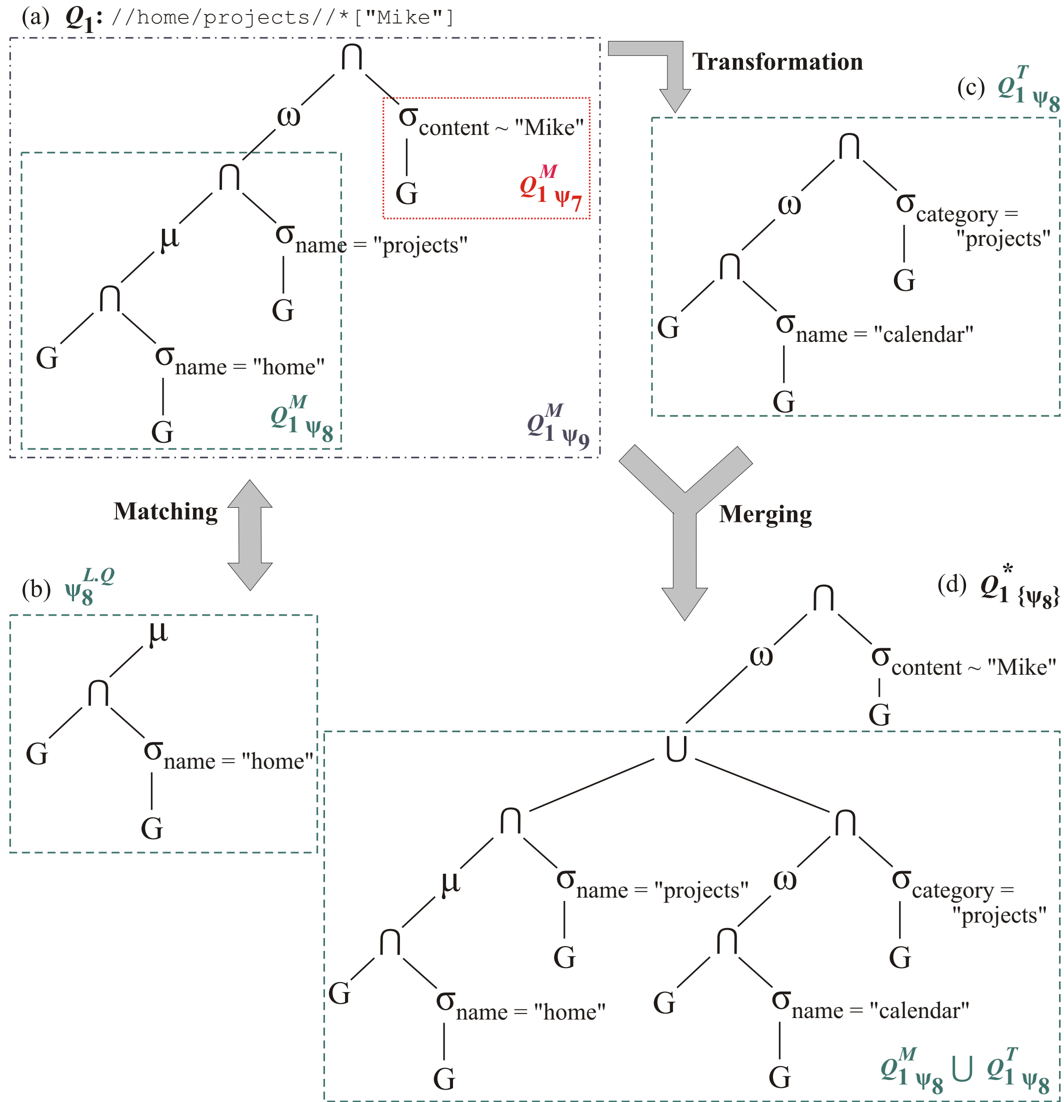


Figure 3.3: Trail Processing: (a) canonical form of Q_1 including matched subtrees $Q^M_{1\,\psi_7}$, $Q^M_{1\,\psi_8}$, and $Q^M_{1\,\psi_9}$, (b) left side of ψ_8, (c) transformation $Q^T_{1\,\psi_8}$, (d) final merged query $Q^*_{1\,\{\psi_8\}}$.

ponent projections, we take $Q^T_{\psi_j} := \psi_j^{R.Q} \cap \sigma_P(G)$. The predicate P is obtained by taking the predicate at the last location step of $Q^M_{\psi_j}$ and replacing all occurrences of $\psi_j^{L.C}$ with $\psi_j^{R.C}$. □

EXAMPLE 3.4 To illustrate Definition 3.9, consider again the trail ψ_8 and query Q_1 of Example 3.3. We know that ψ_8 matches Q_1 and, therefore, $Q^M_{1\,\psi_8}$ has as the top-most selection a selection σ on `name`. Figures 3.3(a) and (b) show $Q^M_{1\,\psi_8}$ and query $\psi_8^{L.Q}$ on the left side of trail ψ_8, respectively. The transformation of ψ_8 is performed by taking $\psi_8^{R.Q}$ and the predicate of the last selection of $Q^M_{1\,\psi_8}$, i.e., `name="projects"`. Both are combined to form a new selection `category="projects"`. Figure 3.3(c) shows the resulting transformation $Q^T_{1\,\psi_8}$ = `//calendar//*[category="projects"]`. □

3.4.4 Merging

Merging a trail is performed by adding the transformation to the matched subtree of the original query. We formalize this notion below.

DEFINITION 3.10 (TRAIL MERGING) *Given a query Q and a trail ψ, the* merging $Q^*_{\{\psi\}}$ *is given by substituting $Q^M_\psi \cup Q^T_\psi$ for Q^M_ψ in $\Gamma(Q)$.* □

EXAMPLE 3.5 Recall that we show the match of trail ψ_8 to query Q_1, denoted $Q^M_{1\,\psi_8}$, in Figure 3.3(a). Figure 3.3(c) shows the transformation $Q^T_{1\,\psi_8}$. If we now substitute $Q^M_{1\,\psi_8} \cup Q^T_{1\,\psi_8}$ for $Q^M_{1\,\psi_8}$ in $\Gamma(Q_1)$, we get the merged query $Q^*_{1\,\{\psi_8\}}$ as displayed in Figure 3.3(d). The new query corresponds to:

$Q^*_{1\,\{\psi_8\}}$ = `//home/projects//*["Mike"]` ∪ `//calendar//*[category="projects"]//*["Mike"]`.

Note that the merge was performend on a subtree of the original query Q_1. Therefore, the content filter on `Mike` now applies to the transformed right side of trail ψ_8. □

3.4.5 Multiple Trails

When multiple trails match a given query, we must consider issues such as the possibility of reapplication of trails, order of application, and also termination in the event of reapplications. While reapplication of trails may be interesting to detect patterns introduced by other trails, it is easy to see that a trail should not be rematched to nodes in a logical plan generated by itself. For instance, as $Q^*_{\{\psi\}}$ always contains Q, then if ψ originally matched Q, this means that ψ must match $Q^*_{\{\psi\}}$ (appropriately rewritten to canonical form). If we then merge $Q^*_{\{\psi\}}$ to obtain $Q^*_{\{\psi,\psi\}}$, once again ψ will match $Q^*_{\{\psi,\psi\}}$ and so on indefinitely. That effect also happens for different trails with mutually recursive patterns.

Multiple Match Coloring Algorithm (MMCA). To solve this problem, we keep the *history* of all trails matched or introduced for any query node. Algorithm 1 shows the steps needed to rewrite a query Q given a set of trails Ψ. We apply every trail in Ψ to Q iteratively and color the query tree nodes in Q according to the trails that have already touched those nodes. The algorithm operates by iterating through *levels* (Line 6). At each level, we match all trails against the current snapshot of the query (Lines 7 to 16). Matches are only allowed on nodes that are not already colored by the matching trail (Line 9). After computing all matches, then all matching trails are simultaneously merged to the query snapshot (Lines 19 to 29). The result is a new query, which is the input to the next level. If there was no change to the query between two levels or if we reach the maximum number of levels allowed in the rewrite, then the algorithm terminates.

EXAMPLE 3.6 (MULTIPLE TRAILS REWRITE) Recall the query `pdf yesterday` and the trails introduced in Use Case 3.1 of Section 3.3.2. Figure 3.4(a) shows the canonical form of that query. In the first level, Algorithm 1 matches and merges both trails ψ_3 and ψ_4 with the query. The result is shown in Figure 3.4(b). Now, all nodes in the merged subtrees have been colored according to the trails applied.

Algorithm 1: Multiple Match Coloring Algorithm (MMCA)

```
Input: Set of Trails Ψ = {ψ1, ψ2, ..., ψn}
Canonical form of query Γ(Q)
Maximum number of levels maxL
Output: Rewritten query tree Q_R
1  Set mergeSet ← <>
2  Query Q_R ← Γ(Q)
3  Query previousQ_R ← nil
4  currentL ← 1
5  // (1) Loop until maximum allowed level is reached:
6  while (currentL ≤ maxL ∧ Q_R ≠ previousQ_R) do
7      // (2) Perform matching on snapshot of input query Q_R:
8      for ψi ∈ Ψ do
9          if (Q^M_{R ψi} exists ∧ root node of Q^M_{R ψi} is not colored by ψi) then
10             Calculate Q^T_{R ψi}
11             Color root node of Q^T_{R ψi} with color i of ψi
12             Node annotatedNode ← root node of Q^M_{R ψi}
13             Entry entry ← mergeSet.getEntry(annotatedNode)
14             entry.transformationList.append(Q^T_{R ψi})
15         end
16     end
17     // (3) Create new query based on {node, transformationList} entries:
18     previousQ_R ← Q_R
19     for e ∈ mergeSet do
20         ColorSet CS ← (all colors in e.annotatedNode) ∪
21                   (all colors in root nodes of e.transformationList)
22         Node mergedNode ←
23                e.annotatedNode ∪ Q^T_{R ψi1} ∪ ... ∪ Q^T_{R ψik},
24                for all colors {i1, ..., ik} in e.transformationList
25         Color all nodes in mergedNode with all colors in color set CS
26         Calculate Q*_{R {ψi1,...,ψik}} by replacing e.annotatedNode by
27                 mergedNode in Q_R
28         Q_R ← Q*_{R {ψi1,...,ψik}}
29     end
30     // (4) Increase counter for next level:
31     currentL ← currentL + 1
32     mergeSet ← <>
33 end
34 return Q_R
```

During the second level, the algorithm successively matches trails ψ_1 and ψ_2 because the query now contains the merging of trail ψ_3. Figure 3.4(c) shows the final query tree after two levels. As nodes are colored with the history of trails applied to them, it is no longer possible to find a query subtree in which

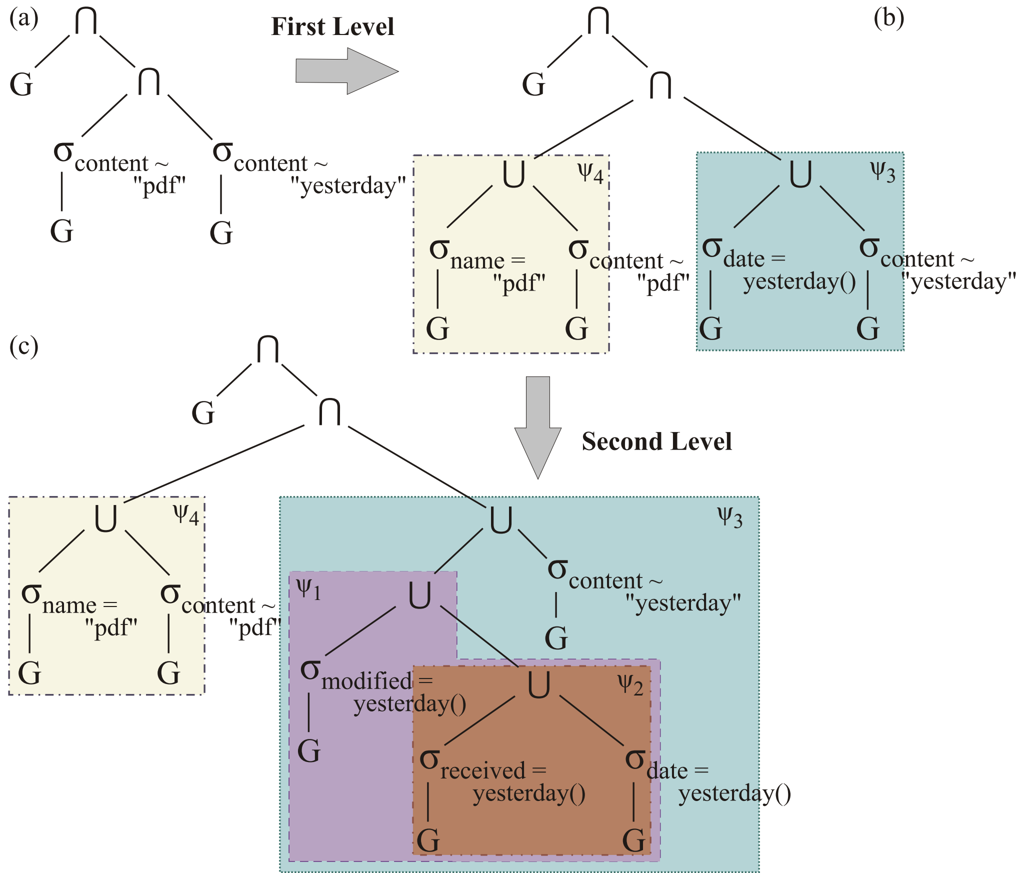


Figure 3.4: Algorithm 1 applied to Example 3.1. (a) original query, (b) rewritten query after one level, (c) after two levels.

a trail from Use Case 3.1 is matched and that is not already colored by that trail. Thus, the algorithm terminates. □

3.4.6 Trail Rewrite Analysis

THEOREM 3.1 (WORST CASE OF MMCA) *Let L be the total number of leaves in the query Q. Let M be a bound on the maximum number of leaves in the transformations of the trails in Ψ. Note that this bound may be obtained by inspecting the right sides of all trails in Ψ, as any transformation introduces at most one extra leaf in the canonical plan of the a trail's right side query (Definition 3.9). Let N be the total number of trails. Let $d \in \{1,\ldots,N\}$ be the number of levels. The maximum number of trail applications performed by MMCA and the maximum number of leaves in the merged query tree are both bounded by*

$$O\left(L \cdot M^d\right).$$

Proof. *Recall that if a trail ψ_i is matched and merged to a query Q, it will color leaf nodes in Q (the*

matched part $Q^M_{\psi_i}$ as well as the entire transformation $Q^T_{\psi_i}$). Thus, any subtree containing only these leaf nodes may not be matched again by ψ_i. In the worst case, in each level only one of the N trails matches, say ψ_i, for each of the L leaves of Q. Each trail match introduces M new leaves for each of those leaves. As a consequence, we obtain a total of LM new nodes plus L old nodes. In summary, we get $L(M+1)$ leaves and L trail applications for the first level. At the second level, ψ_i may not match any of the leaves anymore as they are all colored by the color i. However, all leaves may be matched against $N-1$ colors. In the worst case, again, only one of the trails matches for each of the existing leaf nodes. Thus, in the dth level, we obtain $L(M+1)^{d-1}$ trail applications and a total of $L(M+1)^d$ leaves. □

COROLLARY 3.1 (MMCA TERMINATION) *MMCA is guaranteed to terminate in $O(L \cdot M^d)$ trail applications.* □

The theorem above shows the behavior of MMCA when only one trail matches all leaves of the query tree at each level. It is simple to see that if all trails would match all leaves of the query tree at the first level, then MMCA would terminate after the first level as a consequence of all leaves in the query tree having been colored by all trails. In that case, we obtain a final query tree with *LNM* new nodes and *L* old nodes, i.e., the size of the rewritten query is $O(L \cdot (NM))$. We argue that the average behavior of MMCA will correspond to a scenario inbetween only a single matching trail per level and all trails matching at the first level. We explore such a scenario in the theorem below.

THEOREM 3.2 (AVERAGE CASE OF MMCA) *To model the average case we assume that at each level half of all remaining trails are matched against all nodes. Then, the maximum number of leaves is bounded by*

$$O\Big(L \cdot (NM)^{log(N)}\Big).$$

Proof. *At the first level, $N/2$ trails will match L nodes. This leads to $LN/2$ trail applications and $L(\frac{N}{2}M+1)$ leaves. At the second level, half of the remaining $N/2$ nodes will match. Thus we get $L(\frac{N}{2}M+1)\frac{N}{4}$ trail applications and $L(\frac{N}{2}M+1)(\frac{N}{4}M+1)$ leaves. Note that the number of levels in this scenario is bounded by $log_2(N)$. Consequently, the number of leaves at the last level $d = \lfloor log_2(N) \rfloor$ is developed as:*

$$L \prod_{i=1}^{\lfloor log_2(N) \rfloor} \Big(\frac{NM}{2^i} + 1\Big) \leq L \prod_{i=1}^{\lfloor log_2(N) \rfloor} \Big(2\frac{NM}{2^i}\Big) \leq L \cdot (NM)^{log_2(N)}.$$

□

We conclude that the total number of leaves in the rewritten query is exponential in the logarithm of the number of trails: $log_2(N)$. As this rewrite process may still lead to large query plans, we have developed a set of pruning techniques to better control the number of trail applications (see Section 3.5).

3.4.7 Trail Indexing Techniques

This section briefly presents indexing techniques for iTrails.

Generalized Inverted-List Indexing. To provide for efficient query processing of trail-enhanced query plans, one could use traditional rule and cost-based optimization techniques. However, it is important to observe that trails encode logical algebraic expressions that *co-occur* in the enhanced queries. Therefore, it may be beneficial to precompute these trail expressions in order to speed up query processing. Note that the decision of which trails to materialize is akin to the materialized view selection problem (we point the reader to the work of, for example, Harinarayan et al. [90]). However, in contrast to traditional materialized views, which replicate the data values, in many cases it suffices to materialize the object identifiers (OIDs) of the resource views. Thus, we may use a space-efficient materialization that is similar to the document-ID list of inverted lists [174]. For this reason, our indexing technique can be regarded as a generalization of inverted lists that allows the materialization of any query and not only keywords. An important aspect of our approach is that we do not materialize the queries but only the trails.

Materialize $\psi^{L.Q}$. Our first materialization option is to materialize the results of the query on the left side of a trail ψ as an inverted OID list. When $Q_\psi^M = \psi^{L.Q}$, the materialization of $\psi^{L.Q}$ may be beneficial. However, we may not always benefit from this materialization in general.

Materialize $\psi^{R.Q}$. We may choose to materialize the query expression on the right side of a trail definition. That is beneficial as $\psi^{R.Q}$ is a subtree of Q_ψ^T.

3.5 Pruning Trail Rewrites

In this section we show that it is possible to control the rewrite process of trail applications by introducing a set of pruning techniques. While we certainly do not exhaust all possible pruning techniques and trail ranking schemes in this section, the schemes proposed here demonstrate that trail rewrite complexity can be effectively controlled to make MMCA scalable.

3.5.1 Trail Ranking

Several of the pruning strategies presented in the following require the trails to be ranked. For that we define a trail weighting function as follows:

DEFINITION 3.11 (TRAIL-WEIGHTING FUNCTION) *Given the match Q_ψ^M of a trail ψ to a query Q, a* trail-weighting function *$w(Q_\psi^M, \psi)$ is a function that computes a weighting for the transformation Q_ψ^T.* □

We will use two different ranking strategies: The first takes into account the probability values assigned to the trails; the second also considers scoring factors (see Sections 3.3.4 and 3.3.5).

Ranking Trails by Probabilities

The first strategy to rank trails is to consider the probabilities assigned to the trails (see Section 3.3.4).

DEFINITION 3.12 (PROBABILISTIC TRAIL-WEIGHTING FUNCTION) *Given a probabilistic trail* ψ*, we define a* probabilistic trail-weighting function w_{prob} *as* $w_{prob}\left(Q_{\psi}^{M}, \psi\right) := p$*, where* p *is the probability value of* ψ*.* □

Here, the probability p provides a *static* (query-independent) ranking of the trail set – analogously to PageRank [25], which provides a query-independent ranking of Web sites. An interesting extension would be to rank the trail set *dynamically* (in query-dependent fashion), e.g., by considering how well a query is matched by a given trail. Dynamic trail ranking is an interesting avenue for future work.

Ranking Trails by Scoring Factors

The second strategy to rank trails is to also consider the scoring factors assigned to the trails (see Section 3.3.5).

DEFINITION 3.13 (SCORED TRAIL-WEIGHTING FUNCTION) *Given a probabilistic and scored trail* ψ*, we define a* scored trail-weighting function w_{scor} *as* $w_{scor}\left(Q_{\psi}^{M}, \psi\right) := w_{prob}\left(Q_{\psi}^{M}, \psi\right) \times sf$*, where* sf *is the scoring factor of* ψ*.* □

Ranking Query Results. While ranking trails is important to determine which trails are actually applied to a given query, an additional ranking decision is how to weight the *query results* produced by a trail. The scoring factor determines how relevant the scores of results obtained by a transformation Q_{ψ}^{T} are. Therefore, whenever a scoring factor is available, the trail transformation in Algorithm 1 (Line 11) is extended to multiply the scores of all results obtained from $Q_{\psi_i}^{T}$ by *sf*. In our implementation, we multiply these scores by introducing an additional *scoring operator* on top of $Q_{\psi_i}^{T}$. We also exploit the computed scores for Top-K processing [62] (see experiments in Section 3.6.2).

3.5.2 Pruning Strategies

Based on the trail ranking defined above we propose the following pruning strategies:

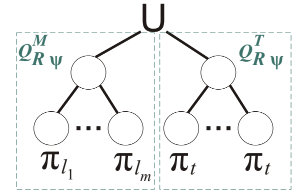
1. **Prune by Level,** i.e., punish recursive rewrites. The deeper the recursion, the further away the rewrite will be from the original query. We prune by choosing a value $maxL < N$ in Algorithm 1.
2. **Prune by Top-K Ranked Matched Trails.** We assume there is a relationship between trail scores and the benefit trails provide, i.e., we assume that the trails with the highest scores tend to provide the greatest benefit. Therefore, in this strategy we modify Algorithm 1 to only apply the $K < N$ top-ranked trails that *matched* in the current level.

3. **Other Approaches.** One approach is to use a timeout (similar to plan enumeration in cost-based optimization), i.e., trail rewrite could be stopped after a given time span. Another interesting approach would be to progressively compute results, i.e., deliver a first quick result to the user but continue the rewrite process in the background. Then, whenever a better result based on more trail rewrites is found, results could be updated progressively.

3.5.3 Rewrite Quality

Pruning may diminish the quality of the rewritten query with respect to the no-pruning case. Ideally, one would measure the quality of the queries obtained using standard metrics such as precision and recall. Obtaining actual precision and recall values implies executing the queries against a data collection with relevance judgements. However, for a pay-as-you-go information-integration scenario such dataset does not exist yet. Furthermore, we would like to have a means to estimate the plan quality *during* the rewrite process *without* having to execute the plan beforehand. Our solution to this challenge is to provide estimates on the plan quality based on the trail probabilities.

The intuition behind our model lies in two assumptions: 1. The original query Q returns only relevant results (precision=1); 2. Each trail rewrite adds more relevant results to the query (i.e., increases recall). If the trail probability is smaller than 1, the increase in recall may come at the cost of precision. The idea behind these two assumptions is that trails are primarily a recall-enhancement method.



We first discuss how to estimate the number of relevant results for an intermediary rewrite Q_R. We assume that every leaf l in the query plan of Q_R contributes a constant number of results c to the final query result. Out of these c results, a fraction π_l is relevant, i.e., $\pi_l \cdot c$ results. In the original query, we set $\pi_l = 1, \forall l$ (Assumption 1). Suppose trail ψ, with probability p, matches Q_R. The match $Q^M_{R\,\psi}$ includes leaves $l_1, \dots, l_m$ of Q_R. This is visualized in the figure on the right. The average proportion of expected relevant results produced by these leaves is $\bar{\pi}_l = \sum_{j=1}^{m} \pi_{l_j}/m$. We compute the proportion π_t of relevant results produced by every leaf node t of $Q^T_{R\,\psi}$ by $\pi_t = \bar{\pi}_l \cdot p$. So, if $p = 1$, we expect $Q^T_{R\,\psi}$ to contribute the same proportion of relevant results to the query as $Q^M_{R\,\psi}$. If $p < 1$, then that proportion is affected by our certainty about ψ. Thus, every leaf in $Q^T_{R\,\psi}$ contributes $\pi_t \cdot c$ relevant results to Q_R.

The expected precision for Q_R is computed as follows. Consider that Q_R has L leaves. Then, the expected precision $E_{Prec}(Q_R)$ is:

$$E_{Prec}(Q_R) := \frac{\sum_{i=1}^{L} \pi_i \cdot c}{L \cdot c} = \frac{\sum_{i=1}^{L} \pi_i}{L}.$$

That is, we sum the expected number of relevant results produced by each leaf and divide by the total expected number of results produced by all leaves.

The expected recall is more difficult to compute, as we do not know the total number of relevant results that exist in the dataset for Q_R. However, we may estimate the expected number of relevant results E_{Rel} returned by Q_R, which is directly proportional to the recall of Q_R. Thus, E_{Rel} is given by:

$$E_{Rel}(Q_R) := \sum_{i=1}^{L} \pi_i c.$$

That is, we sum the expected number of relevant results produced by each leaf.

3.6 Experiments

In this section we evaluate our approach and show that the iTrails method strongly enhances the completeness and quality of query results. Furthermore, our experiments demonstrate that iTrails can be efficiently implemented and scales for a large number of trails.

We compare three principal approaches (as mentioned in the Introduction):

1. **Semi-structured Search Baseline (No schema, NSA).** A search engine for semi-structured graph data providing keyword and structural search using NEXI-like expressions. We use queries that have no knowledge on integration semantics of the data.
2. **Perfect query (Schema first, SFA).** Same engine as in the baseline but we use more complex queries that have perfect knowledge of the data schema and integration semantics.
3. **iTrails (Pay-as-you-go).** We use the same simple queries as in the baseline but rewrite them with our iTrails technique.

We evaluate two major scenarios for iTrails.

Scenario 1: Few high-quality trails. Trails have high quality and a small-to-medium amount of trails are defined with human assistance. In this scenario, recursive trail rewrites may still exhibit high quality; in fact, trails may have been *designed* to be recursive.

Scenario 2: Many low-quality trails. A large number of trails are mined through an automatic process (see Section 3.3.3). Recursivity may be harmful to the final query plan quality and query rewrite times. Thus, it is necessary to prune trail rewrites.

The setup for both scenarios is presented in Section 3.6.1. The results for Scenario 1 are presented in Sections 3.6.2 and 3.6.3. The results for Scenario 2 are presented in Sections 3.6.4 and 3.6.5.

3.6.1 Data and Workload

System. All experiments were performed on top of our `iMeMex` Personal Dataspace Management System (see Chapter 5). The system was configured to simulate both the *baseline* and the *perfect query*

approaches. We have also extended `iMeMex` by an iTrails implementation. Our system is able to ship queries, to ship data, or to operate in a hybrid mode in between. However, as the latter features are orthogonal to the techniques presented here, we shipped all data to a central index and evaluated all queries on top of it.[2] The server on which the experiments were executed is a dual processor AMD Opteron 248, 2.2 Ghz, with 4 GB of main memory. Our code (including iTrails) is open source and available for download from *www.imemex.org*.

	Desktop	**Wiki4V**	**Enron**	**DBLP**	$\sum$
Gross Data size	44,459	26,392	111	713	71,675
Net Data size	1,230	26,392	111	713	28,446

Table 3.3: Datasets used in the experiments [MB].

Data. We conducted experiments on a dataspace composed of four different data sources similar to Figure 3.1. The first collection (Desktop) contained all files as found on the desktop of a colleague (one of the authors of the work by Salles et al. [143], which corresponds to an earlier version of this chapter). The second collection (Wiki4V) contained all English Wikipedia pages converted to XML [166] in four different versions, stored on an SMB share. The third collection (Enron) consisted of mailboxes of two people from the Enron Email Dataset [60], stored on an IMAP server. The fourth collection was a relational DBLP dataset [44] as of October 2006, stored on a commercial DBMS. Table 3.3 lists the sizes of the datasets.

The total size of the data was about 71.7 GB. As several files in the Desktop data set consisted of music and picture content that is not currently considered by our method, we subtracted the size of that content to determine the net data size of 28.4 GB. The indexed data had a total size of 24 GB.

Workload. For the first scenario, we employed an initial set of 18 manually-defined, high-quality trails, displayed on Table 3.4. Furthermore, we defined a set of queries including keyword queries but also path expressions. These queries are shown in Table 3.5. We present for each query the original query tree size, the final query tree size after applying trails, and the number of trails applied. As we may notice from the table, only a relatively small number of trails matched any given query (≤ 5). Nevertheless, as we will see in the following, the improvement in quality of the query results is significant. This improvement is thus obtained with only little investment into providing integration semantics on the queried information.

For the second scenario, we have randomly generated a large number of trails (up to 10,000) and queries with a mutual uniform match probability of 1%, meaning that we expect 1% of the trails defined in the system to match *any* initial query. Furthermore, we expect that *every* right side introduced by a trail application will again uniformly match 1% of the trails in the system and so on recursively. We have also assigned probabilities to the trails using a Zipf distribution ($skew = 1.12$). What we seek to model is a scenario with a small number of relatively high-quality trails and a large number of low-quality trails. This may happen as a consequence of a few trails having been defined manually or confirmed via user

[2]Thus, we made our system behave like an XML-IR search engine. In the future, we plan to explore mediation features of our system.

pics	⟶	//photos//* ∪ //pictures//*
//*.tuple.date	⟷	//*.tuple.lastmodified
//*.tuple.date	⟷	//*.tuple.sent
pdf	⟶	//*.pdf
yesterday	⟶	date=yesterday()
publication	⟶	//*.pdf ∪ //dblp//*
//*.tuple.address	⟷	//*.tuple.to
//*.tuple.address	⟷	//*.tuple.from
excel	⟷	//*.xls ∪ *.ods
//*.xls	⟷	//*.ods
//imemex/workspace	⟶	//ethz/testworkspace ∪ //ethz/workspace
//ethz/testworkspace	⟷	//ethz/workspace
music	⟶	//*.mp3 ∪ //*.wma
working	⟶	//vldb//* ∪ vldb07//*
paper	⟶	//*.tex
//vldb	⟶	//ethz/workspace/VLDB07
email	⟶	class=email
mimeType=image	⟶	mimeType=image/jpeg

Table 3.4: Trails used in Scenario 1.

feedback (using techniques such as presented by Jeffery et al. [98]), while at the same time a large number of trails having been mined by automatic processes. In both cases, trails may be recursive among each other and it is reasonable to assume that not all trails will match any query given as input to the system.

Query Number	Query Expression	Original Tree Size	Final Tree Size	Number of Trails Applied
1	//bern//*["pics"]	6	14	1
2	date > 22.10.2006	2	8	2
3	pdf yesterday	5	44	4
4	Halevy publication	5	12	1
5	address=raimund.grube@enron.com	2	8	2
6	excel	2	8	2
7	//imemex/workspace/VLDB07/*.tex	14	35	2
8	//*Aznavour*["music"]	5	11	1
9	working paper	5	41	5
10	family email	5	8	1
11	lastmodified > 16.06.2000	2	8	2
12	sent < 16.06.2000	2	8	2
13	to=raimund.grube@enron.com	2	8	2
14	//*.xls	2	5	1

Table 3.5: Queries for trail evaluation in Scenario 1.

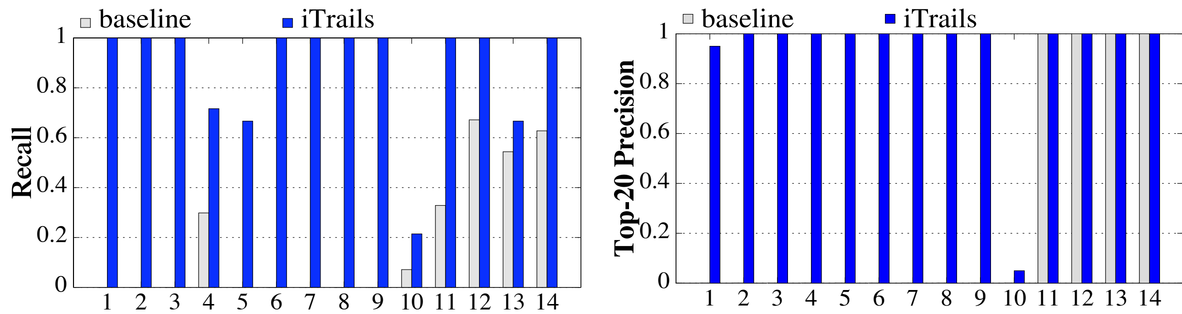


Figure 3.5: Scenario 1 – Recall and Top-20 precision of query results: iTrails compared to baseline.

3.6.2 Quality and Completeness of Results

In this section, we evaluate how iTrails affects the quality of query results. We measured quality of query results using standard precision and recall metrics, where the relevant results were the ones computed by the *perfect query* approach. Thus, the *perfect query* approach has both precision and recall always equal to 1.

Trails versus Baseline. Figure 3.5 shows the total recall and Top-20 precision for iTrails compared to the baseline approach. The baseline approach achieves only low recall and sometimes even zero recall. In contrast, iTrails achieves perfect recall for many queries, e.g., Q1–Q3 and Q6–Q9. This perfect recall is a consequence of our approach exploiting the trails to rewrite the query. For Top-20 precision, the baseline approach fails to obtain any relevant result for Q1–Q10, whereas it achieves perfect precision for Q11–Q14. This perfect precision by the baseline approach for Q11–Q14 is a consequence of these queries having partial schema knowledge that favors the baseline approach. In contrast, iTrails shows perfect precision for all queries except for Q1, which still achieves 95% precision, and Q10, which only achieves 5% precision. This low precision can be sharply improved by adding more trails in a pay-as-you-go fashion. For instance, by simply adding a trail `family` $\longrightarrow$ `//family//*` precision and recall for Q10 could both be increased to 1. The same holds for Q4, Q5, and Q13, where the definition of three additional trails would improve recall.

In summary, our experiments show that iTrails sharply improves precision and recall when compared to the baseline approach.

3.6.3 Query Performance

In this section we evaluate how the iTrails method affects query-response time.

Trails versus Perfect Query. We compared iTrails with the perfect-query approach. Note that we did not compare against the original queries and keyword searches, as they had unacceptably low precision and recall. Moreover, with respect to execution time, it would be simply unfair to compare keyword searches against complex queries with structural constraints. Table 3.6 shows the results of our experiment. In the first two columns of the table, we show the response times for iTrails and for the perfect query approach

Query Number	Perfect Query with Basic Indexes	iTrails	
		with Basic Indexes	with Trail Materialization
1	0.99	2.18	0.21
2	1.10	0.74	0.52
3	4.33	10.72	0.39
4	0.39	1.86	0.07
5	0.29	0.56	0.44
6	0.14	0.32	0.05
7	0.63	1.73	0.67
8	1.55	5.27	0.48
9	186.39	179.02	1.50
10	0.65	10.14	0.29
11	0.68	0.60	0.60
12	0.67	0.60	0.60
13	0.28	0.49	0.44
14	0.14	0.14	0.14

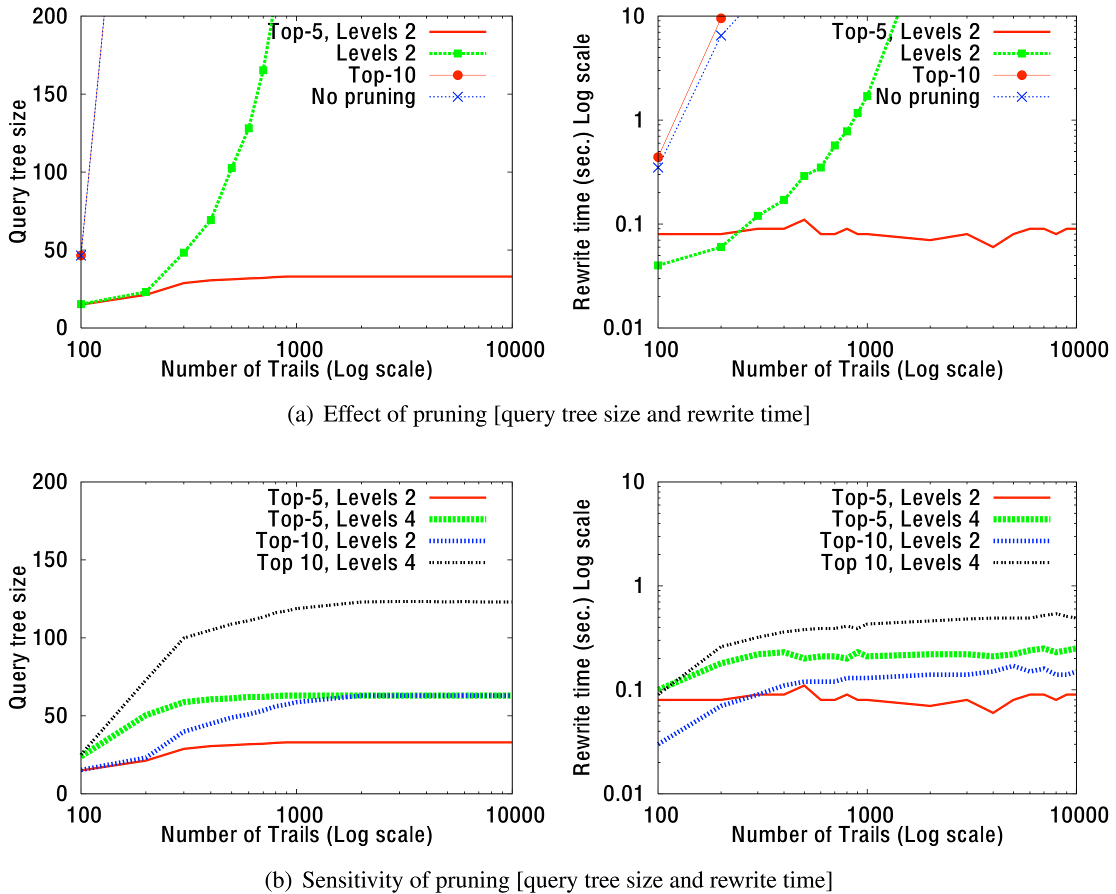
Table 3.6: Scenario 1 – Execution times of queries [sec]: iTrails with and without trail materialization compared to perfect query.

using only the basic indexes provided by our system. These indexes include keyword inverted lists and a materialization for the graph edges. The table shows that iTrails-enhanced queries, for most queries, needed more time than the perfect queries to compute the result. We thus conclude that the overhead introduced by the trail rewrite is *not* negligible. This overhead, however, can be fixed by exploiting trail materializations as presented in Section 3.4.7. The results in the third column of the table show that almost all queries benefit substantially from trail materializations. In contrast to iTrails, the perfect query approach is not aware of the queries encoded in the trails and, therefore, cannot materialize trail queries in advance. Furthermore, as we assume that Q1–Q14 are not known to the system beforehand, additional materialization for the perfect queries cannot be used. With trail materialization, Q3 can now be executed in 0.39 seconds instead of 10.72 seconds (a factor of 27). For Q9 the improvement is from 179.02 to 1.50 seconds (a factor of 119). In summary, using trail materializations, all queries executed in less than 0.7 seconds, except for Q9 which took 1.50 seconds. However, for this query the majority of the execution time was wasted in a deep unnest operator. We could further improve the execution time of this query, for example, by adding a reachability index [158].

In summary, our evaluation shows that trail materialization significantly improves response time with respect to the perfect-query approach.

3.6.4 Rewrite Scalability

We now turn our attention to Scenario 2 and evaluate the effect of trail pruning on the scalability of trail rewrites. In Figure 3.6(a), we show how the number of operators in the rewritten query grows as the num-



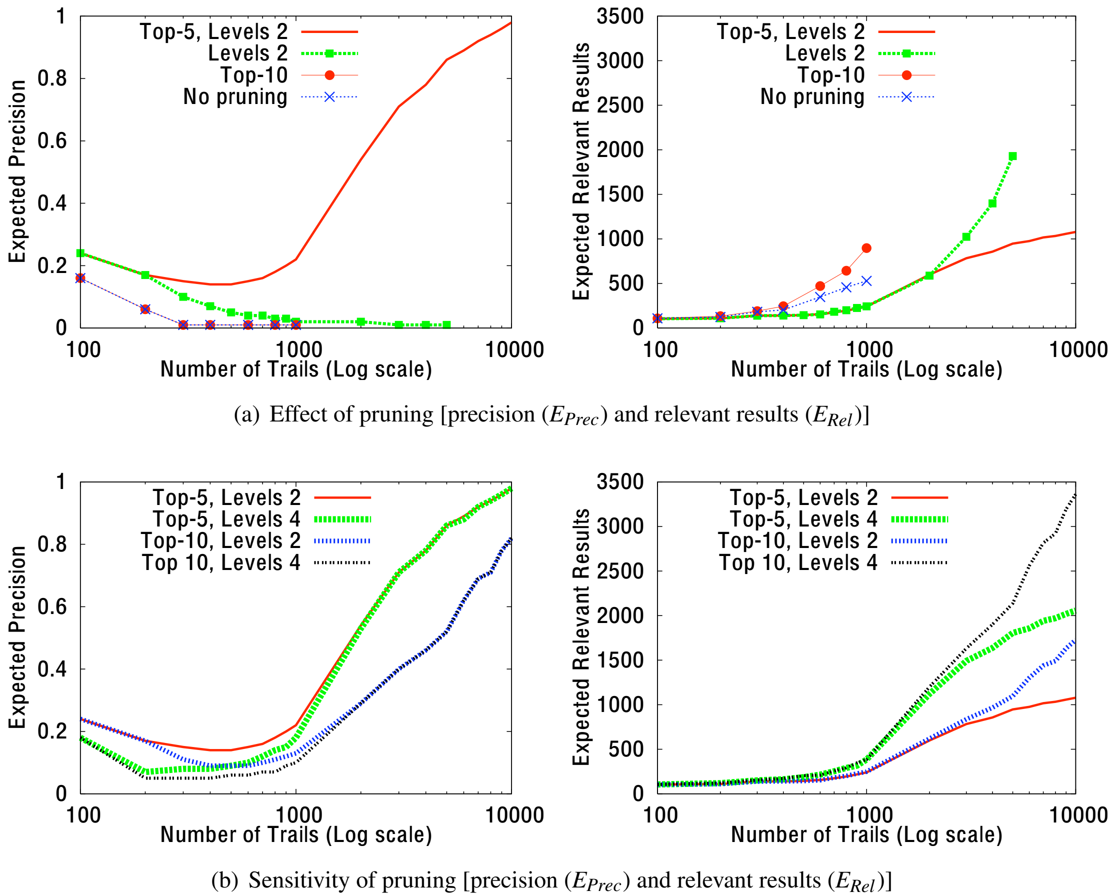
(a) Effect of pruning [query tree size and rewrite time]

(b) Sensitivity of pruning [query tree size and rewrite time]

Figure 3.6: Scenario 2 – Scalability of Trail Pruning: Effect and Sensitivity

ber of trails in the system is scaled. We have evaluated the following strategies: (1) no pruning, (2) Top-K matched pruning, (3) maximum-level pruning, (4) combined Top-K and maximum level pruning (see Section 3.5.2). Although we have experimented with a number of different parameters for the maximum number of trails matched per level (K) and for the maximum number of levels (Levels), in Figure 3.6(a) we show only representative parameter choices for each of the strategies. As expected according to our rewrite analysis (see Section 3.4.6), the *No pruning* approach exhibits exponential growth in the query plan sizes, thus becoming nonviable for barely more than a hundred trails. It may seem surprising that the growth of the *Top-10* pruning approach is as steep (or even steeper as the graph for rewrite times in Figure 3.6(a) suggests) as the growth for the *No pruning* approach. However, that observation is also backed by the worst and average case analyses of Section 3.4.6: given that we do not limit the number of levels, the fewer trails matched at each level, the more trails will be available for matching when the tree size is larger. Thus, as demonstrated in Section 3.4.6, the worst-case behavior occurs when only one trail matches per level at each of the query leaves.

Maximum-level pruning imposes a constant limit on the number of levels. Therefore, the growth function is no longer bounded by an exponential but rather by a polynomial of the number of levels chosen (see



(a) Effect of pruning [precision (E_{Prec}) and relevant results (E_{Rel})]

(b) Sensitivity of pruning [precision (E_{Prec}) and relevant results (E_{Rel})]

Figure 3.7: Scenario 2 – Quality of Trail Pruning: Impact and Sensitivity

Section 3.4.6). If we take the number of levels equal to 2 as shown in Figure 3.6(a), up to about 1,000 trails we obtain reasonably sized query plans with a rewrite time below 1 second. However, as we scale beyond that threshold both tree sizes and rewrite times become unacceptable. We remedy this situation by combining Top-K and maximum-level pruning. Figure 3.6(a) shows that this combination is the most effective rewrite pruning technique (*Top-5, Levels 2*). The combined technique effectively provides for a maximum number of trail applications that may be performed on the query. Thus, as the number of matching trails grow, we eventually reach a plateau on the number of trails that can be applied. As we always choose the trails to be applied based on their rank, we expect to expand the query with only the most relevant trail applications.

Figure 3.6(b) displays the sensitivity of the combined pruning technique to the settings of K and maximum number of levels. Taking larger values for K and for the number of levels allows us to produce larger query trees that include the application of more trails. Depending on the level of skew of the trail ranks and on the focus on precision or recall, different applications may choose different combinations of the parameters K and number of levels. For all settings explored in Figure 3.6(b), rewrite time has remained below 1 second. Rewrite time, just as tree size, also reaches a plateau. This effect is a consequence of our

trail ranking scheme exploiting only static ranking. Dynamic ranking schemes would lead to an expected slight growth in the rewrite time, as we would need to apply threshold-style algorithms [62] to calculate the top-ranked trails.

In summary, our experiments show that our pruning techniques can effectively control the complexity of the trail-rewrite process.

3.6.5 Rewrite Quality

We evaluate the effect on expected quality as defined in Section 3.5.3 using the strategies studied in Section 3.6.4. Figure 3.7 shows the results. We computed expected relevant results (E_{Rel}) by taking $c = 100$ (see Section 3.5.3). Both the *No pruning* and the *Top-10* strategies could not be scaled beyond 1000 trails due to their long rewrite times. *Levels 2* could not be scaled beyond 5000 trails.

Figure 3.7(a) shows that expected precision drops significantly for the *No pruning*, *Top-10*, and *Levels 2* strategies as the number of trails increases. This behavior is due to the fact that these rewrite strategies consider many trails with very low probabilities. Therefore, although these strategies obtain increasing expected numbers of relevant results as the number of trails grow, they pay a high cost in terms of precision. On the other hand, the combined Top-K and maximum level pruning strategy (*Top-5, Levels 2*) shows improvements in expected precision for larger number of trails. As discussed in Section 3.6.4, this strategy effectively limits the rewrite size, performing only up to a fixed number of trail applications. As a consequence, as more trails are available, the combined strategy has more higher-probability matching trails to pick from.

We experiment further with the expected precision versus expected relevant results trade-off by varying K and maximum number of levels in the combined strategy. Figure 3.7(b) shows the results. For all combinations of parameters, we observe that there is a decrease in precision for lower numbers of trails. For example, expected precision decreases for *Top-5, Levels 4*, while the number of trails is below 300. This behavior is due to the fact that when relatively few matching trails are available, we still apply many trails with low probabilities. As the number of trails increases, expected precision also starts to increase.

The effect of picking trails with low probabilities is also evidenced by the fact that the more trail applications that are allowed by the settings of K and maximum number of levels, the lower overall precision is observed (e.g., precision for *Top-5, Levels 2* is higher overall than for *Top-10, Levels 4*). At the same time, the same settings produce higher numbers of expected relevant results. When 10,000 trails are defined, the number of relevant results is three times higher for *Top-10, Levels 4*, than for *Top-5, Levels 2*. In addition, the numbers of relevant results rise for larger number of trails for all parameter settings. This trend is expected since when a large number of trails is available, the trails picked by the pruning strategies have higher probabilities.

In summary, our experiments show that our pruning techniques provide high expected precision rewrites, while allowing us to trade-off for the expected number of relevant results produced.

3.7 Related Work

Query processing in Dataspace Management Systems raises issues related to several approaches in data integration and data management in the absence of schemas. We discuss how our approach compares to important existing methods below.

Search Engines and Information-Integration Systems. We point the reader to Section 1.2 for a review of the related work on search engines and information-integration systems. The important aspects to be highlighted here regarding those systems are that iTrails enables us to build an information-integration architecture that stands in-between search engines and information-integration systems. With iTrails, we are able to enrich a search engine with integration semantics, given in the form of trails, while at the same time not requiring significant up-front integration effort, as it is the case with information-integration systems.

We may regard iTrails as a generalized method to specify equivalences among schema and instance elements. Unlike in previous work on schema matching [136], however, our approach is much more lightweight, allowing users to provide arbitrary hints over time in a pay-as-you-go fashion.

Fuzzifying Schemas. In architectures where structured and unstructured information is mixed, tools are necessary to allow querying without or with partial schema knowledge. Data guides [75] derive schema information from semi-structured data and make it available to users. However, data guides do not allow users to pose queries that are schema unaware. Schema-Free XQuery [111] attacks this problem by employing heuristics to detect meaningfully related substructures in XML documents. These substructures are used to restrict query results. Furthermore, tag names may also be expanded using synonyms similarly to the work of Qiu and Frei [135]. In contrast, iTrails are not restricted to synonyms or semantic relationships in a search-based scenario, allowing richer semantic equivalences to be specified, i.e., for information integration. In addition, our approach does not require any heuristic assumptions about what are meaningful relationships among nodes in the dataspace graph.

Malleable schemas [54] fuzzify the specification of attribute names, by associating several keywords or keyword patterns to each malleable attribute. Our approach generalizes and includes malleable schemas, as we may specify trails with semantic equivalences on attribute names. Chang and Garcia-Molina [33] approximate Boolean predicates to be able to consider the closest translation available on a data source's interface. They also enable users to provide rules to specify semantic correspondences among data source predicates, but their approach is restricted to structured sources, while our approach works for structured and unstructured data. Surf trails [32] are simple logs of a users' browsing history. They are explored to provide context in query processing. In sharp contrast, iTrails is semantically more expressive, allowing several types of query reformulation and not merely providing context information.

Ontologies. RDF and OWL [130, 138] allow data representation and enrichment using ontologies. RDF and OWL are reminiscent of schema-based approaches to managing data, requiring significant effort to declare data semantics through a complex ontology to enrich query processing. By contrast, iTrails allows a much more lightweight specification that enables gradual enrichment of a loosely integrated dataspace,

adding integration information in a pay-as-you-go fashion. It could be argued that trail rewriting may be seen as similar to reasoning with ontologies. Our trail rewriting algorithms, however, coupled with ranking and pruning techniques, provide a simpler and more scalable mechanism to control the complexity of the rewriting process.

Initial Dataspace Research. Franklin, Halevy and Maier [65, 88] have proposed dataspaces as a new abstraction for information management. Dataspaces have been identified as one key challenge in information integration [89]. Recently, web-scale pay-as-you-go information integration for the Deep Web and Google Base has been explored by Madhavan et al. [115]. However, the authors present only a high-level view on architectural issues — the underlying research challenges are not tackled. In contrast to the latter work, the approach developed in this chapter is the first to provide an actual *framework* and *algorithms* for pay-as-you-go information integration. More recently, a dataspace approach has been proposed to reduce the costs to setup an information integration system and exploit user feedback [98, 144]. That approach is focussed on traditional structured data integration and, in contrast to iTrails, does not target the heterogeneous unstructured and structured data mix typically found in personal and social dataspaces.

3.8 Conclusions

The work presented in this chapter has explored pay-as-you-go information integration in dataspaces. As a solution, we have proposed iTrails, a declarative method that allows us to provide integration semantics over time without ever requiring a mediated schema. Our approach is an important step towards the realization of dataspace management systems [65]. We have discussed rewrite strategies for trail-enriched query processing. Our rewrite algorithm is able to control recursive firings of trail matchings. Furthermore, we extended that algorithm to allow pruning of trail rewrites based on quality estimations. Our experiments showed the feasibility, benefits, and scalability of our approach. It is also worth mentioning that the techniques presented in this paper are not restricted to a specific data model, e.g., the relational model or XML, but work for a highly heterogeneous data mix (such as described in Chapter 2), including personal, enterprise, or web sources.

As part of ongoing and future work, we plan to explore other trail classes that allow us to express different integration semantics than the ones currently expressible by iTrails and by the association trails that will be introduced in the next chapter. Furthermore, we plan to further investigate techniques for semi-automatically mining trails from the contents of a dataspace, extending initial work done by Schmidt [146]. In that context, defining trails for mediated sources, such as hidden-web databases, is an especially interesting research challenge.

Chapter 4

Association Trails: Modeling and Indexing Intensional Associations in Dataspaces

Dataspace architectures allow users to take a pay-as-you-go approach for information integration. The core idea is to offer basic search services from the start and to improve the quality of query answers over time by modeling relationships between the data items in the dataspace. Recent dataspace approaches, such as the approach of Das Sarma et al. [144] or our approach from Chapter 3, are able to model relationships among schemas or sets of data items. There are scenarios, however, that necessitate modeling of finer-grained, instance-level relationships not currently supported by those state-of-the-art dataspace approaches. For example, in a social network, we would like to declare relationships among individual persons and to exploit those relationships for querying. Such relationships should be modeled intensionally, via view definitions, rather than extensionally, in an instance-by-instance basis. In this chapter, we propose a powerful technique based on *association trails* that allows users to define a graph of connections among instances in a dataspace declaratively. We study how to support exploratory queries on top of this intensional graph by providing several query processing and indexing strategies. By carefully combining the intensional graph computation with exploratory query processing, our strategies are able to answer queries over the intensional graph without ever having the need to fully materialize it. A set of experiments over a synthetically generated social network confirm that our strategies may provide significant savings when compared to materializing the intensional graph defined by association trails naively.

4.1 Introduction

Dataspace systems have been envisioned as a new architecture for data management and information integration [65]. As explained in Chapter 1, the core idea of dataspace systems is to offer a hybrid information-integration architecture that lies inbetween search engines [9, 25, 29, 156] and traditional information-integration systems [69, 109, 124, 131, 154]. We may build such an architecture by offering search services over all of the data sources from the start and enabling users to add relationships to the

system that increase the level of integration in a pay-as-you-go fashion. The system takes advantage of these relationships to provide better query results, enhancing precision and recall as more is known about the data.

So far, relationships in dataspace systems have been specified at the set-level, as presented in Chapter 3, or at the schema-level, as discussed by Das Sarma et al. [144]. These relationships are exploited to rewrite queries posed by the user to the system. For example, as shown in Chapter 3, a user could define a trail, in this chapter termed *semantic trail*, that relates all items in the path `//projects/PIM` to items in the path `//mike/research/PIM`. During query processing, queries that reference `//projects/PIM` would be rewritten by the semantic trail to obtain also elements that lie in `//mike/research/PIM`.

There are scenarios, however, that necessitate modeling of relationships at a finer-grained, instance level. These include applications such as scientific data management [92], semantic web [126, 164], personal information management [58] (see also Chapter 2), social networking [7], and metadata management [153], to name a few. For example, in scientific applications, it is often necessary to associate provenance information to data products; in personal information management, it is useful to associate messages and documents received in the same context or timespan; in social networking, it is fundamental to create relationships between persons. The set of associations among instances forms a graph, which must be evolved over time to incorporate knowledge about new relationships and new instances, in a pay-as-you-go fashion. Moreover, users wish to navigate and pose exploratory queries on top of this graph.

State-of-the-art approaches are ill-equipped to handle the needs of the applications mentioned above. Previous works in dataspace systems, such as in Chapter 3 and in the approach of Das Sarma et al. [144], only operate at a coarse-grained, set- or schema-level. These systems translate queries among different source semantics or schemas, but do not model a graph of relationships among individual instances. In contrast, we would like to enable users to define instance-level relationships and to navigate the resulting graph with exploratory queries. Some solutions have been developed that define associations extensionally, in an instance-by-instance basis [17, 61, 73, 116]. In contrast, we would like to enable users to work with intensional definitions, which include extensional definitions as a special case. For example, in a social network, we might want to define that all people that attend the same yoga course are connected in this context, without having to explicitly add edges among all these people as done in current systems. Finally, graph indexing approaches [55, 126, 158, 164] improve query-response times for different classes of queries over instance graphs. However, these systems are grounded on a fundamental assumption: The graph is extensionally given as input to the indexing system. In constrast, we would like to enable users to specify the data graph intensionally, via graph view definitions. Furthermore, we would like to optimize query processing across the graph view.

This chapter proposes techniques to provide support for these advanced data-management applications. Our contributions are two-fold. First, we propose a model based on *association trails* to define instance graphs intensionally. Association trails are query-based definitions of fine-grained relationships among instances. The set of instances and relationships forms a graph over which queries may be posed. Second, we show how to process exploratory queries on top of this intensional graph. The intuition we follow is similar to the intuition behind view merging during query rewriting [134]. In view merging, it has been

shown that optimizing the user query and the view definition together can provide orders of magnitude gains in performance when compared to processing the view and the query independently. Likewise, instead of materializing the graph and then processing queries over it, we combine these two steps, resulting in query processing strategies that may answer queries without ever having to materialize the whole graph. Our experiments show that naively materializing the intensional graph and indexing it may turn out to be over an order of magnitude more expensive than following our intuition of combining query processing with the intensional graph view definition.

4.1.1 Motivating Example

We will use a social networking application as a running example for this chapter. Figure 4.1(a) shows an example of a typical social network today. Users are depicted along with a snippet of information from their profiles, stating the universities they have attended, their years of graduation, and hobbies that they have. Moreover, users connect to other users by adding them as friends in the network (isFriend edges in the figure). Users also post comments to communities to express their opinion about a variety of topics (postedComment edges).

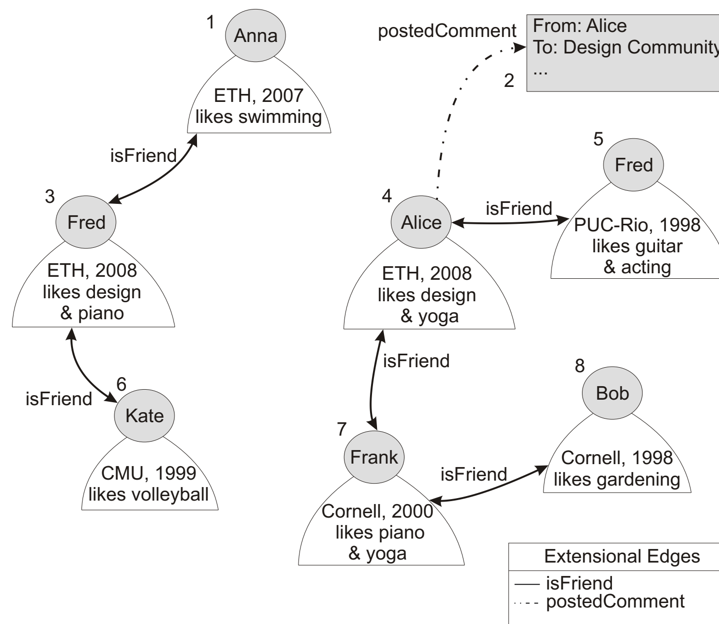
In the following, we discuss two examples of how to connect people in the social network and how to perform exploratory querying on it.

EXAMPLE 4.1 (INTENSIONAL GRAPH) *"When Alice joins the social network, she would like to connect to other people with similar interests. She would also like to record the context in which she relates to each person."*

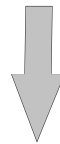
State-of-the-art: Current social networking systems allow Alice to establish isFriend associations with the people she is acquainted with, as depicted in Figure 4.1(a). Alice (Node 4) must explicitly connect to each person by adding an isFriend edge. As a consequence, when Alice joins the network, she has to go through a long and tedious process of finding people and building up her network of acquaintances. Some systems, e.g., Facebook [61], try to offer suggestions of people Alice might know based on information from her profile, e.g., people that went to the same university. Those suggestions only take into consideration one or two fixed attributes from Alice's profile, however, forcing her to review long lists of potential candidates. In addition, there is limited or nonexistent support to record the context of the associations Alice establishes. For instance, Alice cannot state that she relates to Frank (Node 7) because they share a passion for yoga. At a recent panel in the MSR Faculty Summit [137], the problem of providing context for the connections among users has been pointed out as one of the major limitations of current social networking systems.

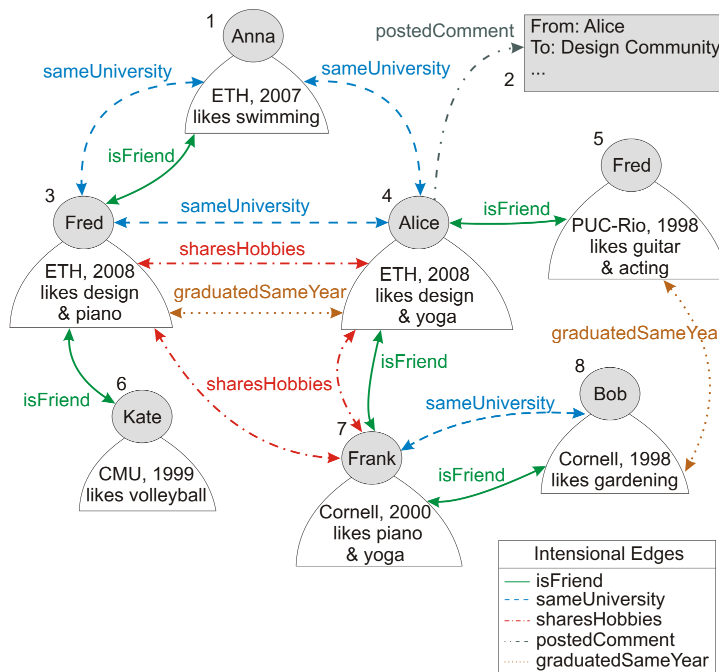
Our goal: To provide a technique that allows Alice to intensionally connect to other people in different contexts. We would like to allow users or administrators of the network to provide to the system general association definitions, in this chapter called *association trails*. Consider that the following definitions are given to the system:

1. People that went to the same university are related.



(a) A Social Network Today





(b) A Social Network with Association Trails

Figure 4.1: Running Example: (a) social networks today offer an extensional graph of users and manually-defined connections; (b) with association trails, a richer graph is defined intensionally.

2. People that graduated on the same year are related.
3. People that share a hobby are related.
4. People that connect to each other as friends are related.
5. People are related to the comments they have posted.

Given the association trails listed above, when Alice joins the network and fills out her profile, she would be *instantaneously* connected to a number of related people in the network. In addition, each connection would specify the context in which it is valid. Although Alice would still be able to add supplementary isFriend edges, she would not have to go through a long build-up period to connect to other people in the social network. In fact, Alice would have the view of the network depicted in Figure 4.1(b), in which she is connected to Frank both because they share a hobby and because they are friends. In addition, Alice's intensional network obviates the need for suggestions based on only a few fixed attributes from her profile. Alice can navigate directly to people that are related to her and has a way to determine how strong their ties are based on the number of intensional edges that connect her to any person in her neighborhood. For example, Alice would be connected by three distinct edges to Fred (Node 3), but by only one to Anna (Node 1). □

EXAMPLE 4.2 (EXPLORATORY QUERYING) *"Alice would like to meet new people. To find interesting people, she would like to find out which people in the network have a lot to do with her friends."*

State-of-the-art: Current social networking systems allow Alice to browse through the network. Alice can visit the pages of her friends and then snowball into the pages of the friends of her friends. The system provides no support for ranking the several thousands of profiles Alice would have to manually examine to find out how much a person is indeed related to her friends.

Our goal: To provide methods for processing exploratory queries in the intensional graph defined by association trails. In this example, we are interested in the *neighborhood* of Alice's friends in the graph of Figure 4.1(b). That neighborhood comprises Bob (Node 8), who is related to both Frank (Node 7) and Fred (Node 5). Unlike current systems, we would like to provide as evidence for ranking the fact that Bob has three relationships with the friends of Alice. Furthermore, if these relationships have different weights, then that should be also taken into account. For instance, the fact that Bob is both a friend of Frank and went to the same university as he did might have more weight than the fact that Bob graduated in the same year as Fred. □

4.1.2 Contributions

In summary, this chapter makes the following contributions:

1. We present a new formalism, termed *association trails*, to define a graph of connections among instances in a dataspace intensionally. Association trails may be added to the system over time, allowing

for pay-as-you-go enrichment of the data graph. Previous dataspace approaches, such as in Chapter 3 and in the work of Das Sarma et al. [144], define relationships at the set- or schema-level. In contrast, association trails define fine-grained connections among individual instances in a dataspace.

2. We show how to evaluate exploratory queries on top of the intensional graph defined by association trails. The key intuition is to combine the query with the selective computation of the necessary portions of the graph view in a single query plan. In this way, we are able to process queries on the graph without having to pre-materialize the whole graph explicitly.
3. In order to speed up query processing with association trails on large graphs of associations, we propose a set of indexing techniques. Each of our indexing options show how to take advantage of different amounts of materialization of the graph view in order to improve query processing.
4. We show a ranking scheme for exploratory queries that takes into consideration the number of edges (in-degree) connecting items in the neighborhood of primary query results with the primary query results themselves.
5. In a set of experiments with a synthetically generated social network, we evaluate the performance of our query processing techniques over intensional graphs. Our results show at least an order-of-magnitude improvement over the naive approach of fully materializing the intensional graph and then processing queries over it.

This chapter is structured as follows. Section 4.2 presents the data and query model adopted in this chapter, which is a subset of the model described in Chapter 2. Section 4.3 introduces association trails. Query processing with association trails is discussed in Section 4.4. We proceed by presenting indexing techniques for the intensional graphs defined by association trails in Section 4.5. Our ranking model is described in Section 4.6. Experiments that evaluate the performance of query processing with association trails and indexing are shown in Section 4.7. Afterwards, we review related work (Section 4.8) and conclude the chapter (Section 4.9).

4.2 Data and Query Model

In this section, we introduce the basic data and query model on top of which we will define association trails. Our goal is to describe how data is represented in state-of-the-art systems, such as displayed on the social network of Figure 4.1(a). Furthermore, we also briefly discuss how these systems provide basic indexing services on the data.

4.2.1 Data Model

We begin by defining the data model used to represent data extracted from the data sources and made available in the dataspace. The data model below is a simplified version of the data model introduced in Chapter 2, with the extension that edges are labeled.

DEFINITION 4.1 (DATASPACE GRAPH) *The data on the dataspace is represented by a* dataspace graph $G := (N, E)$*, where:*

1. *N is a set of nodes $\{N_1, \ldots, N_m\}$. Each node N_i is a set of attribute-value pairs $N_i := \{(a_1^i, v_1^i), \ldots, (a_k^i, v_k^i)\}$. We do not enforce a schema over all the nodes, i.e., the set of attributes of each node may be different.*
2. *E is a set of labeled, directed edges (N_i, N_j, L), such that $N_i, N_j \in N$ and L is a label.* □

EXAMPLE 4.3 The data in Figure 4.1(a) is represented in the data model of Definition 4.1. It is a graph with a set of nodes and labeled edges. Node 4 has a set of attribute-value pairs {(name, Alice), (university, ETH), (gradYear, 2008), (hobbies, design & yoga)} (for better visibility, attribute names have been omitted in the figure). Moreover, Node 4 is connected to Node 2 by an edge labeled postedComment and to Nodes 5 and 7 by edges labeled isFriend. □

Note that as N is a set of nodes in the definition above and each node is a set of attribute-value pairs, two nodes cannot have the same set of attribute-value pairs. If it is necessary to represent different nodes that share the same set of attribute-value pairs, we may introduce an additional identifier attribute on each node. The techniques presented in this chapter can be easily adapted to take into consideration this additional attribute, if available.

4.2.2 Query Model

From the start, we assume a basic search service to be available over the dataspace graph. We have discussed our search-and-query language iQL in Chapter 2. For the purposes of this chapter, only a subset of iQL is of relevance, which is summarized in the following definition.

DEFINITION 4.2 (QUERY) *A query Q is an expression that selects a set $Q(G) \subseteq G.N$. The possible query expressions are:*

1. Keyword Expression: *denoted `K`, returns $\{n \in G : (\exists A \in n : n.A \sim K)\}$, i.e., return all nodes in G such that keyword `K` occurs in some attribute-value pair of that node.*
2. Attribute-value Expression: *denoted `A op V`, returns $\{n \in G : n.A \; op \; V\}$, i.e., return all nodes in G such that the value condition on attribute `A` with operator `op` is verified.*
3. Navigation Expression: *denoted `exp/L`, returns $\{n \in G : (\exists n_e \in exp,\ L_e : (n_e, n, L_e) \in E \wedge (L = * \vee L_e = L))\}$, i.e., return all nodes in G that can be reached by an edge labeled `L` from a node in the query expression `exp`. When `L` is the string `*` we return all nodes that can be reached by any edge, regardless of label.*
4. Intersect Expression: *denoted `exp1 exp2`, returns $\{n \in G : n \in exp1 \wedge n \in exp2\}$, i.e., all nodes in G satisfying both `exp1` and `exp2`.*
5. Union Expression: *denoted `exp1 OR exp2`, returns $\{n \in G : n \in exp1 \vee n \in exp2\}$, i.e., all nodes in G satisfying either `exp1` or `exp2`.* □

EXAMPLE 4.4 In the graph of Figure 4.1(a), the following queries exemplify the expression types described in Definition 4.2:

1. `alice` returns Nodes 2 and 4.
2. `university = ETH` returns Nodes 1, 3, and 4.
3. `alice/isFriend` returns Nodes 5 and 7 (Alice's friends).
4. `alice eth` returns Node 4.
5. `alice OR eth` returns Nodes 1, 3, and 4. □

4.2.3 Basic Index Structures

The queries defined in the previous section allow users to pose simple searches on the dataspace. Furthermore, they allow users to navigate among instances of the dataspace through navigation expressions. Those query services are standard in social networking systems, e.g., Facebook [61]. In order to provide efficient processing for those queries, we assume two index structures, commonly found in state-of-the-art search engines [11, 25], to be present:

1. **Inverted Index:** an inverted index records a set of mappings from each keyword to the corresponding list of nodes containing that keyword. In order to support attribute-value as well as keyword expressions, the inverted index may be implemented by concatenating the values of attributes with the attribute name in which those values occur. In this structure, keyword expressions are translated to prefix queries. That is the approach suggested by both Carmel et al. [29] and Dong and Halevy [55]. Note that keywords need not be strings necessarily, but can be of any data type over which a total order can be imposed (e.g., numbers, strings, and dates).
2. **Rowstore:** sometimes refered to as a repository, the rowstore is a mapping from node identifier to the information associated with that node. This information includes the set of attribute-value pairs of the node as well as the edges that connect this node to other nodes in the graph. While we assume that edge information is in-lined in the rowstore, it could also be represented in a separate edge table [63]. This decision is orthogonal to our approach.

Search engines use ranking schemes, e.g., Okapi BM25F [141] or PageRank [25], to support top-K query processing over the data structures desbribed above. In our work, we are agnostic to the underlying ranking scheme used by the search engine. We assume, however, that we are able to obtain the values of scores produced by ranking in order to re-rank results when processing queries with association trails.

4.3 Association Trails

This section formalizes association trails and provides several use cases. In addition, we point out important differences between association trails and the semantic trails of Chapter 3. We conclude the section by discussing how association trails could be obtained and introducing probabilistic association trails.

4.3.1 Basic Form of an Association Trail

Intuitively, an association trail intensionally defines a set of edges between nodes in the dataspace graph. For example, in Figure 4.1(b), a single association trail would define all sharesHobbies edges. Defining more association trails adds more edges to the graph, potentially between the same nodes. Thus, we may interpret each association trail as defining a *graph overlay* on top of the original dataspace graph. In addition, when we take a set of association trails together, they define a multigraph, i.e., a graph in which nodes may be connected by different labeled edges. In Figure 4.1(b), that is the case for nodes 3 and 4. The following definition formalizes this intuition.

DEFINITION 4.3 (ASSOCIATION TRAIL) *A unidirectional association trail is denoted as*

$$A := Q_L \stackrel{\theta(l,r)}{\Longrightarrow} Q_R,$$

where A is a label naming the association trail, Q_L, Q_R are queries, and θ is a predicate. This definition means that the query results selected by $Q_L(G)$ are associated to the query results selected by $Q_R(G)$ according to the predicate θ, which takes as inputs a query result from the left query and a query result from the right query. Thus, we conceptually introduce in the association graph one edge, directed from left to right and labeled A, for each pair of nodes given by $Q_L \bowtie_\theta Q_R$. We require that the node on the left of the edge be different than the node on the right, i.e., no self-edges are allowed. A bidirectional association trail is denoted as

$$A := Q_L \stackrel{\theta(l,r)}{\Longleftrightarrow} Q_R.$$

The latter also means that the query results selected by $Q_R(G)$ are related to the query results selected by $Q_L(G)$ according to θ. □

As described in the definition above, an association trail relates two sets of elements from the data sources by a *join predicate*. Therefore, association trails cover relational and non-relational theta-joins as special cases. Moreover, note that conceptually the θ-predicate above may be an arbitrarily complex function. As a consequence, Definition 4.3 also models use cases such as content equivalence and similar documents. We will present examples of association trails in the next section.

4.3.2 Association Trail Use Cases

In our running example presented in Figure 4.1, we intuitively showed that a social network graph can be transformed into a rich graph of intensional associations by adding association trails. In the use case below, we present the association trails that would make that possible.

USE CASE 4.1 (SOCIAL NETWORKS) The following association trail definitions model the intensional

graph presented in Figure 4.1(b) and Example 4.1:

$$\begin{aligned}
\text{sameUniversity} &:= \texttt{class=person} \overset{\theta_1(l,r)}{\Longleftrightarrow} \texttt{class=person}, \\
&\theta_1(l,r) := (l.university = r.university). \\
\text{graduatedSameYear} &:= \texttt{class=person} \overset{\theta_2(l,r)}{\Longleftrightarrow} \texttt{class=person}, \\
&\theta_2(l,r) := (l.gradYear = r.gradYear). \\
\text{sharesHobbies} &:= \texttt{class=person} \overset{\theta_3(l,r)}{\Longleftrightarrow} \texttt{class=person}, \\
&\theta_3(l,r) := (l.hobbies \cap r.hobbies \neq \emptyset). \\
\text{isFriend} &:= \texttt{class=person} \overset{\theta_4(l,r)}{\Longleftrightarrow} \texttt{class=person}, \\
&\theta_4(l,r) := (\ (l,r,\text{isFriend}) \in E). \\
\text{postedComment} &:= \texttt{class=person} \overset{\theta_5(l,r)}{\Longrightarrow} \texttt{class=comment}, \\
&\theta_5(l,r) := (\ (l,r,\text{postedComment}) \in E).
\end{aligned}$$

The association trail sameUniversity (respectively graduatedSameYear) defines that given any two persons, there will be an edge between them if they have the same value for the university (respectively gradYear) field. Therefore, edges are defined in terms of an equi-join for these two trails. A more complex existential predicate is introduced in the sharesHobbies association trail. This trail defines that two people are related when they have at least one hobby in common in their collection of hobbies. The isFriend association trail represents information given in the original dataspace graph as an association trail. It states that whenever there is an edge between two people labeled isFriend in the original graph, then there should also be an intensional edge between them. A similar effect is obtained by the postedComment association trail. What makes this trail different, however, is that it is unidirectional, i.e., edges always point from people to comments but not the other way around. In addition, the queries on the left and right sides of this trail are also different queries, allowing associations to be defined among elements of different classes.

To sum up, what we have seen in the previous examples is that association trails are able to define a large set of edges among elements in the dataspace graph intensionally. That is achieved in terms of queries that select the elements to be related and a predicate that specifies join semantics among those elements. Association trails are also able to model information already existing in the dataspace graph. When taken together, the association trails above result in a multigraph (displayed in Figure 4.1(b)). This multigraph is actually a view, which can be refined over time by adding more association trails in a pay-as-you-go fashion. □

Although our running example shows how association trails may enrich a social dataspace, it is important to note that association trails are not restricted to that scenario. In fact, we can apply them to other dataspace types, such as personal dataspaces or even web dataspaces. The next use case shows how to improve search over personal information by leveraging the intensional multigraph defined through association trails.

USE CASE 4.2 (PERSONAL INFORMATION) Personal information includes a heterogeneous mix of emails, files-and-folders, contacts, XML, pictures, music, among others. When searching through personal information, one important aspect is providing *context* along with search results. Consider the association trails:

$$\mathsf{sameFileType} := \texttt{class=file} \overset{\theta_6(l,r)}{\Longleftrightarrow} \texttt{class=file},$$
$$\theta_6(l,r) := (l.mimeType = r.mimeType).$$
$$\mathsf{relatedFolder} := \texttt{class=email} \overset{\theta_7(l,r)}{\Longleftrightarrow} \texttt{class=file},$$
$$\theta_7(l,r) := (\exists f_1, f_2 :\ l \in f_1/* \wedge r \in f_2/* \wedge f_1.name = f_2.name).$$
$$\mathsf{similarModificationTime} := \texttt{class=file} \overset{\theta_8(l,r)}{\Longleftrightarrow} \texttt{class=file},$$
$$\theta_8(l,r) := (r.modified - 1 \leq l.modified \leq r.modified + 1).$$

These association trails define several ways in which items in the dataspace are related to one another. The sameFileType association trail relates files that have the same MIME types. The relatedFolder association trail defines that emails are related to files whenever the folders in which they are contained (respectively in the email server and in the filesystem) have the same name. Finally, association trail similarModificationTime defines two files to be related when their modification dates are at most one day apart.

To understand how these association trails could be leveraged by a search application, suppose that you are working on a given section of a paper you have been writing. You start your day by entering in your search application a few keywords from that section. A current search application would only return you the files in which those keywords appear. With association trails, however, rich context would be provided. You would not only see the section you are working on, but also the emails related to the same paper from your email server along with the other files of the same type you have been touching recently. The search application is processing a query relating the results of your search with their neighborhood in the association trails multigraph. These types of queries support a user in the task of exploring a dataspace. We present algorithms for their processing in Sections 4.4 and 4.5. □

4.3.3 Differences Between Association Trails and Semantic Trails

In Chapter 3, we have proposed a technique for pay-as-you-go information integration based on the idea of gradually enriching a search engine with "hints", termed *trails*, that add integration semantics to a dataspace. One question that naturally arises is whether the semantic trails of Chapter 3 can be compared to association trails in terms of their characteristics and expressiveness.

Semantic trails were created to express relationships among *sets* of items in a dataspace. A semantic trail could 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. That would be denoted by a semantic trail as `//projects/PIM` $\longrightarrow$ `//mike/research/PIM` . In contrast, an association trail expresses fine-grained

relationships among individual *items* in a dataspace. As discussed in Use Case 4.1, the association trail sharesHobbies creates several connections in the association-trail multigraph shown in Figure 4.1(b).

It is arguable that each individual connection in the multigraph of Figure 4.1(b) could be represented by a separate semantic trail, leading a set of a single element to another set of a single element. However, that representation defeats the purpose of expressing the multigraph intensionally as done by association trails, given that a large number of semantic trails would be necessary to represent the same graph. This representation with semantic trails is tantamount to extensionally describing the graph.

Moreover, semantic trail query processing is formalized in terms of query containment (for details see Section 3.4). For example, a query for `//projects/PIM//*["Mike"]` would be matched by the semantic trail described above; however, a query for `//*["Mike"]` would not, even though this query might have intersecting instances with `//projects/PIM`. In contrast, a neighborhood query in the association-trail multigraph as the one described in Example 4.2 considers not the containment of queries, but rather the edges in the multigraph that connect different instances (see Section 4.4 for a formalization of neighborhood queries over an association-trail multigraph).

For the reasons discussed above, we conclude that association trails enable an elegant, intensional representation of relationships among instances in a dataspace, while semantic trails are concerned with relationships on a different, set-based level.

4.3.4 Where do Association Trails Come From?

Although there are differences between semantic trails and association trails, some questions raised in the context of semantic trails also make sense in the context of association trails. In Section 3.3.3, we have discussed four principal techniques to come up with semantic trails, which we believe may be (partially) applied in the context of association trails. We discuss these techniques below.

Association trails could be created by users over time as they explore the data and feel the need to define new ways to navigate to related instances in the dataspace. That process could be supported by the first two techniques described in Section 3.3.3, namely: (1) Define trails using a drag-and-drop front end, and (2) Create trails based on user-feedback in the spirit of relevance feedback in search engines [145].

Option (3) of Section 3.3.3 is to mine trails (semi-) automatically from content. While information extraction techniques could be used for the high-level relationships defined by semantic trails, we believe these techniques are not the most appropriate to create the finer-grained relationships specified by association trails. As association trails are defined in terms of join semantics, a possible way to create association trails automatically would be to generalize techniques such as schema matching [136] to produce candidate join-attribute correspondences between two different schemas representing different entities and not only candidate attribute correspondences among two different schemas corresponding to the same entity. In that way, we could find possible attributes that relate instances in the dataspace. Another interesting idea would be to use extensional associations previously defined by users as a way to check whether automatically suggested associations are relevant. For example, if users in a social network have already

defined several instances to be associated by isFriend edges, then we could validate a new association trail by checking how much intersection there is among the edges produced by it and the extensional edges that already exist in the network. A detailed treatment of how to mine association trails from the data in the dataspace is an interesting avenue for future work. Nevertheless, as we have done for semantic trails, we extend our definition of association trails to include uncertainty in the next section.

Option (4) of Section 3.3.3 is to obtain trail definitions from collections offered by third parties or on shared web platforms. We believe this technique should also be applied in the context of association trails. Thus, similarly to the way people share bookmarks today in platforms such as del.i.cio.us, people would exchange both semantic and association trails. Although it is hard to estimate the number of different ways people would create relationships on their data with association trails, we believe that the number of association trails for a given application, e.g., social networks, should range in the hundreds, as there is typically a limited number of meaningful join paths that make sense for a certain scenario.

4.3.5 Probabilistic Association Trails

Analogously to what has been described in Section 3.3.4 for semantic trails, we use probabilistic association trails to model quality for association trails obtained (semi-)automatically. Probabilistic association trails are defined below.

DEFINITION 4.4 (PROBABILISTIC ASSOCIATION TRAIL) *A probabilistic association trail is defined by extending Definition 4.3 to include a probability value p,* $0 \leq p \leq 1$, *for each trail definition. Thus, a probabilistic trail is denoted as*

$$A := Q_L \overset{\theta(l,r)}{\underset{p}{\Longrightarrow}} Q_R$$ □

Probability values for association trails may be obtained when we produce a set of association trails by (semi-) automatically mining them from the contents of the dataspace (see Section 4.3.4). We will show in Section 4.6 how to use the probability value of trails to rank results obtained from query processing over an association-trail multigraph.

4.4 Association-Trails Query Processing

In this section, we show how to process queries over the intensional multigraph defined by association trails. We focus on a special class of queries on top of that multigraph termed *neighborhood queries*. Neighborhood queries are helpful to explore the dataspace. We discuss them in detail in Section 4.4.1. After that, we introduce a canonical query plan to process neighborhood queries on a dataspace graph and discuss its performance characteristics in Section 4.4.2. Based on that analysis, we propose an improved query processing technique for neighborhood queries in Section 4.4.3. For simplicity, we focus our presentation in the following sections on unidirectional association trails, as it is simple to extend our techniques to the bidirectional case.

4.4.1 Neighborhood Queries

Neighborhood queries were used by Dong and Halevy [55] to model exploratory queries on a dataspace graph. The intuition behind neighborhood queries is that queries to the dataspace should also return elements that are in the neighborhood of the query results in the dataspace graph. For example, a neighborhood query for `alice` over the original dataspace graph of Figure 4.1(a) would return not only Alice herself (Node 4), but also people related to Alice (Nodes 5 and 7) and comments posted by Alice (Node 2). In the notation of Section 4.2, a neighborhood query is the union of a query with a navigation expression, e.g., a neighborhood query for `alice` is equivalent to the query `alice` ∪ `alice/*`.

Dong and Halevy [55], similarly to previous work on graph indexing, assume that the dataspace graph is given extensionally. In sharp contrast to the approach of Dong and Halevy [55], we process queries on the multigraph defined by association trails, which is intensional. Therefore, we must revisit the notion of a neighborhood query to define how it could be posed against an intensional graph. To achieve that, we first define what a neighborhood is in our context.

DEFINITION 4.5 (NEIGHBORHOOD) *Given a query Q and a set of association trails A^*, the neighborhood $N^Q_{A^*}$ of Q with respect to A^* is given by*

$$N^Q_{A^*} := \begin{cases} \varnothing & \text{if } A^* := \varnothing \\ (Q \cap A_i^{L.Q}) \rtimes_{\theta_i} A_i^{R.Q} & \text{if } A^* := \{A_i\} \\ N^Q_{\{A_1\}} \cup N^Q_{\{A_2\}} \cup \ldots \cup N^Q_{\{A_n\}} & \text{if } A^* := \{A_1, A_2, \ldots, A_n\}. \end{cases}$$

where $A_i^{L.Q}$ and $A_i^{R.Q}$ denote the queries on the left and right sides of trail A_i, respectively, and θ_i denotes the θ-predicate of trail A_i. □

The definition above states that the neighborhood includes all instances associated through A^* to instances returned by Q. As association trails are defined in terms of join semantics, then finding the related instances entails joining the instances from Q that also appear on $A_i^{L.Q}$ to $A_i^{R.Q}$ and projecting on the instances from the right side. We thus need to compute the semi-join $(Q \cap A_i^{L.Q}) \rtimes_{\theta_i} A_i^{R.Q}$ for each association trail.

In the definition above, we ignore self-edges, which are disallowed by Definition 4.3. However, it is simple to detect self-edges by pre-processing all nodes in $A_i^{L.Q} \cap A_i^{R.Q}$ and marking them if θ_i generates a self-edge. In addition, for some association trails, such as association trails where $A_i^{L.Q} = A_i^{R.Q}$ and θ_i is an equality, it is possible to detect that they generate self-edges by definition and thus always one self-edge should be ignored per element. For the remainder, we will not consider self-edge detection in order to simplify our presentation.

EXAMPLE 4.5 Consider again that we pose a keyword query `alice`, but this time over the association-trail multigraph of Figure 4.1(b). The result for that query is Alice herself (Node 4). The neighborhood of Alice is given by Nodes 1, 2, 3, 5, and 7. For each of these nodes, there is at least one edge in the association-trail multigraph connecting it to Alice. These edges are present in the multigraph because

these nodes join with Node 4 by the corresponding predicates described in Use Case 4.1. For example, Anna (Node 1) joins with Alice (Node 4) because both have the same value for the university field. □

Now we are ready to introduce the new definition of a neighborhood query over our intensional graph. Once again in contrast to the approach of Dong and Halevy [55], we define neighborhood queries not only for keyword queries, but for any query Q posed to the dataspace.

DEFINITION 4.6 (NEIGHBORHOOD QUERY) *Given a query Q and a set of association trails A^*, the neighborhood query $\overset{\leadsto}{Q}_{A^*}$ is given by*

$$\overset{\leadsto}{Q}_{A^*} := Q \cup N^Q_{A^*}$$

The results obtained by Q are termed primary query results *and the results obtained by $N^Q_{A^*}$ are termed* neighborhood results. □

The definition states that a neighborhood query returns all primary query results along with all neighborhood results to which they have at least one edge in the association-trail multigraph.

EXAMPLE 4.6 If $A^* = \{$sameUniversity, graduatedSameYear, sharesHobbies, isFriend, postedComment$\}$ is the set of association trails defined in Use Case 4.1, then in Figure 4.1(b), a neighborhood query $\overset{\leadsto}{Q}_{A^*} :=$ `alice` would return as a primary query result Alice herself (Node 4) and as neighborhood results all nodes in her neighborhood (Nodes 1, 2, 3, 5, and 7). □

Note that a given neighborhood result may be connected to a primary query result by more than one edge. This situation happens when the instance is connected to the primary query result by several different trails (Nodes 3 and 4 in Figure 4.1(b)). Intuitively, we may think that this neighborhood result is more related to the primary query result than an instance that is connected to it by a single edge (or by a smaller number of edges). Therefore, it is useful to compute the number of edges in the association-trail multigraph that connect a given instance to the primary query results. Fortunately, it is easy to see from Definition 4.5 that this information could be obtained by aggregation when processing the unions among the neighborhoods of different association trails. This information will be exploited in the ranking scheme presented in Section 4.6.

4.4.2 Canonical Plan

The canonical plan to process neighborhood queries follows directly from Definitions 4.5 and 4.6. Figure 4.2 shows its general form for a neighborhood query $\overset{\leadsto}{Q}_{A^*}$, where $A^* = \{A_1, \dots, A_n\}$.

The canonical plan first obtains all items from the query that also appear on the left side of an association trail ($Q \cap A_i^{L.Q}$). These items are then semi-joined with the right side of the association trail $A_i^{R.Q}$ in order to obtain the elements from the right side that are connected to the elements from the query. In other words, we compute the neighborhood $N^Q_{A^*}$ for all trails. Afterwards, we take the union of the original query Q with the neighborhood $N^Q_{A^*}$.

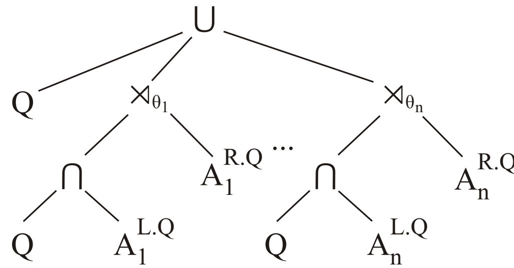


Figure 4.2: Canonical query plan to process neighborhood queries

An important feature of the canonical plan is that it computes all elements in the neighborhood of the query results without ever computing the whole association-trail multigraph. The association-trail multigraph can be fully materialized by computing the joins among left and right sides of all association trails: $A_1^{L.Q} \bowtie_{\theta_1} A_1^{R.Q} \cup \ldots \cup A_n^{L.Q} \bowtie_{\theta_n} A_n^{R.Q}$. Each join may result in $O(N^2)$ edges, where N is the number of nodes selected by the association trail queries. Each semi-join computed in the canonical plan, on the other hand, has complexity $O(N)$. Thus, the canonical plan can bring significant computational savings when compared to the naive strategy of materializing the multigraph. One could argue that the computation of the multigraph could be performed at indexing time, however, resulting in query-time performance superior to that of the canonical plan. We will explore that option in Section 4.5. For now, let us focus on optimizing and modeling the performance of the canonical plan.

Optimizing the Canonical Plan. As a first step, a set of simple optimizations could be applied to the canonical plan. For example, the query processor may detect that $Q \cap A_i^{L.Q} = \varnothing$ [113]. Another simple optimization is to remove the intersection when $A_i^{L.Q} = G.N$.

In the general case, however, we are left with a query plan as shown in Figure 4.2. The plan must compute the query Q one additional time for each association trail. In addition, it always computes the queries on left and right sides of the association trails. As we have illustrated in Use Cases 4.1 and 4.2, we frequently find common expressions being used for these queries. As a consequence, the canonical plan will naturally contain a large number of common subexpressions.

Techniques to reuse common subexpressions include spooling and using push-based query plans [125]. Spooling is simple to be implemented in pull-based query processors, such as search engines and database systems. On the other hand, it incurs in higher runtime overhead than push-based plans. As the investment to convert a pull-based query processor into push-based one is many times tantamount to a reimplementation of the query processing stack, we would like to use pull-based plans while reducing the overhead of having to materialize and re-read results from common subexpressions. We present a solution to this problem in Section 4.4.3. In the following, we model the read performance of the canonical plan when spooling is assumed.

Predicting Canonical Plan Read Performance. Assume without loss of generality that all queries in the association trail definitions are common expressions[1], i.e., $A_1^{L.Q} = A_1^{R.Q} = \ldots = A_n^{L.Q} = A_n^{R.Q}$. Then

[1]If that is not true, the model can be applied piecewise to subsets of common expressions in the plan.

a spooling query processor executes the canonical plan of Figure 4.2 in two basic phases: (1) the materialization phase, in which common subexpressions are computed and materialized (hopefully in main memory), and (2) the scan phase, in which these common subexpressions are scanned as input to the other operators (especially semi-joins) in the plan.

The cost for the materialization phase C_{MAT} is given by the cost to compute the common subexpressions Q, $A_*^{L.Q} = A_*^{R.Q}$, and $Q \cap A_*^{L.Q}$, and transfer their data to a materialization (either in main-memory or disk). In the data structures of Section 4.2.3, processing a query means scanning the appropriate inverted lists and fetching attributes necessary for the semi-joins from the rowstore. When query result lists are above a certain size, these costly fetches dominate query processing time. Say Tr_{fetch} is the rate at which we can fetch data from the rowstore, Sel_Q is the selectivity of a query Q, N is the number of nodes selected by the association trail queries, S_n is the average node size, S_{OID} is the length of a node identifier, and Tr_{spool} is the rate at which we can read data from the spool. Then we can estimate C_{MAT} by:

$$C_{MAT} = Sel_Q \cdot N \cdot S_n \cdot Tr_{fetch} + Sel_{A_*^{L.Q}} \cdot N \cdot S_n \cdot Tr_{fetch} + (Sel_Q \cdot N \cdot S_{OID} \cdot Tr_{spool} + Sel_{A_*^{L.Q}} \cdot N \cdot S_n \cdot Tr_{spool})$$

The cost for the scan phase C_{SCAN} is given by the cost necessary to scan the spool for Q one time plus the cost to scan the spools for $Q \cap A_*^{L.Q}$ and $A_*^{R.Q} = A_*^{L.Q}$ n times. We estimate the selectivity of the expression $Q \cap A_*^{L.Q}$ to be $min(Sel_Q, Sel_{A_*^{L.Q}})$. Thus,

$$C_{SCAN} = Sel_Q \cdot N \cdot S_{OID} \cdot Tr_{spool} + \big(min(Sel_Q, Sel_{A_*^{L.Q}}) \cdot N \cdot S_{OID} \cdot Tr_{spool} + Sel_{A_*^{R.Q}} \cdot N \cdot S_n \cdot Tr_{spool}\big) \cdot n$$

As we may see from this analysis, for $n > 0$, there is a constant start-up cost for the canonical plan equal to $C_{MAT} + Sel_Q \cdot N \cdot S_{OID} \cdot Tr_{spool}$. Apart from that cost, the cost of reading the data for the canonical plan increases linearly in the number of trails. We expect the cost to read the data to be a dominating factor on the canonical plan's performance.

4.4.3 N-Semi-Joins

As we have argued in the previous section, the costs to execute the canonical plan with spooling will be dominated by the cost to re-read the spool. In this section, we propose an intermediate solution in-between spooling and push-based plans.

Instead of converting the whole algebra to be push-based, we replace the union of semi-joins in the canonical plan by one single, more powerful, pull-based operator: an n-semi-join. This operator is pa-

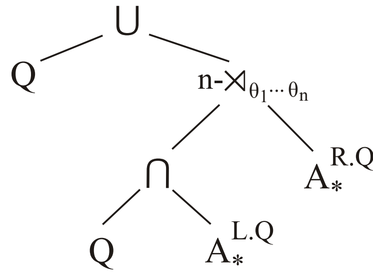


Figure 4.3: N-semi-join query plan to process neighborhood queries

rameterized with a list of θ-predicates $\Theta := (\theta_1, \ldots, \theta_n)$ and takes only two inputs. The core idea of the n-semi-join rewrite is to compress all semi-joins that have the same left and right inputs into one single n-semi-join. For example, if $A_1^{L.Q} = A_1^{R.Q} = \ldots = A_n^{L.Q} = A_n^{R.Q}$, then the canonical plan of Figure 4.2 is rewritten to the n-semi-join plan of Figure 4.3. If all left and right sides are different, however, then the n-semi-join plan is equivalent to the canonical plan of Figure 4.2.

Algorithm 2: Right-preserving n-semi-join

Input: Left Input I_L
Right Input I_R
List of equi-join θ-predicates $\Theta := (\theta_1, \ldots, \theta_n)$
Output: Result Set $Res \subseteq I_R$

1 Create hash tables $H_1, \ldots, H_n$
2 // hash left input for each predicate along with in-degree counts:
3 **for** *Node* $n_L \in I_L$ **do**
4   **for** $i = 1, \ldots, n$ **do**
5         $v_i := n_L.a_j$, where a_j is left-input attribute in θ_i
6   **if** $\exists\, v_i$ **then**
7             $inDegree := H_i.\text{lookup}(v_i)$
8             $H_i.\text{put}(v_i \rightarrow inDegree + 1)$
9   **end**
10   **end**
11 **end**
12 // probe based on right elements:
13 **for** *Node* $n_R \in I_R$ **do**
14   **for** $i = 1, \ldots, n$ **do**
15         $v_i := n_R.a_j$, where a_j is right-input attribute in θ_i
16   **if** $\exists\, v_i \wedge H_i.contains(v_i)$ **then**
17             $n_R.inDegree := n_R.inDegree + H_i.\text{lookup}(v_i)$
18   **end**
19   **end**
20   **if** $n_R.inDegree > 0$ **then**
21         $Res := Res \cup \{n_R\}$
22   **end**
23 **end**

In this respect, n-semi-join plans are "safe", in the sense that they will not produce a plan that is worse than the canonical plan. In practice, we expect n-semi-joins to significantly compress query plans, replacing the n semi-joins in the canonical plan by $k \ll n$ n-semi-joins. In the example given in Use Case 4.1, the canonical plan would have 5 semi-joins; the equivalent n-semi-join plan would have only 2 n-semi-joins. If another hundred association trails were defined relating people as in the first four association trails of that use case, we would still have only 2 n-semi-joins, compared to 105 semi-joins for the canonical plan.

We present a hash-based n-semi-join algorithm in Algorithm 2. It operates in a manner similar to common hash-based semi-joins. In the hash phase, we read the left input and hash the appropriate values for each θ_i predicate in the corresponding i-th hash table. During the probe phase, we read each tuple from the right input and probe all hash tables. In the algorithm, we keep an in-degree count of how many edges from the left input are pointing to each element in the right input. This information is important for ranking in the association-trail multigraph. If that information would not be necessary, however, we could stop probing as soon as the first match is found on one of the hash tables.

A hash-based n-semi-join is able to handle all association trails presented in Use Cases 4.1 and 4.2, but would have to be adapted to handle existential semantics, as in sharesHobbies, and also the band join semantics in similarChangeTime (e.g. [151]). Thus, for certain θ-predicates we may have to use different join techniques, even not only hash-based joins. We refer the reader to the significant work available on join processing in the literature [96, 121]) for a discussion of how to handle different types of θ-join efficiently.

4.5 Association-Trails Indexing

Although in principle the joins specified by association trails may be processed at query time, this may turn out to be infeasible in a real system with hundreds of trails. In this section, we propose index structures to accelerate the processing of queries on the association-trail multigraph.

We will discuss in the following several possible strategies to provide indexing support for neighborhood queries with association trails. First, in order to speed-up the canonical and n-semi-join plans discussed in Section 4.4, we present how to keep materialized copies of $A_*^{L.Q}$ and $A_*^{R.Q}$ for every distinct query expression appearing in the left or right side of an association trail. Second, we show the option of pre-computing and materializing all the edges in the association-trail multigraph by storing the join $A_1^{L.Q} \bowtie_{\theta_1} A_1^{R.Q} \cup \ldots \cup A_n^{L.Q} \bowtie_{\theta_n} A_n^{R.Q}$. Third, we discuss how to take advantage of join grouping properties to create a highly compressed index structure. Fourth, we argue that for attribute-value and keyword expressions, we may precompute neighborhood query results for all possible queries in the system's vocabulary and store the results in a single structure. Fifth, motivated by the possibly high cost of the latter solution, we propose a hybrid approach in which we materialize subsets of the vocabulary and answer the remaining queries based on the best of the other available structures. All of the indexing strategies proposed are built on top of standard index structures, such as inverted lists, materialized views, and B+-trees, and are thus easy to integrate into state-of-the-art systems.

4.5.1 Materialize $A_*^{L.Q}$ and $A_*^{R.Q}$

Recall from Section 4.4.3 that even the improved n-semi-join plan of Figure 4.3 must execute the original user query Q as well as the left ($A_*^{L.Q}$) and right ($A_*^{R.Q}$) sides of the association trails. One important observation is that although the query Q is not directly available to us at indexing time, the association trails are. Thus, we may pre-compute $A_*^{L.Q}$ and $A_*^{R.Q}$ and save the results as materialized views.

Materialize $A_*^{L.Q}$ and $A_*^{R.Q}$ with an OID List. The materialized views for $A_*^{L.Q}$ and $A_*^{R.Q}$ may be physically represented as lists of node identifiers (OIDs) and pre-fetched attributes necessary at query time. If $A_*^{L.Q}$ and $A_*^{R.Q}$ are executed by lookups in an inverted index such as the one described in Section 4.2.3, they will produce result lists containing the node identifiers of all matching nodes. These result lists, however, do not contain the necessary attributes for each node that are required to process the θ-predicates in the subsequent n-semi-joins in the plan. As a consequence, these additional attributes must be fetched from the rowstore at query time, resulting in one random access per node identifier. Thus, the costs to execute $A_*^{L.Q}$ and $A_*^{R.Q}$ using the typical data structures described in Section 4.2.3 may be high, resulting in significant query-time gains from pre-computation. As a result, the materialized views created for $A_*^{L.Q}$ and $A_*^{R.Q}$ at indexing time should contain all necessary attributes to avoid fetches at query time, i.e., these materialized views comprise a projection of all attributes used in the association trails' predicates.

OID	university	gradYear
1	ETH	2007
3	ETH	2008
4	ETH	2008
5	PUC-Rio	1998
7	Cornell	2000
8	Cornell	1998

Figure 4.4: Materialization of $A_*^{L.Q}$ and $A_*^{R.Q}$ with an OID List for association trails sameUniversity and graduatedSameYear. The materialization keeps a projection of the attributes referenced in the association trails' predicates.

Figure 4.4 shows the single materialized view that should be created to support the first two trails in Use Case 4.1. A simple optimization, included in the figure, is to only record nodes that participate in the n-semi-join at query time, i.e., materialize $A_*^{L.Q}$ $n\text{-}\ltimes_{\theta_1,\ldots,\theta_n}$ $A_*^{R.Q}$ and $A_*^{R.Q}$ $n\text{-}\ltimes_{\theta_1,\ldots,\theta_n}$ $A_*^{L.Q}$, respectively. This strategy implies additional indexing time to compute the n-semi-joins, but may bring savings when scanning the materialization at query time if many of the tuples do not participate in the join.

Materialize $A_*^{L.Q}$ with a B+-tree and $A_*^{R.Q}$ with an OID List. Another optimization that may be carried out when materializing $A_*^{L.Q}$ and $A_*^{R.Q}$ is to change the physical structure used to materialize $A_*^{L.Q}$ from an OID list to a B+-tree. In the query plan of Figure 4.3, $A_*^{L.Q}$ is only used in the expression $Q \cap A_*^{L.Q}$. If $A_*^{L.Q}$ is stored as an OID list, the intersection may be processed either by a hash- or merge-based approach. On the other hand, if we index $A_*^{L.Q}$ on OID, then the intersection may be processed using indexed nested loops. When the selectivity of the query is high enough, index nested loops provides better execution times than alternative join approaches [46]. In Figure 4.4, materialization of $A_*^{L.Q}$ as a

B+-tree is equivalent to creating a copy of the materialized view in the figure organized as a clustered index on OID.

One interesting observation is that if we again consider the data structures described in Section 4.2.3, then the rowstore itself may be used to represent the indexed version $A_*^{L.Q}$ at no additional storage cost. As the rowstore provides a mapping from node identifier to the information associated with the node, it is reasonable to assume that this mapping would be efficiently implemented itself as a B+-tree. The downside of this approach is that many attributes that are not related to the trail joins may have to be read from this structure when looking up the OIDs from Q. This trade-off is similar to the one encountered in row versus column stores [91].

4.5.2 Materialize $A_1^{L.Q} \bowtie_{\theta_1} A_1^{R.Q} \cup \ldots \cup A_n^{L.Q} \bowtie_{\theta_n} A_n^{R.Q}$

A natural indexing strategy is to compute all edges in the association-trail multigraph and explicitly materialize those edges. As noted previously, this strategy is equivalent to materializing the join $A_1^{L.Q} \bowtie_{\theta_1} A_1^{R.Q} \cup \ldots \cup A_n^{L.Q} \bowtie_{\theta_n} A_n^{R.Q}$. At query time, we can take advantage of the join materialization by executing the query plan shown in Figure 4.5. It takes all node identifiers from the query and semi-joins them with the join materialization to find all edges which have the query node as left side. It then projects the edges on the right sides to obtain the neighborhoods of each node. The neighborhoods are unioned with the original query results, as specified in Definition 4.6. Note that to compute the union of joins $A_1^{L.Q} \bowtie_{\theta_1} A_1^{R.Q} \cup \ldots \cup A_n^{L.Q} \bowtie_{\theta_n} A_n^{R.Q}$, it may be profitable to apply the same idea discussed in Section 4.4.3 in order to compress that union to a smaller union of n-joins. We take that approach in our implementation.

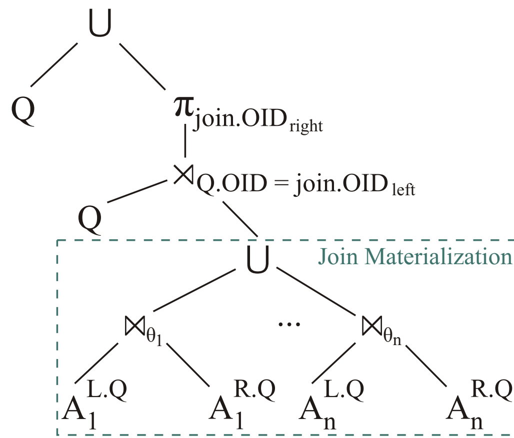
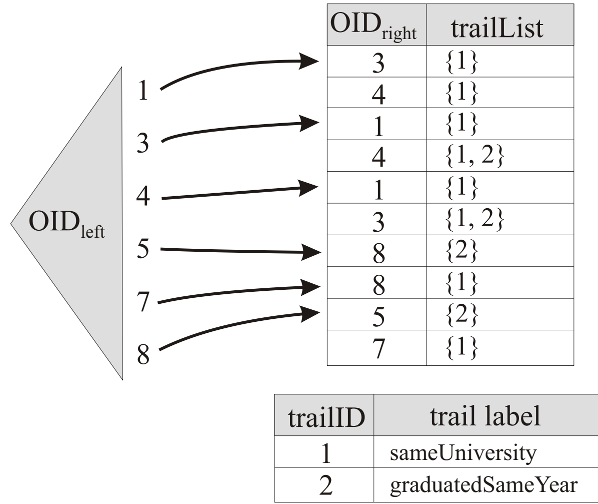


Figure 4.5: Join materialization query plan to process neighborhood queries

Materialize $A_1^{L.Q} \bowtie_{\theta_1} A_1^{R.Q} \cup \ldots \cup A_n^{L.Q} \bowtie_{\theta_n} A_n^{R.Q}$ with an Inverted List. The association trail joins could be materialized with a join index [160]. Typically, join indices are implemented as clustered B+-trees, according to the need to access the index from left to right or from right to left. In our scenario, only access from left to right is necessary, leading to the use of a single clustered B+-tree with the left OIDs as a key. This representation still has some unnecessary overhead. If a given OID in the left maps

to several OIDs in the right, then the join index will repeat the left key several times, one time in each $\langle OID_{left}, OID_{right}, trailList \rangle$ tuple.



OID_right	trailList
3	{1}
4	{1}
1	{1}
4	{1, 2}
1	{1}
3	{1, 2}
8	{2}
8	{1}
5	{2}
7	{1}

trailID	trail label
1	sameUniversity
2	graduatedSameYear

Figure 4.6: Materialization of association-trail joins with an inverted list for association trails sameUniversity and graduatedSameYear

By employing an inverted list, we factor out each occurrence of a left OID, representing it only once for all right matching OIDs. Figure 4.6 shows the resulting structure for the first two trails in Use Case 4.1. Note that extending an inverted list to keep a trail list per right OID is straightforward. That information may be interesting when not all trails are being applied to the query (e.g., when one trail is marked as removed or disabled). When we have a scenario where it is known that all trails will be applied, the trail list could be aggregated into an edge count.

Materializing the join may yield significant query-time benefits. However, obtaining these benefits can have a high indexing-time cost, especially if the join has skew. This high-indexing cost is due to the need to materialize a large number of edges to represent the association-trail multigraph extensionally. We provide a simple argument below to give a sense of the number of edges produced by an association trail with an equality θ-predicate.

Estimating the size of the join materialization. Suppose an association trail $A := A^{L.Q} \stackrel{\theta(l,r)}{\Longrightarrow} A^{R.Q}$ where $\theta(l,r) := (l.a_j = r.a_k)$. Suppose further that attributes a_j and a_k follow the same uniform distribution with cardinality *card*. If each of $A^{L.Q}$ and $A^{R.Q}$ select N nodes from the dataspace graph G, then we expect the number of edges $E_{\{A\}}$ generated by A to be:

$$E_{\{A\}} := card \cdot \left(\frac{N}{card}\right)^2 = \frac{N^2}{card}$$

In other words, the N nodes are partitioned into *card* groups with the same value for the join attribute. In each group, all nodes join one another, i.e., each group forms a clique in the graph. As association trails generate graph overlays independently, the argument above may be extended to a set of trails $A^* :=$

$\{A_1, A_2, \ldots, A_n\}$ to obtain the number of edges E_{A^*} in the association-trail multigraph defined by A^*:

$$E_{A^*} := \sum_{i=1}^{n} \frac{N_i^2}{card_i}$$

What this argument tells us is that when $card_i$ is not comparable to N_i, association trails with an equality θ-predicate generate graphs with a quadratic number of edges. This result comes as no surprise, given the grouping nature of the underlying equi-join being computed. This effect should be even more pronounced when attribute distributions are skewed. We can see a reduced example of the effect in Figure 4.1(b), where Nodes 1, 3, and 4 form a clique when we consider the sameUniversity association trail.

4.5.3 Grouping-compressed Index

In the previous section, we have seen that the size of a join materialization may grow significantly as a function of the number of nodes selected by the association trails. This growth is related to grouping effects in the join. In an equi-join, for example, all nodes with the same value for a join key will join with one another, generating a clique in the graph for that association trail. As a consequence, in the join materialization we would have to represent all the edges connecting all the nodes to each other. In a clique of size C, we would naively store C^2 edges in the join materialization.

Compressing the Join Inverted List with Lookup Entries. As an alternative representation, we could: (1) explicitly represent the edges from a given node n_1 in the clique to all the other nodes $\{n_2, \ldots, n_C\}$ and (2) for each remaining node in the clique, represent only special *lookup edges* $\langle n_i, n_1, lookup \rangle$ that state n_i connects to the same nodes as n_1. In that alternative representation, we would represent the information in the clique with C normal edges for n_1's connections plus $C-1$ edges for the lookup edges of all remaining nodes. In short, a reduction in storage space (and consequently join indexing time) from C^2 edges to $2 \cdot C - 1$ edges. We call this more efficient representation *grouping compression*, given that we are representing information that a group of nodes is interconnected with space linear in the number of nodes.

The grouping compression may be applied not only to equi-join predicates in the association trails, but also for other types of joins in which edges outgoing from one node n_i point to exactly the same nodes as the edges outgoing from another node n_j. In such situations, we may compress the edges of n_j by emitting the lookup edge $\langle n_j, n_i, lookup \rangle$. For example, in the existential predicate of association trail sharesHobbies in Use Case 4.1, whenever two nodes have the exact same set of hobbies, they must point to the exact same set of nodes. So that predicate is another example of a predicate where grouping effects on the join exist and, therefore, grouping compression may be used.

Clearly, we would like to represent the compressed edges from a set of association trails in the same grouping-compressed join materialization. Note, however, that one must take care for having the same node n_i at times being reported in normal edges and at times being reported in lookup edges. That situation may happen if n_i has edges pertaining to distinct association trails. Because of that, we extend

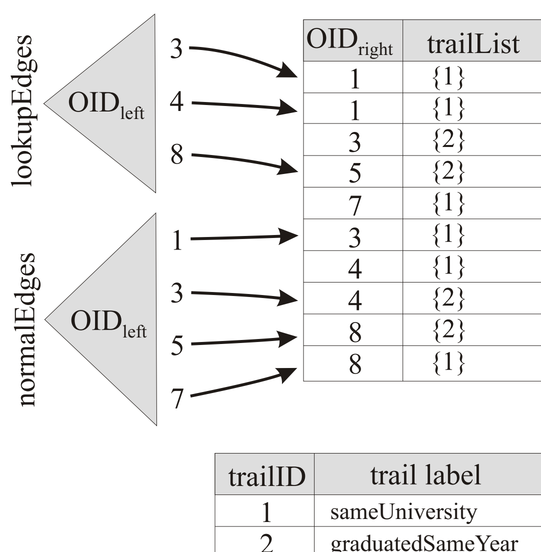


Figure 4.7: Grouping-compressed materialization of association trail joins for association trails sameUniversity and graduatedSameYear

the inverted list representation used for a join materialization to include two lists for each node n_i. One list contains all normal edges for that node, while the other contains all lookup edges. The extended structure may be seen in Figure 4.7. Note that in the figure we show two directories, but the same logical partitioning could be achieved with key concatenation and the use of a single sorted directory, e.g., a B+-tree. In that way, we can reuse generic code written for an inverted list data structure to materialize also the grouping-compressed index. That is the approach we take in our implementation.

To understand how we have mapped normal and lookup edges to the data structure shown in Figure 4.7, consider for example the edges in which the left node is Node 3. Node 3 has a lookup list in which the lookup edge $\langle 3, 1, lookup \rangle$ for association trail sameUniversity is represented. This edge signals that if we wish to know which nodes are connected to Node 3, then we should include Node 1 along with nodes in the normal list of Node 1. Thus, Node 3 is connected to Node 1 and to Node 4 by that association trail (note that we ignore self-edges). In addition, the normal list of Node 3 includes Node 4 for association trail graduatedSameYear. Therefore, Node 3 is additionally connected to Node 4 by this other association trail.

Processing Mini-n-semi-joins on the Grouping-compressed Index. In the structure of Figure 4.7, we could use a two-step naive strategy to perform query processing. In the first step, we would look up the normal edges for each OID in the primary query results and report them. In the second step, we would then process the lookup list for each OID. Processing a lookup list involves triggering one additional lookup in the normal list for every element in the lookup list. In addition, that lookup in the normal list should be filtered according to the trails recorded in the lookup list entry. For example, processing the lookup list of Node 4 in Figure 4.7 entails performing two additional lookups, one for Node 1 and another for Node 3. In the lookup of the normal list of Node 1, we must consider only entries for the association trail sameUniversity; similarly, in the lookup of the normal list of Node 3, we must consider entries for

association trail graduatedSameYear. In the example, all entries in those lists will be returned by the lookups. If also we need to process the lookup list of Node 3, then that processing would generate an additional lookup of the normal list of Node 1.

Algorithm 3: Mini-n-semi-join Algorithm

Input: Primary Query results $Q(G)$
Grouping-compressed index GCI

Output: Result Set $Res = \overset{\leadsto}{Q}_{A^*}$

1 Create Map *oidToTrailCountMap* of type $OID \rightarrow \{(trail_1, count_1), \ldots, (trail_n, count_n)\}$
2 // build phase — build *oidToTrailCountMap* from lookup edges:
3 **for** *Node* $n_i \in Q(G)$ **do**
4 **for** *Entry* $entry_{Lookup} \in GCI.lookupList(n_i.OID)$ **do**
5 TrailCountList *trailCountList* := *oidToTrailCountMap*.getOrCreate($entry_{Lookup}.OID$)
6 *trailCountList*.increaseTrailCounts($entry_{Lookup}.trailList$)
7 **end**
8 **end**
9 $Res := (Q(G) \cup$ getNodes(*oidToTrailCountMap*.keySet()))
10 // probe phase — scan normal lists to expand *Res*:
11 **for** *Node* $n_i \in Res$ **do**
12 $n_i.inDegree := n_i.inDegree + \sum$ all counts in list *oidToTrailCountMap*.get($n_i.OID$)
13 **for** *Entry* $entry_{Normal} \in GCI.normalList(n_i.OID)$ **do**
14 int *entryInDegree* := 0
15 // count all trails when access caused by primary query result:
16 **if** $n_i \in Q(G)$ **then**
17 $entryInDegree := entry_{Normal}.trailList.length$
18 **end**
19 // count extra lookup edges for $entry_{Normal}.OID$:
20 **if** $oidToTrailCountMap.containsKey(n_i.OID)$ **then**
21 $entryInDegree := entryInDegree + \sum$ all counts in list intersectTrailCounts(*oidToTrailCountMap*.get($n_i.OID$), $entry_{Normal}.trailList$), except self-edges when $entry_{Normal}.OID \in Q(G)$.getOIDs()
22 **end**
23 // add $entry_{Normal}.OID$ to result if there is some edge to it:
24 **if** $entryInDegree > 0$ **then**
25 Node $n_{Normal} := Res$.getOrCreate($entry_{Normal}.OID$)
26 $n_{Normal}.inDegree := n_{Normal}.inDegree + entryInDegree$
27 **end**
28 **end**
29 **end**

The strategy above is in effect creating all the edges in the graph for the given OIDs. These edges can then be used in the same way described for the query plan of Figure 4.5. This approach, however, ignores one important intuition: query processing and graph view processing can be combined, analogously to view merging in traditional query rewriting [134].

Alternatively, we could push down the semi-join processing necessary to compute neighborhoods directly to the index lookups. A hash-based algorithm is presented in Algorithm 3. The algorithm is described assuming that all trails in A^* should be taken into consideration for query processing. Filtering for a given set subset of the trails, however, is straightforward.

First, we begin by processing lookup lists (Lines 1 to 8). For every node in $Q(G)$, we check the lookup list for that node (Line 4). For each element in the lookup list, we record the trails in which that element appears along with the number of times it appears per trail (Lines 5 and 6). These steps make up the *build* phase, in which for each element in the lookup lists, we have built a small mapping containing in-degree information for each association trail (map *oidToTrailCountMap*). This logic is very similar to the one used in n-semi-joins, except that the mappings are restricted to single OIDs. The next phase is called the *probe* phase, as it consists of probing the normal lists of all nodes either in $Q(G)$ or *oidToTrailCountMap* in order to expand the result (Lines 9 to 29). We first update the in-degree of each node according to its lookup edges (Line 12). For each element in the normal list of that node (Line 13), we calculate its appropriate in-degree, depending on whether the lookup was triggered by a primary query result (Lines 15 to 18) or by an element from a lookup list (Lines 19 to 22). Note that if the element appears in a lookup list, we must increase its in-degree only for association trails in the intersection of its trail list and the trails in *oidToTrailCountMap* (function intersectTrailCounts, Line 21). Finally, if the element is indeed connected through some edge, then we add it to the result, updating its in-degree (Lines 23 to 27).

EXAMPLE 4.7 (MINI-N-SEMI-JOIN ALGORITHM) Suppose $Q(G) := \{3,4\}$ and the index of Figure 4.7 are given as input to Algorithm 3. In the build phase, the algorithm will inspect the lookup lists for Nodes 3 and 4. It will find Node 1 represented twice for sameUniversity and Node 3 represented once for graduatedSameYear. That information will be saved in the map *oidToTrailCountMap* $:= \{(1 \rightarrow \{(1,2)\}), (3 \rightarrow \{(2,1)\}\}$. The probe phase then calculates which nodes are related to one of Nodes 1, 3, or 4 ($Q(G) \cup \text{getNodes}(\textit{oidToTrailCountMap}.\text{keySet}())$). We first process Node 1, adding it to the result with in-degree equal to 2 (Line 12). We then lookup the normal list for Node 1. Both entries, again Nodes 3 and 4, qualify for association trail sameUniversity and are thus added to the result with a partial in-degree of 1 (Line 21; note that we detect self-edges). We proceed by processing Node 3. As Node 3 is in $oidToTrailCountMap$ with an edge count of 1, we increase its result in-degree to 2 (Line 12). We then process the normal list of Node 3, updating the in-degree of Node 4 to 2 (Line 17) as Node 3 is also in $Q(G)$. Note that the count of Node 4 is not increased again in Line 21, as we detect self-edges. Finally, we try to lookup the normal list of Node 4, but it is inexistent. The final result contains the nodes and in-degrees $Res := \{(1,2),(3,2),(4,2)\}$, i.e., Nodes 1, 3, and 4 are all returned with in-degree 2. Contrast that result with Figure 4.1. If Nodes 3 and 4 are in the original query result, then Node 1 is in the neighborhood of both of them via association trail sameUniversity. Nodes 3 and 4 are in the neighborhood of one another and are connected by both association trails sameUniversity and graduatedSameYear. □

Note that using a grouping-compressed index may still be more expensive at query time than a simple uncompressed join materialization because of the larger number of necessary random lookups to the index structure. On the other hand, indexing-time savings may be significant.

4.5.4 Query Materialization and Hybrid Approach

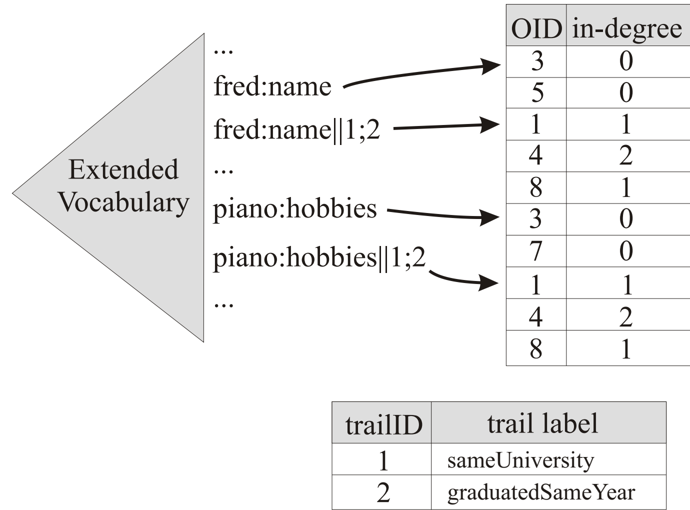
In the previous sections, we have presented several techniques to pose neighborhood queries over an association-trail multigraph. What all of these techniques have in common is that they assume that the query posed by the end-user is *not* known at indexing time. Therefore, only materialization of information given in the association trails is exploited. However, traditional text indexing with inverted lists actually assumes that queries are *partially* known. In particular, we know that users can only select keyword and attribute-value expressions out of the vocabulary from the dataset. To extend that intuition to neighborhood queries, one could index for every possible entry in the vocabulary not only the primary query results associated to it, but also all neighborhood results. Building such extended indexes over extensional graphs has been discussed by Carmel et al. [29] and Dong and Halevy [55]. However, it remains unclear how this technique could be used in our context of intensional multigraphs. In the following, we will show how to incorporate this style of query materialization in our approach and how to couple it with the materialization strategies presented previously to create a hybrid query-processing approach.

Representing Neighborhoods for the Vocabulary with an Inverted List. We call an entry in the vocabulary a *token*. In the data structures of Section 4.2.3, a token corresponds to one attribute-value pair. The first problem we should address is how to compute not only primary query results but also neighborhood results at indexing time. Basically, for every token in the vocabulary, we must handle it as a *query* being submitted to the system and compute its primary query results and its neighborhood results. If we have already built an inverted index as in Section 4.2.3, obtaining the primary query results is straightforward. Obtaining the neighborhood results over the association-trail multigraph is tantamount to answering a query for that token with one of the strategies discussed in Sections 4.4 to 4.5.3. The query-processing strategy should be chosen taking into consideration its *total processing time*. Thus, strategies may perform a pre-indexing step in order to accelerate the processing of the queries for all vocabulary tokens.

EXAMPLE 4.8 Suppose we have a vocabulary with 200,000 tokens. Suppose further that a non-indexed strategy computes results to every query on average in 1 minute, and an indexing strategy takes 2 hours to pre-index, but reduces query-processing time to half a minute per query on average. Then the total processing time for the non-indexed strategy would be 200,000 min = 3,334 h, while the indexing strategy would require 120 min + 200,000×0.5 min = 1669 h. □

The second problem to be tackled is how to represent in a single inverted list structure both the primary query results and the neighborhood results. A simple approach here, discussed by both Carmel et al. [29] and Dong and Halevy [55], is to create new tokens $t'_1, \dots, t'_n$ for every token t in the original vocabulary. A token t'_i is formed by the concatenation of t with a suffix derived from the context being recorded. For example, we could have a suffix s_{A_*} that represents all association trails in the system. The new token $t||s_{A^*}$ would contain the neighborhood results $N^t_{A^*}$. The same could be done for individual association trails in the system, if we would wish to expand neighborhoods for only a subset of the association trails.

Figure 4.8 shows the index structure described above for $A^* = \{\text{sameUniversity, graduatedSameYear}\}$. We call this index structure a *query-materialization index*. The figure shows the tokens `fred:name` and



trailID	trail label
1	sameUniversity
2	graduatedSameYear

Figure 4.8: Query materialization index for association trails sameUniversity and graduatedSameYear. Only tokens `fred:name` and `piano:hobbies` are displayed.

`piano:hobbies` in the index, i.e., keyword `fred` in the `name` attribute of a node and respectively keyword `piano` in the `hobbies` attribute. The suffix `||1;2` is concatenated to each token to create a new token recording the corresponding neighborhood for both association trails sameUniversity and graduatedSameYear. For example, querying for $Q :=$ `name=fred` will select Nodes 3 and 5, as shown in the postings for token `fred:name`. The respective neighborhood query $\overset{\leadsto}{Q}_{A^*}$ obtains the union of lists for tokens `fred:name` and `fred:name||1;2`, namely Nodes 1, 3, 4, and 5.

As pointed out above, the query-materialization index in Figure 4.8 could be built by scanning all vocabulary entries in an already available inverted list (see Section 4.2.3). Since all writes to the query-materialization index are then done sequentially, we can regard this loading strategy as a *bulk-loading* algorithm to create a query-materialization index. Given that observation, it becomes apparent that the time to load a query materialization will be dominated by the time needed to process the neighborhood queries for each vocabulary token with one of the strategies of Sections 4.4 to 4.5.3.

Limitations of Query Materialization. The query-materialization index reduces the processing of neighborhood queries with a single token to a single lookup in an inverted list. We thus expect to achieve low response times for these queries. When we query for multiple tokens, however, the query-materialization index may produce incorrect results, as outlined in the observation below.

OBSERVATION 4.1 It is not always possible to reconstruct the neighborhood of a query Q from lookups to the query-materialization index. In particular, suppose we are processing queries of the form $Q = e^1 \cap e^2$, where both of e^1 and e^2 are attribute-value expressions. Then,

$$\exists\, e^1, e^2 : \left(Q = e^1 \cap e^2\right) \wedge \left(\overset{\leadsto}{Q}_{A^*}(G) \neq \overset{\leadsto}{e}^1_{A^*}(G) \cap \overset{\leadsto}{e}^2_{A^*}(G)\right)$$

Take, for example, $e^1 =$ `name=fred`, $e^2 =$ `hobbies` $\sim$ `piano`, and $A^* = \{$sameUniversity, graduatedSameYear$\}$. Inspecting the association-trail multigraph of Figure 4.1(b), we can see that the answer to $Q = e^1 \cap e^2$

is only Node 3 and therefore $\overset{\leadsto}{Q}_{A^*}(G) = \{1,3,4\}$. On the other hand, e^1 selects Nodes 3 and 5, making $\overset{\leadsto}{e}^1_{A^*}(G) = \{1,3,4,5,8\}$; e^2 selects Nodes 3 and 7, making $\overset{\leadsto}{e}^2_{A^*}(G) = \{1,3,4,7,8\}$. The intersection $\overset{\leadsto}{e}^1_{A^*}(G) \cap \overset{\leadsto}{e}^2_{A^*}(G) = \{1,3,4,8\}$ is thus different from $\overset{\leadsto}{Q}_{A^*}(G) = \{1,3,4\}$. □

The observation above shows us that the query-materialization index can only be used for a restricted subset of the query expressions in Definition 4.2, unlike the strategies of Sections 4.4 to 4.5.3. In addition to intersection expressions, navigation expressions are also not supported by the index, as they do not directly reference a token from the vocabulary. Thus, the query-materialization index is unable to answer the query in Example 4.2. However, in a ranked retrieval, keyword-only query model, we would allow users to pose only keyword, attribute-value, and union expressions. For those expressions, it is possible to combine the results from single-token lookups to the query-materialization index. This simplifying assumption is made, for example, by Dong and Halevy [55] when using a similar structure for extensional graphs. In contrast, we propose to adopt a hybrid approach to answer all query types over intensional graphs: The query-materialization index should be used to answer only a subset of the queries, while other strategies should be used to answer the queries not supported by this materialization. As discussed below, a hybrid approach may be justified not only for correctness but also for performance considerations.

Materializing a Subset of the Vocabulary. It is important to observe that creating a query-materialization index is only relevant if interactive query times cannot be offered by one of the strategies discussed in Sections 4.4 to 4.5.3. However, in that situation, constructing the index itself may be problematic. In Example 4.8, we have seen that strategies with average query-processing time of half a minute may still lead to indexing times in the order of months. One could argue that this problem may be resolved by massive parallelism. In other words, if a given installation has hundreds of machines to spare, then the indexing process could be partitioned by token in the vocabulary and parallelized, resulting in indexing times in the order of a few hours.

However, when such massive parallelism is not available, another solution is to exploit the distribution of queries to improve average query-response time. It is often the case that queries are skewed towards a few of the tokens in the vocabulary [150, 169]). It is therefore feasible to index in the query-materialization index only these tokens and process all remaining (infrequent) queries with a query-processing strategy from Sections 4.4 to 4.5.3. This combination results again in a hybrid approach, given that several distinct strategies are used at the same time. Note that the problem is similar to the materialized-view-selection problem [90]. The amount of indexing that will fit a given installation depends on the level of service to be offered, on the skew of the query distribution, and on the amount of computational resources available. Experimental results exploring these trade-offs are shown in Section 4.7.4.

4.5.5 Extensions

In this section, we briefly discuss two possible extensions to the indexing techniques presented above.

Handling updates. Following the same design philosophy used for traditional inverted list indexing, we may incorporate updates to the index structures described above via differential indexing [147]. When

association trails are added to the system, we build indexes for those association trails and combine results from all indexes at query processing. Likewise, when elements are added or removed from the dataspace, we record their neighborhoods in a differential index structure that is merged at query processing time with the main indexes. Thus, update performance will be proportional to the performance observed for small index creations (see Section 4.7).

Metric-space Indexing. An alternative view of association trails is that they are a declarative mechanism to define similarity among items in a dataspace. We could exploit that observation to index association trails with a metric-space approach. Given $n_j, n_l \in G.N$, we define a metric $d(n_j, n_l) = \sum_{i=1}^{n} d_i(n_j, n_l)$ where $d_i(n_j, n_l)$ is a distance function for association trail A_i with probability p_i, s.t.:

$$d_i(n_j, n_l) := \begin{cases} 0 & \text{if } n_j = n_l \\ 2 - p_i & \text{if } \exists (n_j, n_l, A_i) \\ 2 & \text{otherwise.} \end{cases}$$

Note that d respects triangle inequality. To answer neighborhood queries, we could index the nodes in the dataspace into a metric-space index [35]. While this approach seems to be compelling, it remains unclear how to integrate the query processing of association trail left and right queries into the computations of the metric d. In fact, materialization approaches such as the ones described in the previous sections would be necessary to provide efficient support. Exploring metric-space indexing on top of our materialization techniques is an interesting area for future work.

4.6 Ranking Association Trail Results

In this section, we briefly present a scheme to rank results from neighborhood queries posed over the association-trail multigraph. We begin by motivating the need for ranking and proceed by discussing how we may take advantage of query-dependent in-degrees and association trail probabilities.

Argument for ranking. For the sake of argument, assume that we have an association trail A_i with an equality predicate. The attributes on the equality predicate follow the same uniform distribution with cardinality *card*. In that case, the probability that two nodes n_1 and n_2 are connected by an intensional edge is $p_i(n_1 \rightarrow n_2) = 1/card$. If we have n association trails with the same characteristics, the probability rises to $p(n_1 \rightarrow n_2) = n/card$, given that association trails are independent from one another. For example, if $n = 100$ and $card = 100{,}000$, then $p(n_1 \rightarrow n_2) = 0.1\%$. Assume that association trail queries select N nodes from the dataspace graph. The probability above means that if we take a random sample of 0.1% of N the we will with high likelihood be able to reach all N nodes from this random sample by at least one association trail. In other words, a query that has a selectivity as low as 0.1% is able to produce as neighborhood results all N nodes selected by the association trails.

What the argument above tells us is that depending on the type of predicate used for the association trails, on the number of association trails, and on the selectivity of the original query Q, it may become easy for

neighborhood queries to produce a large number of results. It is therefore important to rank these results in order to determine which ones have higher chances of being of interest to the end user.

Ranking by Query-dependent In-degree and Association Trail Probabilities. As we have discussed previously, there may be several intensional edges connecting two nodes in the association-trail multigraph. For example, Nodes 3 and 4 in Figure 4.1(b) are connected to each other by three intensional edges. We argue that nodes that are connected by several edges tend to be more strongly related than nodes that are connected by fewer edges. In addition, we should also consider the quality of the intensional edges that connect two nodes. The quality of association trails is modeled through probabilistic association trails (see Section 4.3.5). We may assign to each intensional edge an edge weight equal to the probability p_i of the corresponding association trail that generates that edge. When evaluating how strongly related two elements are, we should take into account not only the number of edges, but actually the sum of the edge weights connecting these two elements.

Given the intuition above, now we may formalize our ranking scheme.

DEFINITION 4.7 (NEIGHBORHOOD RANKING) *Assume a query Q produces $Q(G) = \{qr_1, \ldots, qr_k\}$. The basic ranking scheme of the search engine used to retrieve those primary query results assigns corresponding scores $S_{qr_1}, \ldots, S_{qr_k}$ to those results. Assume further that we have a set of association trails $A^* = \{A_1, \ldots, A_n\}$, with corresponding probabilities $p_1, \ldots, p_n$. Then the neighborhood score $S^N_{nr_i}$ of each neighborhood result $nr_i \in N^Q_{A^*}$ is given by:*

$$S^N_{nr_i} := \frac{\sum_{j=1}^{k} (S_{qr_j} \cdot \sum_{l:\, \exists\, (qr_j, nr_i, A_l)} p_l)}{\sum_{m=1}^{n} p_m}$$

The final score $S^*_{r_i}$ of a result r_i is given by:

$$S^*_{r_i} := \begin{cases} S_{r_i} & \text{if } r_i \notin N^Q_{A^*} \\ S^N_{r_i} & \text{if } nr_i \notin Q(G) \\ S_{r_i} + S^N_{r_i} & \text{otherwise.} \end{cases}$$

□

Definition 4.7 states that the neighborhood score of a result is given by borrowing from the scores assigned by the search engine to the corresponding primary query results. The amount of score that a primary query result transfers to a neighborhood item is proportional to the weight of all edges connecting the two nodes, normalized by the total possible weight.

EXAMPLE 4.9 Assume $A^* = \{$sameUniversity, shareHobbies, graduatedSameYear, isFriend$\}$ with probabilities $p_1 = 0.2, p_2 = 0.5, p_3 = 0.1, p_4 = 0.7$ and a query result $Q(G)$ containing Nodes 4 and 8 with scores $S_4 = 42$ and $S_8 = 21$ over the multigraph of Figure 4.1(b). Then $N^Q_{A^*} = \{1, 3, 5, 7\}$ and:

1. $S^N_1 = (S_4 \cdot p_1) / \sum_{i=1}^{4} p_i = 5.6$;

2. $S_3^N = (S_4 \cdot (p_1 + p_2 + p_3)) / \sum_{i=1}^{4} p_i = 22.4$;
3. $S_5^N = [(S_4 \cdot p_4) + (S_8 \cdot p_3)] / \sum_{i=1}^{4} p_i = 21$;
4. $S_7^N = [(S_4 \cdot (p_2 + p_4)) + (S_8 \cdot (p_1 + p_4))] / \sum_{i=1}^{4} p_i = 46.2$.

As none of the nodes in the neighborhood is also a query result, then their final scores are given by their neighborhood scores, yielding the ranked list $\langle 7, 4, 8, 5, 3, 1 \rangle$. Notice one interesting effect: Node 7 acts as a "hub" that connects to both primary query results through high-quality associations. Therefore, it is ranked even higher than each individual primary query result. □

Note that our ranking scheme is simply a combination of the text-scoring model of the underlying search engine with a query-dependent in-degree metric. In this sense, the ranking scheme is related to other query-dependent metrics such as HITS [105]. In contrast to HITS, however, our ranking scheme is suited to our scenario, as it is computed on top of an intensional graph and takes into consideration association-trail probabilities. It has been shown recently that a simple combination of BM25F [141] with in-degree outperforms a number of other metrics, such as PageRank [25] and HITS [105], in the evaluation of Najork et al. [123]. These other ranking schemes could be evaluated on top of the association-trail multi-graph as well. Exploring these ranking alternatives is an interesting avenue for future work.

4.7 Experiments

In this section, we present the results of an experimental evaluation targeted at understanding the relative performance of the query processing techniques proposed for association trails in Sections 4.4 and 4.5. The goals of our experiments are:

1. to determine how the performance of query processing methods that do not use indexing changes as we scale on the number of trails. In addition, we also aim to understand how well the estimate developed in Section 4.4.2 predicts canonical-plan performance (Section 4.7.2).
2. to evaluate how the performance of query processing methods that do use indexing changes as we scale on the number of trails. Furthermore, we compare these methods to the best alternative without indexing (Section 4.7.3).
3. to estimate how indexing and query processing performance would be affected by Section 4.5.4's hybrid approach (Section 4.7.4).
4. to evaluate the sensitivity of all methods to the selectivity of the primary query Q (Sections 4.7.2, 4.7.3, and 4.7.4).

parameter	setting
number of people N	1,600,000
profile attribute skew z	0.5
profile attribute cardinality c	100,000
number of trails n	$0 \dots \mathbf{50} \dots 100$
query Q selectivity s	$0.01\% \dots \mathbf{0.1\%} \dots 20.48\%$

Table 4.1: Parameter settings used in the experiments

4.7.1 Setup and Datasets

Datasets. We have implemented a synthetic data generator to simulate the social network scenario used as motivating example throughout this chapter. The generator creates a scale-free graph [13][2] of N person nodes, in which each edge represents an isFriend connection. Moreover, each person has a profile with $2n+1$ attributes, where one attribute is a random person name and the remaining attributes are generated following a Zipf distribution with skew z and cardinality c [79]. In addition to the person nodes, we also generate nodes for comments and communities and associate them to people using a Zipf distribution on the number of connections. These nodes do not affect our evaluation directly, but they make the scenario more realistic by making sure that the data in the network is not composed of only person nodes.

Association trails are created among people making use of their profile attributes. All association trails follow the template $A_i := \texttt{class = person} \stackrel{\theta_i(l,r)}{\Longrightarrow} \texttt{class = person}$, $\theta_i(l,r) := l.a_k = r.a_{k+1}$, where a_k, a_{k+1} are attributes from the profiles (excluding person names). We scale up to a maximum of n trails. We expect such trails to strain the performance of our query processing methods, especially because attributes follow a skewed distribution. We have seen in Section 4.5.2 that association trails over equality predicates can generate graphs with a number of edges quadratic on the number of elements selected by the trail (in our case, all N people in the social network).

We summarize the main parameters we have used in the experiments in Table 4.1. Default settings for the parameters are highlighted. In addition to the person nodes, we have generated on the order of 500,000 comment nodes and 160 communities as additional data on the network. An important parameter for the experiments is the skew z of the attribute distributions. We present results with a mediumly skewed setting of $z = 0.5$. Skew in the distribution implies that some attribute values are much more popular than others, appearing in a significant fraction of the nodes. Intuitively, however, such frequent values are less discriminating than rarer ones. So one could argue that values with a frequency above a certain threshold of the nodes in the dataset should be excluded when creating meaningful association trails. For example, if a profile attribute states that you sleep between 6 and 9 hours per night, that is of little added value in terms of finding other people in the network, as the vast majority of the users will have the same attribute value. If you would sleep only 3 to 5 hours per night, however, then the attribute value becomes more discriminating. Using milder skew is equivalent to using only attribute values that are more discriminating. In spite of these considerations, we have also run our experiments with higher

[2] Silke Trissl and Ulf Leser have kindly made their scale-free graph generator available to us.

Original Data Size [MB]	Total Index Size [MB]	Rowstore Size [MB]	Inverted Index Size [MB]	Indexing Time [min]
2,768	3,138	2,056	1,082	43

Table 4.2: iMeMex Basic Indexes: size and creation time.

values for z and found, as expected, that the trends displayed the experiments in the following become even more dramatic. For brevity, only values with $z = 0.5$ are reported.

Setup. All experiments have been run on a 2-way AMD Opteron 280 2.4 Ghz Dualcore server, with 6 GB of RAM, and a 400 GB ATA 7200 rpm hard disk with 16 MB disk cache. Association trails, along with our neighborhood-query processing techniques, have been implemented in the iMeMex Personal Dataspace Management System (see Chapter 5). The iMeMex PDSMS already provides the basic index structures described in Section 4.2.3 and supports all query types described in Section 4.2.2. In addition, iMeMex provides reusable data structures for materialized views, bulk-loaded B+-trees, and inverted lists. For querying, we have not made use of multi-threaded parallelism in our implementation, so our code takes explicit advantage of only a single core; for indexing, on the other hand, we have used a multi-threaded mergesort implementation, resulting in a partial usage of the multiple cores available in the server.

We have indexed the dataset described above in iMeMex, resulting in the index sizes and time shown in Table 4.2. For all query-processing experiments, queries have been run with a warm cache and time averages were taken over at least 10 measurements. In order not to have our results influenced by the time necessary to process the original query Q, we have simulated Q by choosing nodes randomly from the persons in the dataset and creating a materialized view as an OID list for those nodes. Thus, processing Q is always equivalent to a scan in an OID list and can be typically carried out in subsecond response times. As a consequence, the performance differences observed in the experiments are due to association trail query processing strategies alone.

For all query-processing strategies, we obtain as results not only the nodes that qualify $\overset{\leadsto}{Q}_{A^*}$, but also an in-degree count for each node that states how many edges connect that node to a primary query result. These in-degree counts are a basic input for the ranking scheme described in Section 4.6 and their computation may affect the efficiency of our query processing techniques. We have not ranked the results after computing in-degree counts, as our primary goal was to compare the relative performance of the association trail query processing strategies in Sections 4.4 and 4.5, and the ranking step would be carried out in the same manner for all strategies. Note that exploring top-K techniques in the style of the algorithms proposed by Fagin [62] exceeds the scope of this chapter.

4.7.2 Association-Trails Query Processing

In this section, we evaluate the query processing performance of the strategies presented in Section 4.4: the canonical plan and the n-semi-join plan. Neither of these strategies uses association-trail indexing; however, all queries are processed over the basic index structures of Section 4.2.3.

Scalability in Number of Association Trails

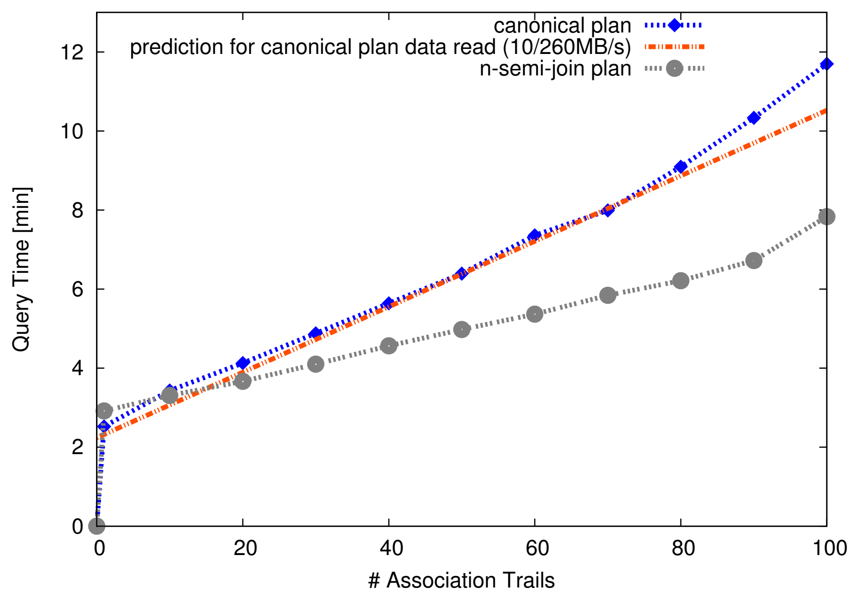


Figure 4.9: Query-response time versus number of association trails for methods without indexing

Query-Response Time. We have executed the canonical plan and the n-semi-join plan strategies to process neighborhood queries given an original query Q selecting 0.1% of the person nodes in our social network. Figure 4.9 shows how the performance of both strategies evolves as we add more association trails to the system. The first point in the graph is measured without association trails and the second with only a single association trail. As we may see, processing the query without association trails is extremely cheap, exhibiting response times in the order of 10 ms. When the first association trail is added, however, query-processing time rises sharply, reaching 2.5 min for the canonical plan and 2.9 min for the n-semi-join plan. This large query-response time is due to our having to process both the association trail left- and right-side queries and the additional semi-join to calculate the association trail neighborhood. As expected, the n-semi-join plans brings no benefit for a single association trail over the canonical plan; in fact, it has slight overhead.

As we scale on the number of trails, the differences between the two plans become apparent. The n-semi-join plan is able to process the queries in the left and right sides of the trails only once, while the canonical plan must spool the association trail queries at query time and scan this spool multiple times. At 100 association trails, the canonical plan needs 11.7 min to process the query; in contrast, the n-semi-join plan uses 7.8 min. As expected, the major cost involved in processing canonical plans comes from rescanning the spool as we have more association trails. This observation is confirmed by the formal model presented in Section 4.4.2, which is also plotted in Figure 4.9. We have tuned the transfer rates used as parameters to the model by instrumenting the canonical query plan and measuring the times necessary to process the materialization phase and the scan phase. Our measurements showed

Tr_{fetch} =10 MB/s and Tr_{spool} =260 MB/s in our implementation. As we may see in the figure, actual performance closely matches the prediction. For larger number of association trails, the canonical plan shows some overhead added on top of the prediction for data read. This overhead is due to the additional computational resources needed to process the multiple semi-joins in the plan.

In summary, we have observed that both canonical and n-semi-join plans do not offer interactive query processing performance for neighborhood queries. Canonical-plan performance deteriorates as a consequence of rescanning the spool for the queries on the left and right sides of association trails. While n-semi-join plans resolve that issue, they still must process the left- and right-side queries at least once and, in addition, determine through Algorithm 2 which results actually qualify.

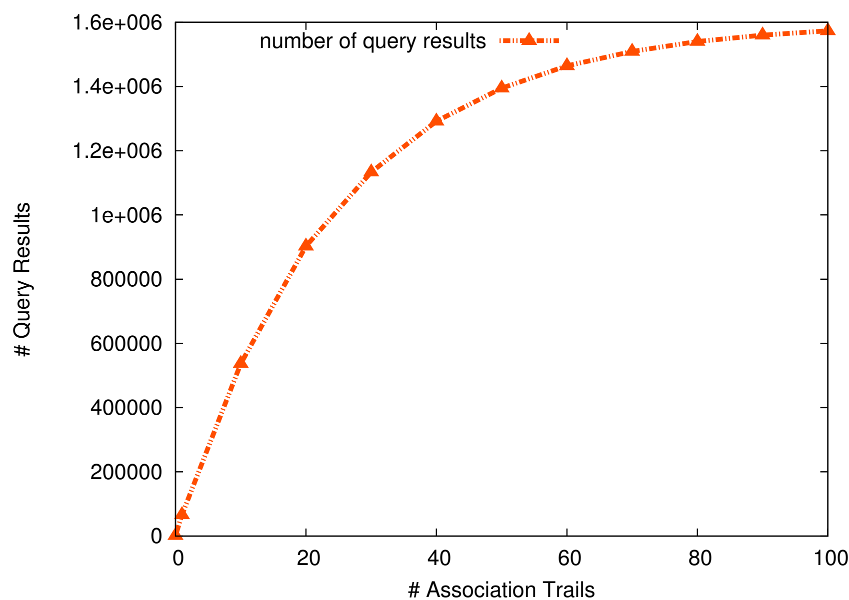


Figure 4.10: Number of query results versus number of association trails

Number of Query Results and In-degree. Two other important aspects that we have measured are the number of query results returned by a neighborhood query and the average query-dependent in-degree of those results. Figure 4.10 shows that, as we scale on the number of association trails, more query results are found by the neighborhood query. Note that this behavior occurs in spite of the fact that the original query Q has a fixed selectivity of 0.1% of the person nodes. This behavior can be explained by the argument in Section 4.6: More association trails imply a larger probability that any two given nodes will be connected in the association-trail multigraph. In contrast to that argument, however, which assumed a uniform distribution for attributes used in the association trails, we see in the figure that the rise in the number of results is not linear, being sharper in the first half of the graph. This sharper rise is expected because we have generated attribute values using a Zipf distribution with medium skew, meaning that some attribute values are more likely to appear in the query results. For these values, there are also connections to a larger number of elements in the neighborhood.

As we see in Figure 4.10, not only the increase in the number of neighborhood query results is sharp, but

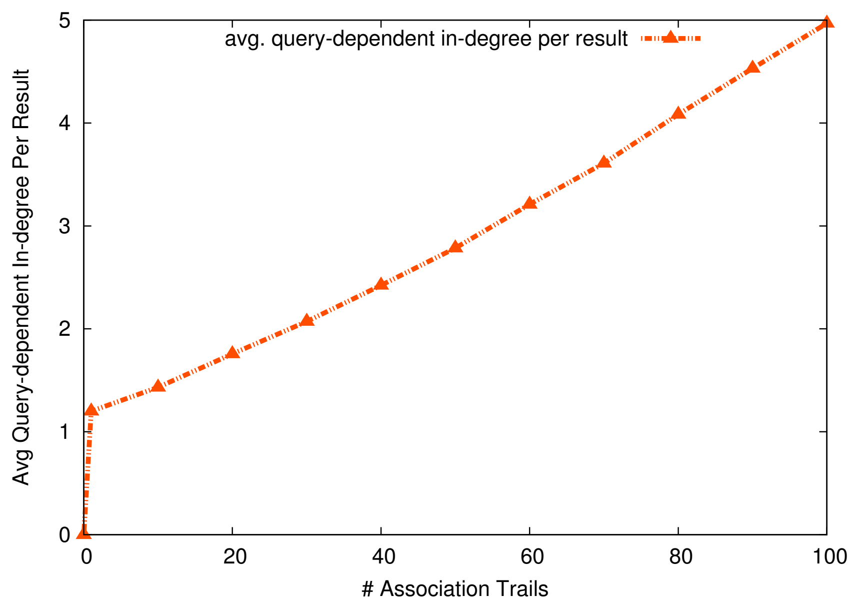


Figure 4.11: Query-dependent in-degree versus number of association trails

also the absolute number of results reaches almost the whole set of person nodes in our social network when we have 100 association trails. This result strongly motivates the need for association trail ranking. Figure 4.11 provides additional support. As we have seen in Section 4.6, a query-dependent in-degree metric could be used to rank neighborhood query results in addition to the basic scoring scheme in use for the primary query results. We show in Figure 4.11 that the average value for this metric scales linearly with the number of association trails, reaching an average of five connections between a neighborhood result and all primary query results at 100 association trails. Because fewer neighborhood results are added as more association trails are present, some neighborhood results will be more densely connected to the primary query results than others for large numbers of trails. Thus, query-dependent in-degree could be used as a feature for ranking.

In summary, for all methods processing neighborhood queries in an association-trail multigraph, we have further observed that the number of neighborhood results retrieved may grow significantly as more association trails are added to the system. In spite of that growth, ranking based on a query-dependent in-degree metric could be used to restrict attention to only a distinctive subset of those instances. It remains to be investigated, however, how to design algorithms that would obtain this distinctive subset without producing the whole neighborhood for the query. This challenge is an interesting direction for future work.

Sensitivity to Selectivity of Original Query

Figure 4.12 shows how the performance of the canonical and n-semi-join plans varies as we vary the selectivity of the original query Q to obtain a larger fraction of the person nodes in the social network. We have kept the number of association trails fixed at 50. We observe that while both plans are hardly

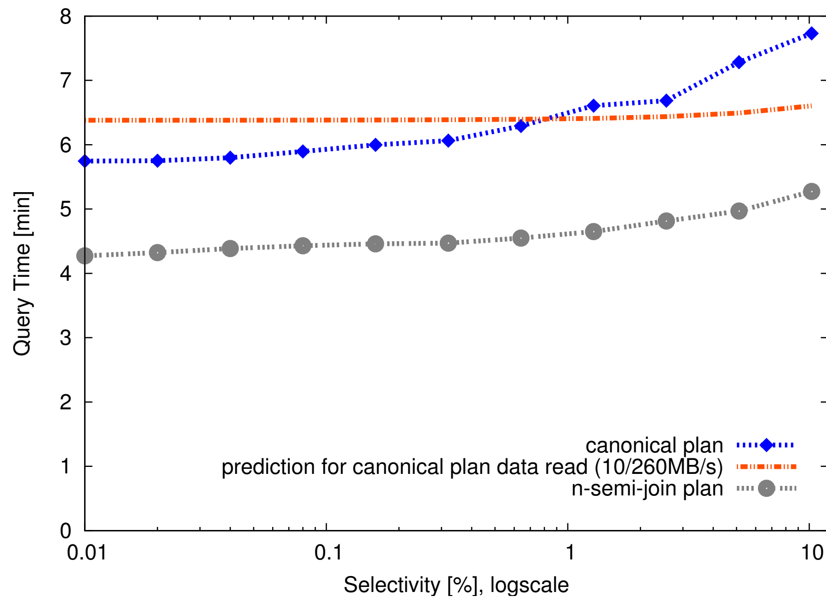


Figure 4.12: Query-response time versus selectivity of original query for methods without indexing

affected by the selectivity of the original query, the n-semi-join plan is more robust than the canonical plan. This effect can be explained by the fact that the canonical plan must scan a spool for the original query as many times as there are association trails. In addition, we observe that our prediction for the read performance of the canonical plan, while close to actual performance, was not as effective. An effect manifest in the figure is that when we have higher selectivities, the actual transfer rate Tr_{fetch} to obtain data from the basic index structures is higher due to better caching. Thus, actual canonical-plan performance is better than predicted. The inverse happens when the selectivity is low: Fetching results becomes more expensive and canonical-plan performance is worse than predicted. In order to accurately model this situation, we would have to convert the parameter Tr_{fetch} from a constant into a function of the original query selectivity. We do not believe that this additional complexity in the model would bring significant added value, however.

4.7.3 Association-Trails Indexing

This section presents the experimental results we have obtained with indexing strategies for neighborhood query processing over association-trail multigraphs (Section 4.5), with the exception of the hybrid approach of Section 4.5.4. Results for the hybrid approach are detailed in Section 4.7.4. Here we will focus on the following strategies: (1) materializing the left- and right-side queries of association trails as materialized views, (2) materializing the left-side query as a B+-tree and the right-side query as a materialized view, (3) materializing the join that specifies the full association-trail multigraph as an inverted list, and (4) materializing the grouping-compressed index of Section 4.5.3. For the first strategy, note that in our setup, the queries on both sides of the association trails are the same (`class=person`) and thus only one materialization is necessary.

Scalability in Number of Association Trails

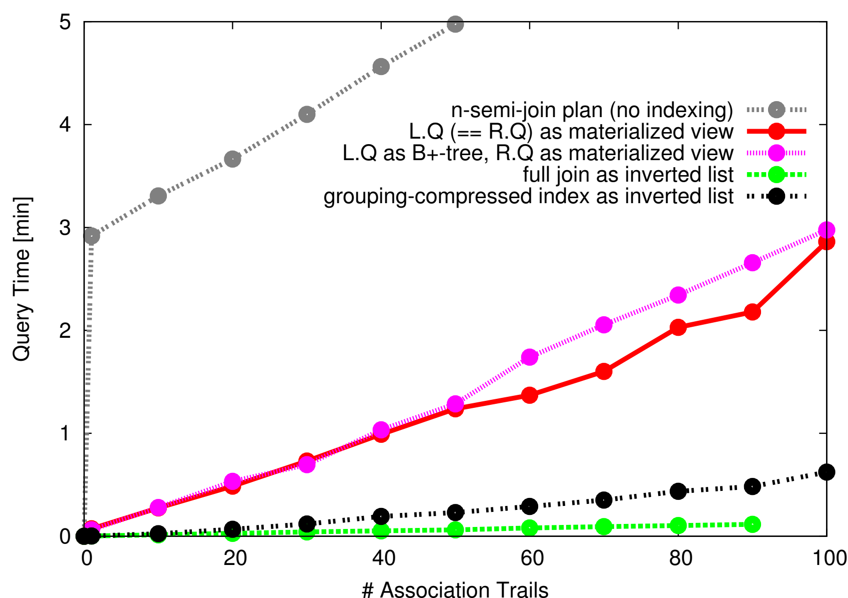


Figure 4.13: Query-response time versus number of association trails for methods with indexing

Query-Response Time. Figure 4.13 shows how the query performance of indexing strategies changes as we scale on the number of association trails. For reference, we also show the performance of the n-semi-join plan strategy, which is the best strategy that does not make use of association-trail indexing (see Section 4.7.2). As we may see from the figure, indexing has a significant effect in query-response time. For 50 trails, the n-semi-join plan takes 5 min to complete. Contrast that to a time of 1.2 min for both strategies that materialize the left- and right-side queries of the association trails, 13.8 sec for the grouping-compressed index, and 3.7 sec for the full join materialization. In addition, indexing strategies do not have as dramatic a jump in query-processing time when the first association trail is added to the system as the n-semi-join-plan. Recall that the n-semi-join plan has a processing time of 2.9 min for that point, while the left- and right-side materializations have times around 4 sec and the full and grouping-compressed join materializations have times under 0.5 sec.

The query-response time of all indexing strategies scales linearly with the number of association trails. Materializing both left- and right-side queries as materialized views had slightly better processing times than materializing the left side as a B+-tree and the right side as a materialized view for larger number of trails, though differences are not significant. This behavior indicates that for a selectivity of 0.1%, executing the expression $Q \cap A_i^{L.Q}$ using indexed nested loops rather than a hash join makes little difference. The grouping-compressed index significantly outperforms both strategies that materialize only left- and right-side queries. For 50 trails, the grouping-compressed index is a factor of 5.6 faster (1.2 min versus 13.8 sec); for 100 trails, it is still 4.8 times faster (3 min versus 37.4 sec). We expect that factor to shrink if a high number of lookups to the grouping-compressed index becomes necessary, e.g., due to low selectivity of the original query Q.

We have observed an interesting behavior for the full join-materialization strategy: Its query-response time, while also linearly increasing, remains rather low. It reaches 6.9 sec for 90 trails, in contrast to 29 sec for the grouping-compressed index. The reason for this behavior is that the grouping-compressed must perform more random lookups to process the neighborhood queries using Algorithm 3 than the full join materialization at the selectivity level of 0.1% for the original query. We could not scale the full join materialization above this point, however, due to large index sizes (reported below). In particular, we did not have enough temporary disk space in our server to build the full join materialization. In addition, in a set of separate experiments, we have also observed that the query performance of full join materialization is extremely sensitive to skew in the data, performing significantly worse than the grouping-compressed index when the skew of the attributes is increased to 0.95. We omit the detailed results for brevity.

In summary, indexing strategies bring significant query-response time benefits when compared to strategies that do not perform association-trail indexing. While the full join materialization can deliver good response times, it is strongly affected by indexing time and skew in the dataset, limiting its scalability. Among the methods that could support up to 100 association trails, the grouping-compressed index had the best performance, but query-response times become large (over 5 sec) when we scale above 20 association trails.

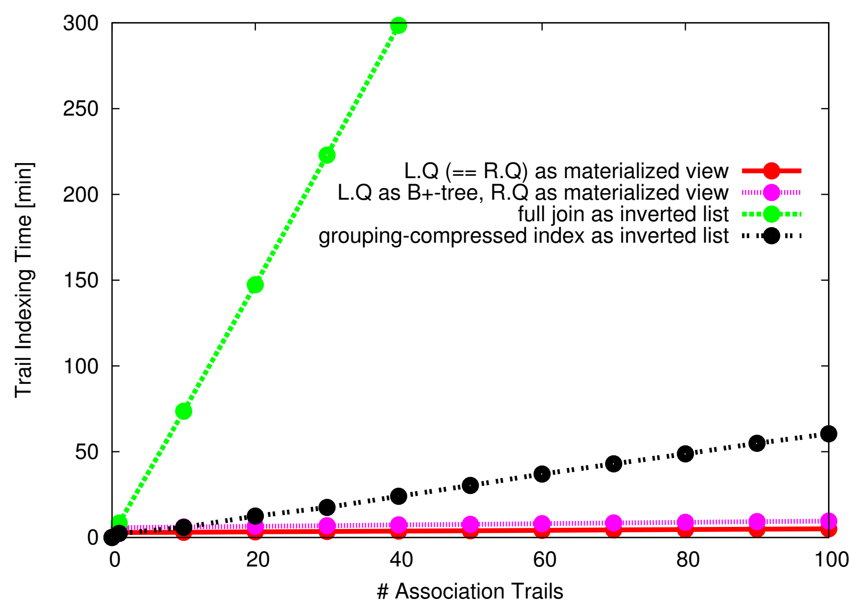


Figure 4.14: Indexing time versus number of association trails for methods with indexing

Indexing Time and Index Size. Figures 4.14 and 4.15 report indexing times and sizes for the indexing strategies discussed above when we scale in the number of association trails. To streamline the full join materialization, we have not represented association trail labels for each edge in the multigraph as in Figure 4.6, but have aggregated that information to a count of trails (necessary for in-degree counting). Even with that optimization, the figures show that indexing time and size for the full join materialization are dramatic. For 40 association trails, indexing time is already 5 hours in comparison to 24.0 min for the

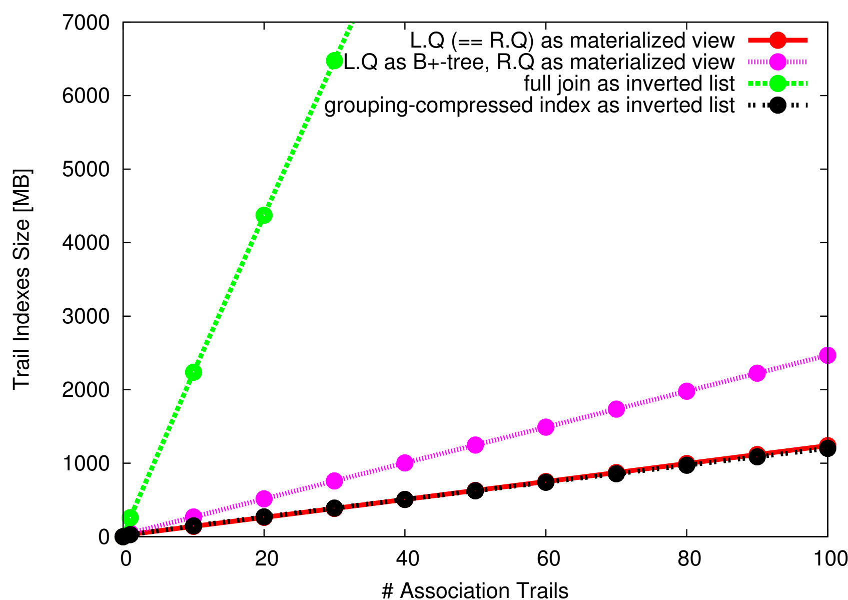


Figure 4.15: Index size versus number of association trails for methods with indexing

grouping-compressed index (a factor of 12.4), 7.3 min for materializing the left-side query as a B+-tree and the right as a materialized view (a factor of 40.8), and 3.7 min for materializing both sides of the association trails as materialized views (a factor of 80). Index size for the full join materialization was 8.3 GB for 40 association trails in comparison with 507 MB for the grouping-compressed index (a factor of 16.8), 1003 MB for materializing the left-side query as a B+-tree and the right as a materialized view (a factor of 8.5), and 507 MB for materializing both sides of association trails as materialized views (a factor of 16.8). We have not scaled the full join materialization above 90 trails, a point in which indexing time reached about 13.2 hours and index size was 17.8 GB. As mentioned previously, the temporary disk space needed to build the index was excessive beyond that point. The second most time-demanding strategy, the grouping-compressed index, required only 54.9 min to build an equivalent index (a factor of 14.4) that could be stored in 1.1GB (a factor of 16.8). In addition, the indexing time for the full join materialization is over an order of magnitude higher than the 43 min time needed to build the basic indexes over the dataset (Table 4.2, a factor of 18.4).

In spite of the large indexing times and sizes obtained for the full join materialization, all indexing strategies scaled linearly on the number of association trails. The second most time-demanding indexing strategy, the grouping compressed index, took about 1 hour to index 100 association trails, while the remaining materializations took 9.5 min (B+-tree variant, a factor of 6.3) and 5.1 min (materialized-view-only variant, a factor of 11.8). Those differences in indexing times are not impressive given the comparable differences in favor of the grouping-compressed index in query-response times. In terms of index sizes, however, the grouping-compressed index was in fact slightly more space-efficient than the best of the two variants. This advantage may be explained by the fact that the grouping-compressed index only needs to store OIDs, while the materializations based on the left- and right-side queries of association trails need to store a projection containing the attributes needed for the θ-predicates of the association trails.

The B+-tree variant of materializing left- and right-side queries of association trails exhibited indexing times and sizes roughly twice as large as the variant based on materialized views only. The reason for this behavior is that in our setup the queries on both sides of the association trails are the same (`class = person`) and thus the variant that uses only materialized views has the advantage of being able to create a single materialized view for both sides of the association trails. The increase in indexing times and sizes for those variants as more association trails are added to the system is due to the need to add more attributes from the association trails to the projections that are materialized.

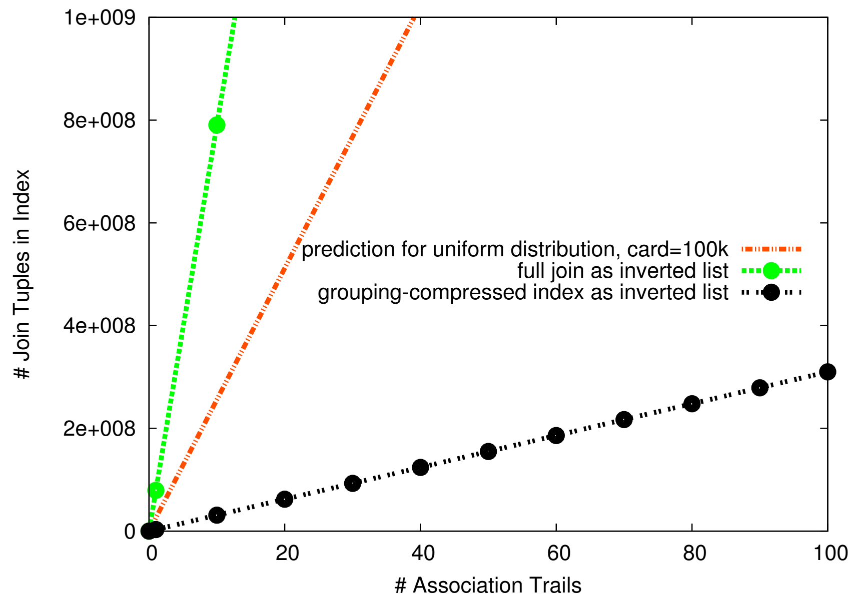


Figure 4.16: Number of join tuples (edges) versus number of association trails for join and grouping-compressed materializations

It is important to note that while the grouping-compressed index was more resilient to skew and created reasonably small indexes with adequate indexing time, it is representing the same information that is encoded in a full-blown join materialization. To better understand this fact, we evaluate an implementation-independent metric of those materializations: the number of join tuples stored in each of them. Recall from Section 4.5.3 that the full join materialization stores all edges in the association-trail multigraph, while the grouping-compressed index stores lookup edges to represent the same information. Each edge, regardless of its type, is counted in Figure 4.16 as one join tuple. We can see that the number of edges for both materializations seems to grow linearly on the number of association trails. That is in accordance to the argument presented in Section 4.5.2. In addition to stating that the full join materialization of association trails with equality predicates would grow linearly in the number of trails, the argument also tells us that the full join materialization requires a number of edges quadratic in the number of nodes selected by the trails (in our setup, the number of person nodes in the social network). For comparison, we show in the figure a curve with the predicted number of edges that the full join materialization would require according to that argument. Note that this curve assumes that attributes follow a uniform distribution. As expected, the number of join tuples created on the actual full join materialization was even higher

than predicted, because the attributes in our generated social network follow a Zipf distribution with medium skew. For 40 trails, the full join materialization created about 3.2 billion join tuples compared to 124 million for the grouping-compressed index; for 90 trails, the maximum we could scale the full join materialization to, we had amazing 7.1. billion join tuples being created by the full join materialization vis-à-vis 279 million for the grouping-compressed index.

In summary, the additional indexing investment necessary for the grouping-compressed index can be easily offset by the savings it provides in terms of query-response times. The trade-off is much worse for the full join materialization, which exhibits index sizes and indexing times larger by at least an order of magnitude. While the remaining strategies based on materializing the left- and right-side queries of the association trails have the fastest indexing times, their non-interactive query-response times greatly limits their applicability. In addition, little performance difference is seen between the variant based on a B+-tree and the variant based on materialized views only.

Sensitivity to Selectivity of Original Query

Figure 4.17 shows how query-response time for association-trail indexing strategies reacts to varying the selectivity of the original query Q. We can see that the methods based on materializing left- and right-side queries of the association trails are robust to lower selectivities, with their query-response times increasing from 1.1 min to 2.1 min when selectivities are lowered by three orders of magnitude from 0.01% to 10%. In constrast, the full join materialization is highly sensitive to lower selectivities, with response time increasing from 0.5 sec to 7.5 min over the same interval. That is a factor 935.4 increase, indicating that increases in query-response time for the full join materialization are roughly proportional to decreases in the original query selectivity. That behavior is consistent with the fact that lower selectivities imply that a proportionally larger fraction of the full join materialization must be processed in order to answer a neighborhood query. The grouping-compressed index was also affected by lowering query selectivities, but not as dramatically as the full join materialization. When compared to the full join materialization, the grouping-compressed index answered queries by a factor 3.7 faster at a 10% selectivity. It also outperformed the variants based on left- and right-side materialization. However, the gap in performance between the grouping-compressed index and these other strategies tended to diminish for lower selectivities. Lower selectivities induce more random lookups in the grouping-compressed index to process a neighborhood query, leading to poorer processing times.

4.7.4 Hybrid Approach

In this section, we evaluate the hybrid approach suggested in Section 4.5.4. Recall that the hybrid approach consists of distributing queries to two materializations: the query-materialization index and one of the materializations discussed in the previous section. The query-materialization index processes a (hot) subset of the queries by simple inverted list lookups, while the remaining queries are sent to the alternative materialization. For the alternative materialization, we have chosen the grouping-compressed

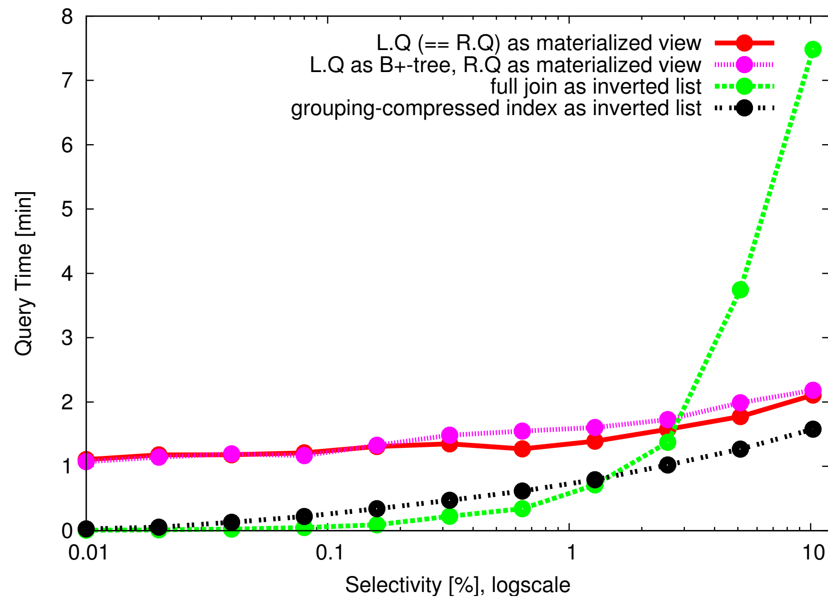


Figure 4.17: Query-response time versus selectivity of original query for methods with indexing

index, because our evaluation in the previous section indicates that it is the most robust of the structures we have studied.

The subset of the vocabulary indexed by the query-materialization index will determine its build time as well as the overall aggregated query performance. Building the query-materialization index implies executing neighborhood queries over all tokens in the vocabulary . As shown in Example 4.8, the time required may be on the order of several months if a materialization of the whole vocabulary is to be built. As we did not have enough resources to handle such a long indexing time, we have taken the following approach in our evaluation: (1) we have indexed 200 tokens from the vocabulary of our dataset using our implementation of the query-materialization index, and (2) we report in the following sections estimates for query processing, indexing time, and index sizes, based on extrapolating these basic measurements for 200 tokens. Note that as each node in the data has 200 attributes with cardinality 100,000, then 200 tokens represents 0.01% of a vocabulary of 2,000,000 tokens. While extrapolation for indexing time and index sizes may be done linearly, extra care must be taken to extrapolate query processing numbers, because queries will likely not follow a uniform distribution [150, 169]. In our estimates, we have assumed queries to follow a Zipf distribution with skew equal to 0.8. The estimates for query-processing time are made assuming that the hottest k tokens of the vocabulary are in the query-materialization index, for different values of k. All queries to other tokens are processed by the grouping-compressed index.

Scalability in Number of Association Trails

Query-Response Time. Figure 4.18 shows the estimated query-processing time for the hybrid approach. We have plotted several curves, each one corresponding to a different fraction of the vocabulary being indexed by the query-materialization index. When no tokens are in the query materialization, then we

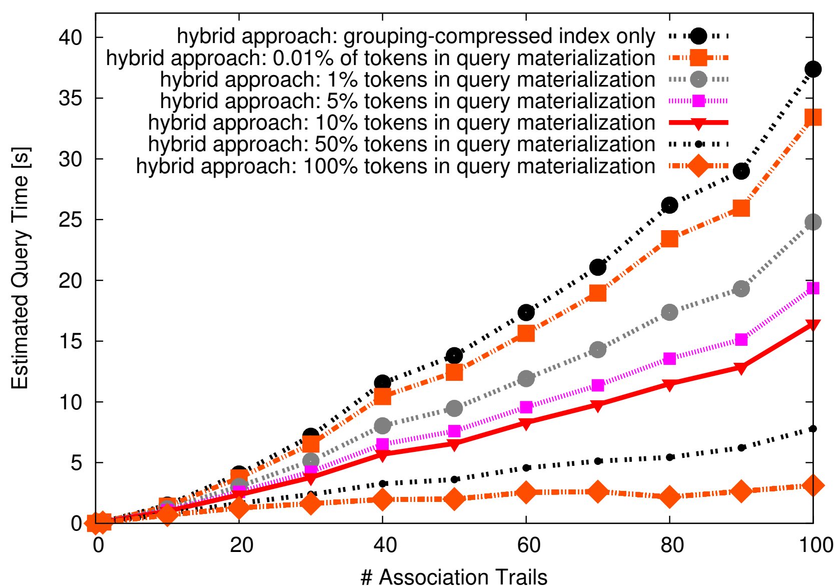


Figure 4.18: Query-response time versus number of association trails for the hybrid approach

assume all queries to be processed by the grouping-compressed index evaluated previously. Likewise, numbers reported for 100% of the tokens in the vocabulary are obtained by assuming all queries are processed with the query-materialization index. We show these two extremes and several other estimates in-between in the graph.

As noted previously, above 20 trails, query-response time is always above 5 sec for the grouping-compressed index. In addition, it increases linearly as we add more association trails to the system. At the other extreme, for the query materialization of 100% of the tokens, we have query-response times under 5 sec for all numbers of association trails used. In fact, query-response time for the full query materialization does not increase linearly with the number of association trails, but rather is correlated to the number of results the association trails obtain (see Figure 4.10). Thus, times increase faster until about 40 trails and then have a lower pace of growth, ending around 3.1 sec at 100 trails. This behavior can be explained by the fact that a query to the query-materialization index is answered by a simple inverted-list lookup, and the size of the inverted list to be scanned determines to a great extent the query answering time. As a consequence, query-response time is limited by the time to process an inverted list containing all nodes selected by the association trails (in our case, at most 1,600,000 person nodes). Note that for a response time of 3.1 sec, we have returned 1,573,543 results. Thus, response time cannot increase much further than 3.1 sec for the query-materialization index in our evaluation.

As expected, the behavior for different percentages of tokens in the query materialization stays inbetween these two extremes. For example, we could handle up to 60 association trails with average query-response time under 5 sec for a query-materialization index holding 50% of the tokens in the vocabulary. With 10% of the tokens in the vocabulary, average query-response times under 5 sec would only be achieved up to 30 association trails.

In summary, the hybrid approach allows us to reduce average query-processing time by providing increasing amounts of materialization. The correct amount of materialization will depend on the application's QoS requirements. One must observe, however, that materialization implies indexing costs, which are discussed below.

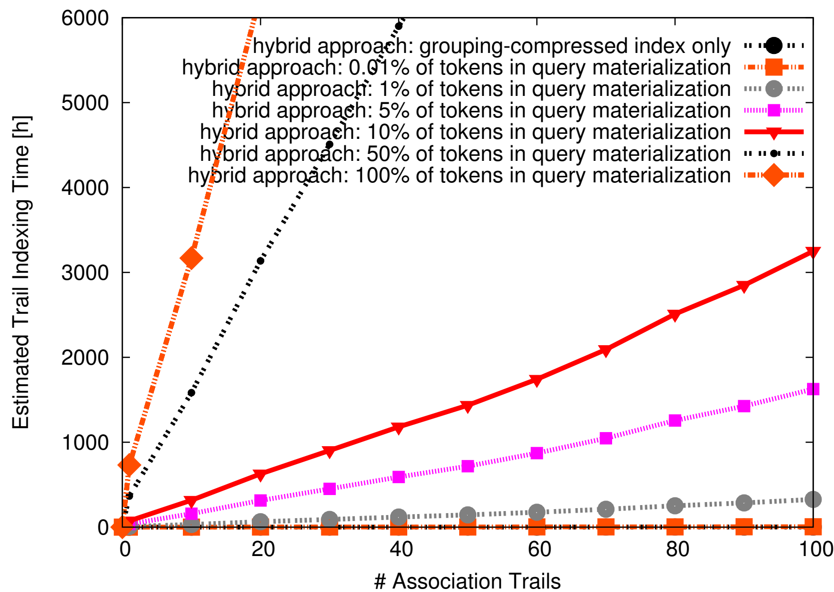


Figure 4.19: Indexing time versus number of association trails for the hybrid approach

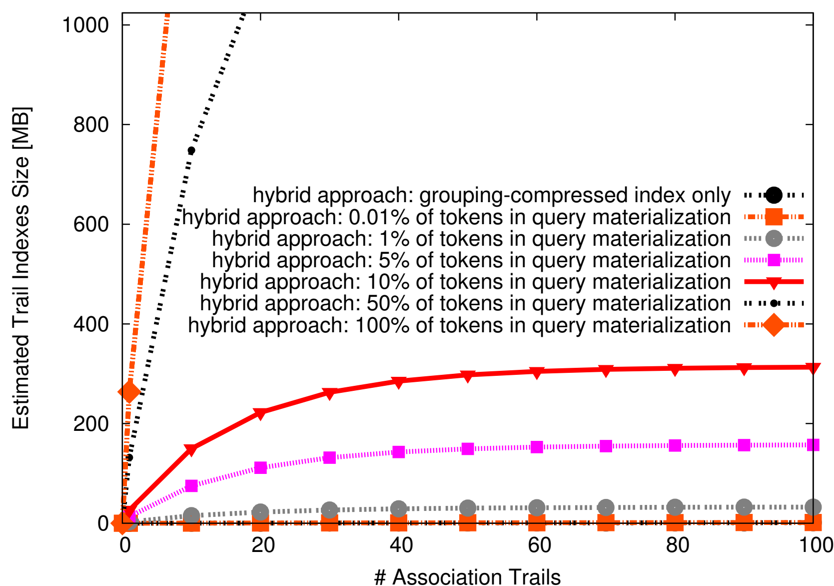


Figure 4.20: Index size versus number of association trails for the hybrid approach

Indexing Time and Index Size. Another two important aspects when choosing the amount of materialization to be performed are the total indexing time and index sizes. We evaluate these aspects for the

hybrid approach in Figures 4.19 and 4.20. In contrast to the results for query-response times, extremes are inverted: the grouping-compressed index has the shortest indexing time and smaller index size, while a query-materialization index with 100% of the tokens has the longest indexing time and largest index size. The grouping-compressed index, discussed previously, takes about 1 hour and 1.17 GB to index 100 association trails. A query-materialization index with 100% of the tokens would require as much as 45 months and 3.05 TB for the same amount of association trails. That dramatic difference is due to the fact that the query-materialization index must execute one neighborhood query for each token in the vocabulary using the grouping-compressed index. As we have seen, for 100 association trails, the grouping-compressed index takes 37.4 sec to process a query with 0.01% selectivity. If the vocabulary has 2,000,000 tokens, then simply executing that amount of neighborhood queries with the grouping-compressed index amounts to 28 months. However, the vocabulary contains tokens with different selectivities, making query-processing times even worse.

While the indexing time and size for the query-materialization index could be tackled by massive parallelism, we believe in many current scenarios such computational resources may not be available. Figures 4.19 and 4.20 show that indexing time and size can be reduced by indexing a smaller portion of the vocabulary in the query-materialization index. For example, if 10% of the tokens are indexed, then we would need 4.5 months and 313 GB to index 100 association trails, numbers proportionaly inferior to the numbers for 100% of the tokens.

In summary, it is possible to trade off query-response time for indexing cost using the hybrid approach. The correct trade-off will depend on the available computational resources and on the QoS requirements of the specific application scenario.

Sensitivity to Selectivity of Original Query

In the following, we estimate the sensitivity of the hybrid approach to the selectivity of the original query Q. Our results are shown in Figure 4.21. As we have seen previously, the grouping-compressed index is sensitive to selectivity, with its query-response time increasing as selectivities lower. Response time varies from 1.7 sec for a 0.01% selectivity up to 94.6 sec for a 10% selectivity, a factor 55.1 increase over three orders of magnitude increase on selectivity. The query-materialization index with 100% of the tokens, on the other extreme, is sensitive to selectivity of the original query Q only as long as it implies obtaining more query results. When we reach the maximum number of possible query results (1,600,000 person nodes in our scenario), then response times stabilize at about 3 sec. Different amounts of materialization in the query-materialization index allow us to achieve a behavior inbetween the two extremes of the grouping-compressed index and the complete query-materialization index. For example, if we index 10% of the tokens in the query materialization, then average query-response time would be 38.6 sec for 10% selectivity. Furthermore, it would remain below 5 sec up until 0.04% selectivity.

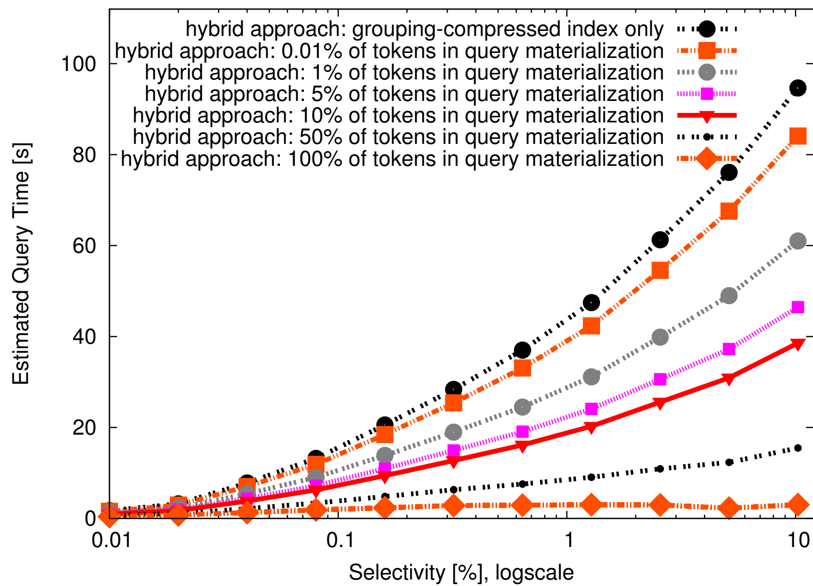


Figure 4.21: Query-response time versus selectivity of original query for the hybrid approach

4.8 Related Work

Metadata and Associations. Systems such as DBNotes [17] and Mondrian [73] extend the relational model to include annotations. Superimposed information [116] generalizes the notion of annotations by allowing users to associate any piece of information on a superimposed layer to any piece of information in a base layer through marks and links. These systems, however, represent annotations, marks, and links extensionally, while our approach allows users to relate any item in the dataspace (including data and annotations) intensionally. Srivastava and Velegrakis [153] propose an extension to the relational model in which queries may be used as data values to model associations between tuples. Their approach is similar to ours in two aspects: (i) associations are defined intensionally and (ii) no distinction is made between data and metadata. In contrast to the approach of Srivastava and Velegrakis [153], however, our approach does not require a schema to be created from the start for the data and it applies not only to the relational model but to a general graph data model. Furthermore, our query interface is a simple extension of keyword search with structural hints, which allows an end-user to explore the dataspace once association trails are defined by a data administrator, while Srivastava and Velegrakis [153] adopt full-blown SQL. In addition, we propose a number of querying and indexing techniques specific for neighborhood queries that may be easily implemented on top of data structures such as inverted lists, B+-trees, and materialized views, while Srivastava and Velegrakis [153] focus purely on relational indexes.

The idea of multiple hierarchies modeled by different colors in Colorful XML [97] is similar to creating graph overlays that model different relationships on the data. Our framework, in contrast, is more general, as we use a general graph model. Furthermore, little is said about indexing in the Colorful XML work [97] and we present efficient indexing techniques for important classes of queries over the graph overlay.

XML and Graph Indexing. Extensive work [26, 120] has been done on processing path queries on tree-

structured XML data. An extension of that line of research [34, 158, 161] explores indexing techniques to efficiently answer reachability queries in graphs instead of trees. Some more recent approaches have focussed on indexing support for more general RDF query patterns [126, 164]. Our focus is different, as we target exploratory neighborhood queries and not general path queries. In addition, this previous graph indexing work assumes that the graph is given extensionally as input to the indexing system. Our approach enables users and administrators to define and query the graph intensionally.

Several approaches work on providing support for queries that combine keywords and structure. Integrating keyword and structural restrictions is explored by Florescu et al. [64]. More recently, Kaushik et al. [103] describe an algorithm to combine structure indexes and inverted lists for the evaluation of simple path and keyword expressions (i.e., path expressions ended by a keyword). Their technique uses a structure index to reduce the number of inverted-list joins needed to evaluate a given path expression. As the inverted-list join result is then a superset of the correct result, the structure-index result is used as a post-filter. In contrast to our work, they do not explore how to intensionally specify relationships in a dataspace nor how to handle graph data. Furthermore, in our scenario, applying their technique amounts to building a join index over the associations, which entails unacceptable index size, as discussed in Section 4.7.

Other approaches that combine keyword and structural restrictions include studies on answering keyword queries over structured databases [6, 100, 173] and XML-IR search engines [9, 29, 156]. When answering keyword queries over structured databases, one important consideration is which elements to combine to form a query answer. In the case of XML, usually a lowest-common-ancestor subtree (LCAS) is computed, while for relational databases, the equivalent concept would be the smallest group of tuples containing the keywords. Thus, the neighborhood of the items that contain keyword matches is investigated to compose a query result. XML-IR search engines are rather more concerned with providing effective ranking schemes for selecting the top-K matching items for a query. The result granularity is determined based on the aggregated scores that each element receives. Propagation of scores from primary keyword matches to surrounding elements again entails (implicitly or explicitly) inspecting the neighborhood of nodes in the graph. All of this work on keyword and structure search is concerned with extensional graphs and their techniques would have to be revisited when the graph is defined intensionally, as in our approach.

Similarity Search. One could compare our approach of relating elements by intensional edges from distinct association trails to relating elements based on different dimensions. Similarity search on vector spaces has traditionally been supported by multidimensional index structures [70]. Applying those techniques to our scenario is not straightforward, however, as it is not clear how to translate association trails into a vector space. As discussed in Section 4.5.5, a possible approach is to rely on metric space techniques [35], where only a distance metric is necessary and not a vector space mapping. Such an approach would need to leverage the techniques developed in this paper to make the computation of the metric efficient. In addition, association trails are much more expressive as a model than metric-space approaches, allowing users to declaratively specify intensional graphs. Not only neighborhood queries are possible over such graphs, but also other more complex query types, e.g. path queries. Combining

intensional graph definitions with these more complex query types is an interesting avenue for future research.

WHIRL [39] introduces the notion of similarity joins based on text metrics. The goal is to find joining tuples that qualify a similarity metric above a certain threshold. Similarity predicates are one special type of θ-predicate that could be used to create edges among instances in the association-trail multigraph. Graph-mining approaches [31] target finding anomalous or frequent patterns occuring in extensional graphs, which can be represented as an N:M relation. In contrast, our approach offers a mechanism to define intensional graphs by expressing them using queries and N:M join semantics.

Dataspaces. Dataspaces were initially proposed in a vision paper by Franklin, Halevy, and Maier [65]. Indexing for exploratory queries in dataspaces has been studied by Dong and Halevy [55]. As in our approach, they also aim to process neighborhood queries; however, their approach is restricted to extensional graphs. Following a similar approach as the one suggested by Carmel et al. [29], Dong and Halevy [55] propose an index structure for neighborhood queries based on concatenating keywords with the context in which those keywords appear. We have constructed a similar index for association trails, resulting in good query performance and high indexing costs (Section 4.7). In contrast to the approaches of Carmel et al. [29] and Dong and Halevy [55], we have studied how to index a dataspace graph that is defined intensionally by association trails. Our approach enables us to explore different materialization techniques that allow trade-offs among index sizes, indexing time, and query performance. In addition, we have proposed a hybrid approach that allows users to choose the appropriate trade-off in these dimensions according to their needs.

In Chapter 3 of this thesis, we discuss a pay-as-you-go information-integration technique for dataspaces. In our approach, two *sets* of instances in the dataspace are related by defining a trail between these two sets. Whenever the user queries for the first set, then the second set will also be returned. To achieve that behavior, we propose a query-reformulation algorithm based on query containment. In contrast, association trails define relationships among *instances* on a dataspace. Therefore, even if containment is not verified, associations among *instances* are still exploited. A similar observation is valid regarding the approach of Das Sarma et al. [144]. Das Sarma et al. [144] represent relationships on a relational data integration system at the schema level using a probabilistic mediated schema. Association trails, on the other hand, work on general dataspace graphs and are concerned with instance-level relationships.

4.9 Conclusions

In this chapter, we have presented a technique based on *association trails* to model fine-grained relationships among instances in a dataspace. Association trails declaratively define a graph of instances over which queries may be posed. Our technique is general and may be applied to model such intensional graphs on a variety of scenarios, such as social networks and personal dataspaces. We have shown how to process exploratory neighborhood queries on top of an intensional graph defined by association trails. Our query-processing techniques follow the same intuition behind view merging in query rewriting [134]: We

combine the computation of the graph view with query processing in order to be able to answer queries without ever having to materialize the complete intensional graph naively.

Our evaluation has shown how diverse indexing and query-processing strategies react when the number of association trails is increased as well as when query selectivity is varied. One conclusion of that evaluation is that many strategies do not exhibit interactive query-response times for large number of association trails. In order to address this challenge, we have proposed a hybrid approach based on mixing association-trail indexing with extensive query materialization. It enables users to trade indexing cost for query response times in a way that matches their computational resources and application-specific QoS requirements.

As future work, we plan to adapt our techniques to better support top-K query processing in the style of the algorithms described by Fagin [62]. In that vein, we would like to compare different ranking schemes when computing top-K answers over the association-trail multigraph. In addition, we would like to evaluate how our techniques perform over large real social networks.

Chapter 5

A Dataspace Odyssey: The iMeMex Personal Dataspace Management System

A personal dataspace includes all data pertaining to a user on all his local disks and on remote servers such as network drives, email, and web servers. This data is represented by a heterogeneous mix of files, emails, bookmarks, music, pictures, calendar entries, personal information streams and so on. This chapter presents the architecture of a new breed of system that is able to handle the entire personal dataspace of a user. Our system, named `iMeMex` (integrated memex), is a first implementation of a *Personal DataSpace Management System (PDSMS)*. Visions for this type of systems have been proposed recently [48, 51, 65]. We begin the chapter by discussing the functionality that should be provided by a PDSMS such as `iMeMex`. We then show how the contributions discussed in the previous chapters of this thesis are brought together into a consistent system architecture. Finally, we present as a use case an application built on top of `iMeMex` that shows how `iMeMex` allows dataspace navigation *across data-source–file boundaries* and how `iMeMex` offers *rich contextual information* on query results.

5.1 Introduction

In 1945, Bush [27] presented a vision of a personal information management (PIM) system named *memex*. That vision has deeply influenced several advances in computing. Part of that vision led to the development of the *Personal Computer* in the 1980's. It also led to the development of *hypertext* and the *World Wide Web* in the 1990's. Since then, several projects have attempted to implement other memex-like functionalities [15, 42, 68, 102]. In addition, PIM regained interest in the Database research community [53, 87]. Moreover, it was identified as an important topic in the Lowell Report [3], discussed in a VLDB panel [104], and became topic of both SIGMOD 2005 keynotes [15, 122] and of a workshop running in conjunction with SIGIR and CHI [132, 133].

Personal Information Jungle. We argue that a satisfactory solution has not yet been brought forward to many central issues related to PIM. In fact, today's users are faced with a jungle of data-processing

solutions and a jungle of data and file formats [51]. This situation illustrates two key problems in the current state-of-the-art for PIM: *physical* and *logical data dependence* for personal information. *Physical data dependence* relates to the fact that users need to know about devices and formats that are used to store their data. *Logical data dependence* relates to the fact that users cannot define user-centric views over the data model that is used to represent data. Currently, there is no single system capable of offering an abstraction layer that overcomes both problems and enables data processing (e.g., querying, updating, performing backup and recovery operations) across files, formats, and devices.

From Databases to Dataspaces. DBMS technology successfully resolved the physical and logical data dependence problem for structured data, but not for the highly heterogeneous data mix present in personal information. Franklin, Halevy, and Maier [65] recognize this situation for a variety of domains, including PIM, and present a broad vision for data management. They argue for a new system abstraction, a *DataSpace Support Platform (DSSP)*, capable of managing all data of a particular organization, regardless of its format and location. Unlike standard data integration systems, a DSSP does not require expensive *semantic data integration* before any data services are provided. For example, keyword searches should be supported at any time on all data (*schema later* or *schema never*). A DSSP is a *data co-existence approach* in which tighter integration is performed in a *"pay-as-you-go"* fashion [65].

From Vision to Reality. In this chapter, we focus on *personal dataspaces*, that is the total of all personal information pertaining to a certain person. In contrast to the dataspace vision [65], we propose a concrete *Personal Dataspace Management System (PDSMS)* implementation, named `iMeMex`: integrated memex [48, 95]. A PDSMS can be seen as a specialized DSSP. Note, however, that our system is not restricted to personal dataspaces and can also be applied to other scenarios, e.g., social dataspaces. A first prototype of our system was demonstrated in previous work [51]. After that, we developed a unified data model for personal dataspaces (Chapter 2), a pay-as-you-go information integration framework (Chapter 3), and a technique to model intensional instance graphs in a dataspace (Chapter 4). This chapter presents the second prototype of `iMeMex`, which offers the architecture in which all of the above contributions are brought together.

5.1.1 Contributions

In this chapter, we make the following contributions:

1. We summarize the vision of `iMeMex`. In addition, we list current and upcoming features of our system.
2. We present the logical architecture of our system and also discuss how the its logical layers are implemented.
3. The `iMeMex` PDSMS frees the data contained in a dataspace from its formats and devices, by representing it using a logical graph model (Chapter 2). We present how to navigate, search, and query a dataspace managed by `iMeMex` using our AJAX interface.
4. We discuss how querying in `iMeMex` may be facilitated by trails (Chapter 3) and showcase how `iMeMex` allows users to exploit rich contextual information on query results (Chapter 4).

This chapter is organized as follows. The next section outlines the vision of our system. After that, we summarize iMeMex's logical architecture in Section 5.3. We then discuss in more detail the implementation of iMeMex's logical layers in Section 5.4. In Section 5.5, we list the current features of our software, which is open source under an Apache 2.0 License [95]. Section 5.6 presents a search application built on top of iMeMex as a use case. Sections 5.7 discusses related work. Section 5.8 concludes the chapter.

5.2 The iMeMex Vision

Some aspects of our vision were already presented in previous work [48, 51]. This section summarizes the core ideas. The ultimate goal of the iMeMex project is to free users from logical and physical data management concerns. What this goal means for a user is discussed in the following paragraphs:

PIM today: Assume that Mr. John Average owns a set of devices including a laptop, a desktop, a cellular phone, and a digital camera. His personal files are spread out among those devices and include music, pictures, PDF files, emails, and office documents. Today, Mr. Average has to copy files from one device to the other, he has to download data to his desktop to see the pictures he shot, he has to upload pictures to sites such as *flickr.com* or *picasa.com* to share them with his family. He has to make sure to regularly back up data from the different devices in order to be able to retrieve them in case of a device failure. Furthermore, he uses two different modes of searching: a local desktop search engine enabling search on his local devices and a web search engine enabling search on public web sites. Mr. Average may organize his files by placing them in folder hierarchies. However, the files and data items stored on his different devices are not related to each other.

iMeMex vision of 2010: Mr. John Average still owns several nifty devices with growing processing and storage capabilities. Instead of handling "devices" he assigns all his data to a logical *dataspace* named `John's space`. Whenever he listens to a piece of music, takes a picture, gets an email, etc., those items are assigned to `John's space`. His dataspace management system takes care of the low-level issues including replicating data among devices and enabling search and querying across devices. Whenever John Average wants to share data, he simply creates a subdataspace like `John's space:pictures` and selects a list of people who may see that data, e.g., his family or friends. There is no need to "upload" or "download" data: John's family and friends will just *see* John's pictures without requiring them to access web servers or to mess around with files. The boundary between the Web and the different operating systems running on his local devices is gone. However, John Average still *owns* his data: All master copies of his data are physically stored on devices he owns. Searching a dataspace is not restricted to certain devices (such as the local desktop), but includes all devices containing data assigned to his dataspace. Other than simple keyword queries, structural queries similar to NEXI [159] are enabled. John Average may also search and query the dataspaces of his friends and his family. The search granularity is fine-grained "resource views" (Chapter 2) and *not* files. Other than just searching or querying, John Average may also use iMeMex to integrate the information available in his dataspace or his friends' dataspaces in a pay-as-you-go fashion. John may provide "hints" to the system that help it find the items he is looking

for or that simply work as shortcuts to structural constraints, as a generalized form of bookmarking. In addition, his dataspace management system analyzes the data in the dataspace and proposes relationships among data items. It enhances his dataspace over time and helps him to turn a set of unrelated data items into integrated information (Chapters 3 and 4). Finally, Mr. Average may also update data using iMeMex. However, he may still update his data using any of his applications, bypassing iMeMex.

5.3 iMeMex Core Architecture

In this section, we discuss the core architecture of the iMeMex PDSMS. The iMeMex PDSMS is based on a layered architecture that is described in Section 5.3.1. Following that, Sections 5.3.2 and 5.3.3 discuss important services that are provided by the different layers.

5.3.1 Logical Layers

The DSSP vision of Franklin, Halevy, and Maier [65] defines a dataspace as a set of participants (or data sources) and relationships among the participants. We term the set of data sources the *Data Source Layer*. Although Franklin et al. [65] present services that should be provided by a DSSP, little is said on how a DSSP would provide those services on top of the Data Source Layer. In fact, in the current state-of-the-art for personal information management, applications (e.g., search-and-browse, email, Office tools) access the Data Source Layer (e.g., file systems) directly. This comes at the cost of physical data dependence, e.g., system dependence. This situation is depicted on the left of Figure 5.1.

To remedy that situation, we argue that what is missing is a logical layer between the applications and the Data Source Layer that provides services on the dataspace. We propose to add the iMeMex PDSMS as that intermediate logical layer. It is depicted on the right of Figure 5.1. The iMeMex PDSMS abstracts from the underlying subsystems, from data formats, and from devices, providing a coherent view to all applications. However, iMeMex does not have full control of the data as it is the case with DBMSs. Thus, applications may also access the data sources bypassing iMeMex, e.g., email or office applications do not have to be rewritten to interact with iMeMex: They work directly with the data sources. Other applications, however, may be rewritten to directly operate on iMeMex, e.g., browsers and OS shells.

In the following, we discuss the characteristics of each layer of the iMeMex PDSMS as well as of the layers with which it interacts. All of these layers are shown on the right of Figure 5.1.

Data Source Layer. This layer represents all subsystems managed by the PDSMS. A subsystem that participates on the dataspace may offer either an API that enables full access to the data on that subsystem, access through querying only, or a hybrid of these two options. Thus, the PDSMS must be aware of data versus query shipping trade-offs [106] to enable efficient query processing.

Physical Data Independence Layer (PHIL). This layer is responsible for resolving the data model, access protocol, and format dependence existing on the data sources participating in the dataspace. PHIL offers unified services such as *data model integration* and *indexing and replication*. We provide more

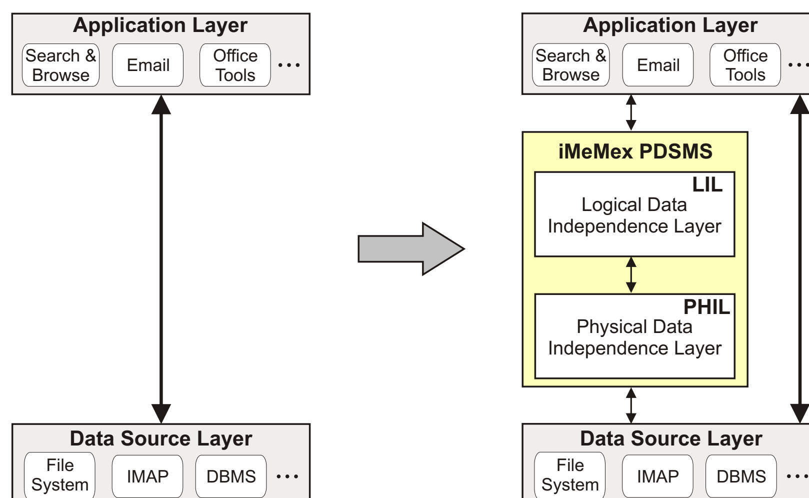


Figure 5.1: iMeMex improves the current state-of-the-art in PIM by introducing logical layers that abstract from underlying subsystems, data formats, and devices.

details on these services in Section 5.3.2.

Logical Data Independence Layer (LIL). This layer provides view definition and query processing capabilities on top of PHIL. LIL offers services such as a *personal dataspace search&query language* and *pay-as-you-go information integration* on top of the data unified by PHIL. We discuss important aspects of these services in Section 5.3.3.

Application Layer. This layer represents the applications built on top of the iMeMex PDSMS. As a PDSMS does not obtain full control of the data, applications may choose to either benefit from the services offered by the PDSMS or access the underlying data sources and use specialized APIs. To enable legacy applications to directly interface with the PDSMS, a PDSMS may offer a mechanism for integrating seamlessly into the host operating system, as demonstrated in previous work in the iMeMex project [51].

5.3.2 PHIL Services

The primary goal of PHIL is to provide physical data independence. Thus, PHIL unifies data reachable in distinct physical storage devices, access protocols, and data formats. We present the main services offered by PHIL below.

Data-Model Integration. Data-model integration refers to the representation of all data available in the data source specific data models using a common model: the *iMeMex Data Model (iDM)* (see Section 2.3). In a nutshell, iDM represents each piece of personal information by fine-grained logical entities. These entities may describe files, structural elements inside files, tuples, data streams, XML, or any other piece of information available on the data sources. These logical entities are linked together in a graph that represents the entire personal dataspace of a given user. The iMeMex approach is in sharp contrast to *semantic integration*, in which expensive up-front investments have to be made in schema mapping, in order to make the system useful. We follow a pay-as-you-go philosophy [65], offering basic services on

the whole dataspace regardless of how semantically integrated the data is.

Indexing and Replication. Given a logical data model to represent all of one's personal information, the next research challenge is how to support efficient querying of that representation. One may consider a pure mediation approach, in which all queries are decomposed and pushed down to the data sources. Though this strategy may be acceptable for local data sources, it may incur long delays when remote data sources are considered. In order to offer maximum flexibility, PHIL offers a hybrid approach. Our approach is based on a *tunable mechanism to bridge warehousing and mediation*. Any data source registered in iMeMex may be either mediated (query shipping) or warehoused (data shipping). On top of PHIL, however, that decision is transparent. In addition, we may choose to replicate name, tuple, and group components of resource views that come from remote data sources, but neither index nor replicate their content. Thus, basic graph navigation among resource views can be accelerated by efficient local access to the replica structures, while retrieval of bulky resource view content will incur costly access to (possibly remote) data sources. This mechanism allows users to tune iMeMex to act in different configurations, such as a mediation system, a warehousing system, a metadata repository, or a mix thereof.

5.3.3 LIL Services

The primary goal of LIL is to provide logical data independence. LIL enables posing complex queries on the resource view graph offered by PHIL. We discuss the services provided by LIL in the following.

Personal-Dataspace Search-and-Query Language. LIL processes expressions written in a new search-and-query language for schema-agnostic querying of a resource view graph: the *iMeMex Query Language (iQL)* (see Section 2.5). In our current implementation, the syntax of iQL is a mix between typical search engine keyword expressions and XPath navigational restrictions. The semantics of our language are, however, different from those of XPath and XQuery. Our language's goal is to enable querying of a resource view graph that has not necessarily been submitted to semantic integration. Therefore, as in content and structure search languages (e.g., NEXI [159]), our goal is to account for impreciseness in query semantics, which can be resolved using trails. In addition, we aim to provide an intuitive syntax for end-users. For example, by default, when an attribute name is specified (e.g., size>10K), we should not require exact matches on the (implicit or explicit) schema for that attribute, but rather return fuzzy, ranked results that best match the specified conditions (e.g., size, fileSize, docSize). This flexibility allows us to define malleable schemas as in the work of Dong and Halevy [54]. A PDSMS, however, is not restricted to search. Other important features of iQL are the definition of extensible algebraic operations such as joins (see Chapter 2).

Pay-as-you-go Information Integration. In order to define the possible semantics of query expressions, LIL offers users the possibility to define semantic trails (Chapter 3). In the example above, we could specify the distinct interpretations of the attribute size by different trails, e.g.:

//*.tuple.size⟶//*.tuple.fileSize and //*.tuple.size⟶//*.tuple.docSize.

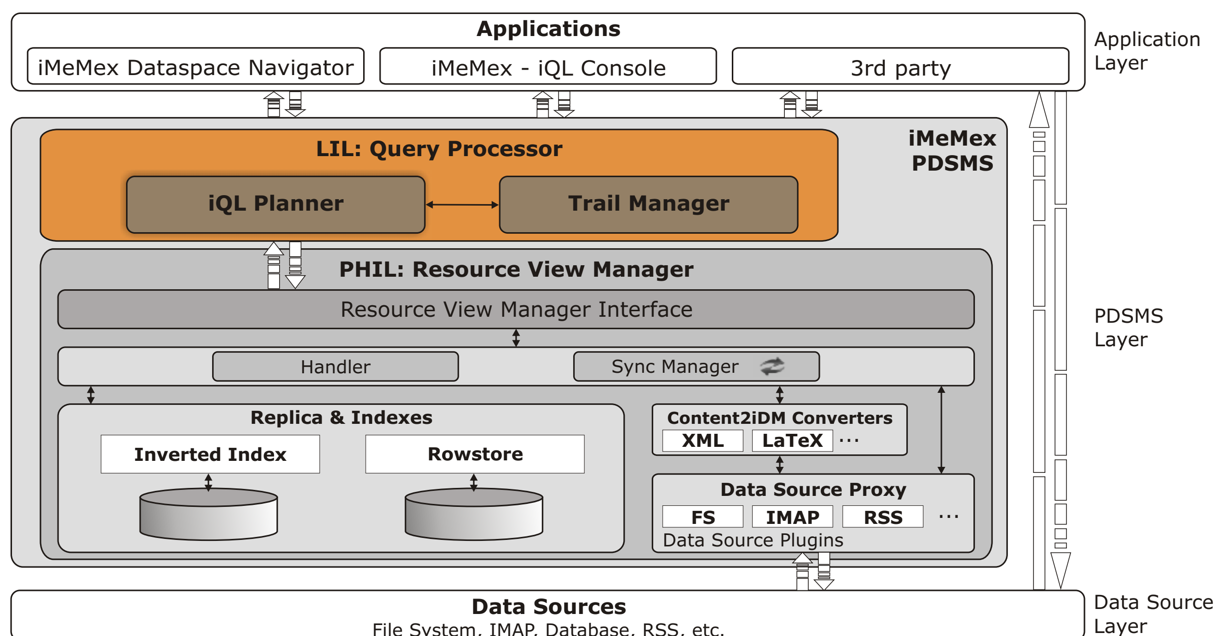


Figure 5.2: iMeMex Component Architecture

Furthermore, trails allow users to specify other types of integration "hints", using combinations of keyword and path queries. As users add more trails to the system, they gradually increase the level of integration of their personal dataspace, providing increasingly more information about semantic equivalences among sets of instances in their dataspace. In addition to relationships among sets of instance, LIL also offers users the possibility to define instance-level relationships with association trails (Chapter 4).

5.4 Implementation of Logical Layers

This section gives an overview of the implementation of the logical layers LIL and PHIL, presented in the previous section. As we have seen, PHIL is a logical layer that abstracts from underlying subsystems and data sources such as file systems, email servers, network shares, iPods, and RSS feeds. It is implemented by a component termed the *Resource View Manager*. LIL focusses on providing advanced querying and pay-as-you-go information integration. It is implemented by our *Query Processor* component. Both components are depicted in Figure 5.2, along with their implementation in the iMeMex PDSMS. We explain them in detail below.

5.4.1 Query Processor

The main task of the *Query Processsor* is to parse incoming queries and to create logical query plans for them. These logical query plans include all rewrites that must be performed due to semantic and association trails. In order to achieve this goal, the Query Processor is further broken down into two

major subcomponents: (1) the iQL Planner and (2) the Trail Manager.

(1) The **iQL Planner** transforms iQL query strings into query plans expressed in a logical algebra (similar to the one presented in Section 3.2.3). It then interacts with the Trail Manager to obtain a version of the logical plan refined by both semantic and association trails. After obtaining this logical plan, the iQL Planner also applies a set of optimization rules to the query plan and pushes down the plan to the Resource View Manager. At the moment, our planner is mostly rule-based. We have also started investigating cost-based optimization; this feature will be further explored as another avenue of ongoing and future work.

(2) The **Trail Manager** is responsible for rewriting logical query plans with both semantic and association trails. Note that semantic trails may change query sub-trees, while association trails consider the original query as a whole. Therefore, we first apply all semantic trails to the query and then perform association-trail rewrites. This strategy also has the added advantage that all set-level heterogeneity is removed by semantic trails *before* related instances are computed by association trails.

5.4.2 Resource View Manager

The *Resource View Manager (RVM)* is the central component to managing resource views. It consists of four major components: (1) Data Source Proxy, (2) Content2iDM Converters, (3) Replica-and-Indexes Module, and (4) Synchronization Manager.

(1) The **Data Source Proxy** provides connectivity to the different types of subsystems. It contains a set of *Data Source Plugins* that represents the data from the different subsystems as an initial iDM graph. Currently we provide plugins for file systems, IMAP email servers, JDBC databases, and RSS feeds.

(2) The **Content2iDM Converter** further enriches the iDM graph provided by the data source proxy. This enrichment is achieved by converting content components to iDM subgraphs that reflect the structural information. Currently, we provide converters for XML, LaTeX, and PDF files, among others.

(3) The **Replica-and-Indexes Module** consists of a set of indexes and replicas. A *replica* creates a copy of a set of resource view components inside the RVM. For instance, one strategy could be to replicate the *group components* of all resource views that were retrieved from remote data sources. As a consequence, queries referring to the group component of resource views can then be executed exploiting the replicas only. We thus avoid time consuming lookups at the remote data source. As replication may require additional disk and memory space, there is a general trade-off between *data versus query shipping* [106] that has to be considered when creating replication strategies. An *index* creates specialized data structures to speed up look-up times, such as inverted lists or B+-trees.

In our current implementation, we employ an Inverted Index and a Rowstore. The **Inverted Index** tokenizes resource view components (name, tuple, and content) and records lists of occurrences for each token found. Tokens include the actual data value in the component concatenated with the component name, as described in Section 4.2.3. The **Rowstore** is a replica of all resource view components (name, tuple, content, and group). During the development of `iMeMex`, we have provided several different implementations for the Replica-and-Indexes Module. In particular, we experimented with keeping a Resource

View Catalog in a relational DBMS (Apache Derby [45]) and also with vertically partitioning resource view components into separate inverted lists (Apache Lucene [112]). Our experience, however, is that implementations based on those libraries presented us with several scalability problems. So we have implemented the index structures mentioned above from scratch, while componentizing basic data structures such as B+-trees and inverted lists. Although this strategy implies that we reuse less freely available code, it gives us complete control over our index structures.

(4) The **Synchronization Manager** observes all registered data sources for updates. When a new data source is registered at the RVM, the Synchronization Manager will analyze the data found on that data source and send each resource view definition to the Replica-and-Indexes Module. Depending on the current replication and indexing strategy, the Replica-and-Indexes Module will then trigger replication and indexing for the newly created resource views. In the future, we plan to integrate support into the Synchronization Manager for processing update events commonly provided by operating systems such as Windows, Mac OS X, and Linux.

5.5 System Features

5.5.1 Current Features

In this section, we present current features of our system as of August 2008.

1. The server is implemented in Java 5 and is platform independent. It currently consists of approximately 120,000 lines of code and 1,500 classes.
2. `iMeMex` is based on a service-oriented architecture as defined by the OSGi framework (similar to Eclipse). As a consequence, services, e.g., data source plugins or content converters, can be composed into different system configurations. As an OSGi implementation, we currently use Oscar [128].
3. All data is represented by the *iMeMex Data Model*, presented in Chapter 2.
4. Our query parser supports an initial version of iQL as presented in Section 2.5. Our language allows for a mix of keyword and structural search expressions.
5. We allow users to perform pay-as-you-go information integration based on iTrails, presented in Chapter 3.
6. We support neighborhood queries to explore a dataspace that return results over the intensional graph defined by association trails, as discussed in Chapter 4.
7. We allow users to create materialized views for any iQL query and the system exploits these materialized views during query processing.
8. We provide a rule-based query processor that is able to operate in three different querying modes per data source: warehousing (only local indexes and replicas are queried), mediation (local indexes are ignored, queries are shipped to the data sources), and metadata repository (same as warehousing, except that content is not replicated, i.e., accesses to content components are routed to the specific data source).

9. We provide two basic indexing strategies for data sources that are fully or partially warehoused: an inverted index and a rowstore. These structures are implemented on top of external memory inverted lists and B+-trees. The indexes that are employed by `iMeMex` are also configurable, due to the flexibility of OSGi.
10. Scalability: We have been able to handle up to 25 GB of indexed data (net size, excluding image or music content) on a single iMeMex instance. The biggest file indexed was 7 GB.
11. We have implemented wrapper plugins for the following data sources:
    1. File systems (platform independent: works for Windows, Linux, and MAC OS X)
    2. Network shares (SMB)
    3. Email server (IMAP)
    4. Databases (JDBC)
    5. Web documents (RSS/ATOM, i.e., any XML data that is accessible by a URI)
    6. Specialized web sources (so far, Google [76] and TelSearch [155])
    7. Large ZIP Files (of up to 2 GB in size)
12. We provide content converters for LaTeX, Bibtex, XML (SAX-based), PDF, MP3, JPEG, and TIFF.
13. We offer developers a simplified interface to develop data source plugins and content converters in Python.
14. `iMeMex` provides two important interfaces:
    1. A text console that allows users to perform all administration tasks and also allows them to query the server.
    2. An HTTP server supporting two different data delivery modes: XML and binary (GWT [82]). We are also developing an `iMeMex` client that accesses the `iMeMex` server through the HTTP interface. The current version of that client is shown in Section 5.6.
15. The `iMeMex` server has been open source since December 2006 under an Apache 2.0 License (for more details, please visit `http://www.imemex.org`).

5.5.2 Upcoming Features

We are planning to provide the following features with upcoming releases of our software:

1. OS integration for file events (Mac and Windows, using native libraries and C++)
2. Cost-based query optimization
3. Integration of updates from data sources
4. Data replication and sharing framework
5. Support for datasets larger than 25 GB, scaling beyond 1 TB using distributed instances

5.6 Use Case

Our use case consists of two parts. First, we introduce our AJAX GUI built on top of `iMeMex` (Section 5.6.1). Second, we present how to navigate the personal dataspace using that client (Section 5.6.2).

5.6.1 iMeMex Dataspace Navigator

In this section, we introduce the `iMeMex` Dataspace Navigator GUI. It is displayed in Figure 5.3. Our GUI consists of three main components: (1) the iQL Search-and-Query Box, (2) the iDM Graph Display, and (3) the Result Display.

The **iQL Search-and-Query Box** can be found in the upper part of the GUI. This box can be used to enter arbitrary iQL expressions. As introduced in Section 2.5, iQL is the search-and-query language we use to query the iDM graph. In contrast to search engine approaches (e.g., Google), operating systems (e.g., Windows Explorer), and DBMSs (e.g., languages such as SQL and XQuery), our iQL language unifies keyword search, structural querying, addresses and paths into a single language. Our goal for iQL is to offer an intuitive syntax such that it may be used by non-experts, while at the same time offering some advanced features that will be useful for the expert user.

The **iDM Graph Display** occupies the left part of the GUI. It displays the dataspace managed by `iMeMex`. Currently, we use a tree-like representation to represent that graph (i.e., we show all possible paths on the graph starting from nodes chosen by the user). The reason is that today's users are already familiar with tree-like interfaces. This way, little learning effort is required to switch from managing traditional files and folders to managing a logical dataspace. However, we plan to explore other visualizations as part of future work.

The **Result Display** occupies the biggest area of the GUI. It is found in the middle and right of Figure 5.3. It shows the results returned by the iQL query typed on the Search-and-Query box or clicked in the iDM Graph Display. For each result, we show its name, its properties (e.g., `from` and `to` for emails), and a URI that locates the result in its underlying data source. In contrast to standard desktop search engines, we also find links to rich contextual information associated to that result. We describe how the user may benefit from that contextual information in the following.

5.6.2 Querying and Navigating the Dataspace

We present a typical `iMeMex` session performed by a user. The user may begin her exploration of the dataspace by navigating through the iDM Graph Display (left of Figure 5.3). That display unifies data from several data sources into one single browsing interface. In the example of Figure 5.3, we show data coming from the file system and from an email server. Furthermore, the iDM Graph Display not only allows the user to browse files and folders, but also the structural information inside the files. In Figure 5.3, we display the document structure for the file `CIDR 2007.tex` present in the `papers` folder in the filesystem. Note that the same functionality is available across data sources (e.g., email file attachment

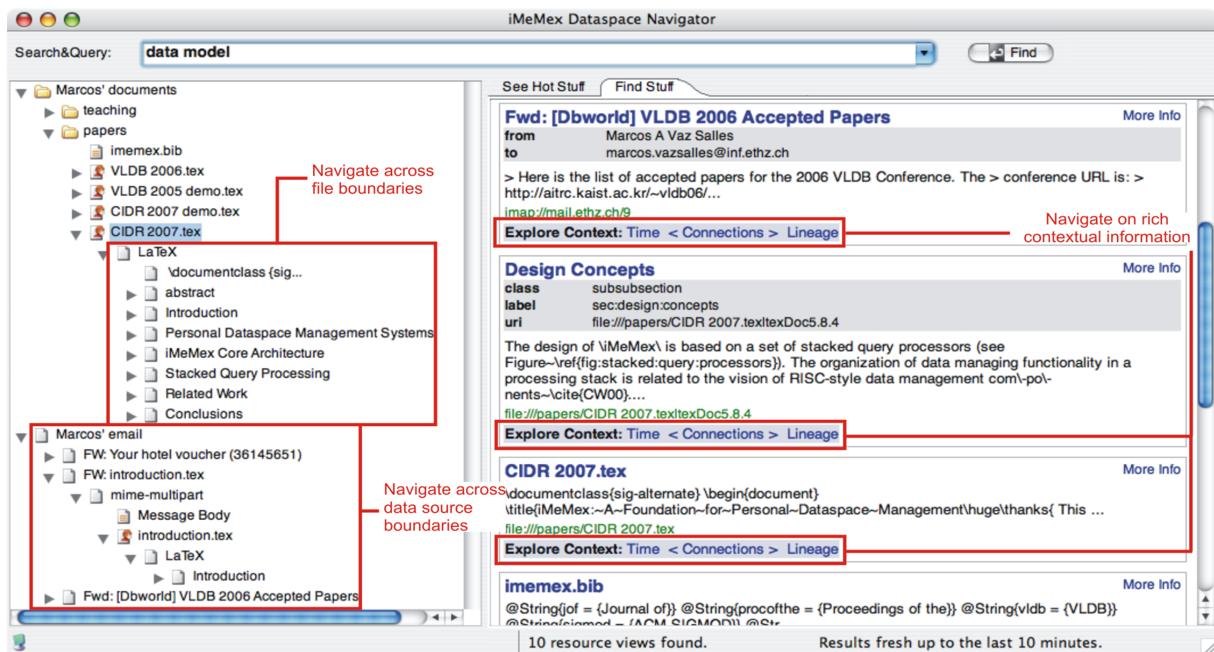


Figure 5.3: The iMeMex Dataspace Navigator displays a global view of the user's personal dataspace. It enables users to query that global view and to navigate on rich contextual information.

`introduction.tex` in Figure 5.3). When a user selects a node in the iDM Graph Display, details for that node are displayed in the Result Display.

We proceed by showing how a user may submit an iQL query to the system. Let us assume the user wants to start by entering a simple keyword such as `CIDR`. That keyword may be entered in the iQL Search-and-Query Box. Note that a typical user will enter only keywords in that box. Advanced users, however, may enter more complex expressions that restrict querying on certain subgraphs or attributes. For instance, the following query could also be entered in that box:

```
//ETH//*["dwh" and from~"Donald Kossmann"]
```

This query returns all resources that can be reached by graph traversal via a node named `ETH`, i.e., we generalize the *subfolder* relationship. In addition, all results should contain the keyword `dwh` and have an attribute `from` similar to `Donald Kossmann`.

The user entered the query above in order to obtain all the material she has received for the Data Warehouses lecture she is taking this semester. In order to avoid having to type such a complex query every time for this recurring information need, the user creates the following trail:

```
dw08⟶//ETH//*["dwh" and from~"Donald Kossmann"]
```

Now, whenever the user enters the keyword `dw08` in the iQL Search-and-Query Box, `iMeMex` will rewrite that query to return the results from the more complex query encoding schema information from the user's folder hierarchy and attributes. Note that the same trail could have been created automatically if

the user would have refined the query during a given session. For example, the user could have started her querying with the keyword `dw08` and progressively refined it until she obtained the complex query above. By mining the query log, we could have detected that pattern and suggested the trail to the user. This style of semi-automatic trail creation is an interesting avenue for future work.

For every query input by the user, the Result Display shows the top ranked results that were computed by `iMeMex`. For each result shown, the user may benefit from a rich source of contextual information that includes:

Connections. In the iDM Graph, a given query result has a set of incoming and a set of outgoing connections (edges) to other nodes. Navigating the neighborhood of query results in the iDM graph may help the user find related items, e.g., "what is the name of the person that sent me Donald Kossmann's Data Warehouses lecture notes?".

Time. The time context is useful to find resources "that were touched about the same time as the query result". Note that not only time, but any other ordered attribute recognized by an application may be used to display nodes that are *near* a given query result. Geographic locations could be used to attach information to maps, e.g., assigning pictures from the user's last holiday to his travel itinerary. This concept is similar to a drill-down in OLAP.

Lineage. Lineage refers to the history of data transformations that produced a given node. Users may be interested in obtaining previous versions of a given query result, e.g., to see how a project proposal looked one month ago. Furthermore, it may also be interesting to understand how a node was created. If a node *a* was copied from a node *b*, then previous versions of *b* may be of interest. For example, an error in a project proposal *a* may have been caused by an error in the proposal's template *b*. If the underlying data source tracks data transformations, then this information would be readily available for `iMeMex`. If that is not the case, we could try to detect candidate lineage relationships based on a content similarity predicate applied to a subset of the items in the dataspace or by leveraging reference reconciliation techniques for evolving objects [53, 56].

All of the rich contextual information above can be modeled in `iMeMex` through association trails. Some association trails, such as the ones explained above, could be shipped with `iMeMex` right from the start. Advanced users, however, may provide new association trails at any time. As a consequence, users may exploit other forms of rich context for their query results.

5.7 Related Work

The abstraction of *personal dataspaces* calls for a new kind of system that is able to support the entire personal dataspace of a user. We term this kind of system a *Personal DataSpace Management System (PDSMS)*. So far, no such system exists. We briefly review a few related solutions in this section. We focus on positioning these solutions in an overview of the design space for PDSMSs.

Figure 5.4 displays the design space of existing solutions for information management. The horizontal

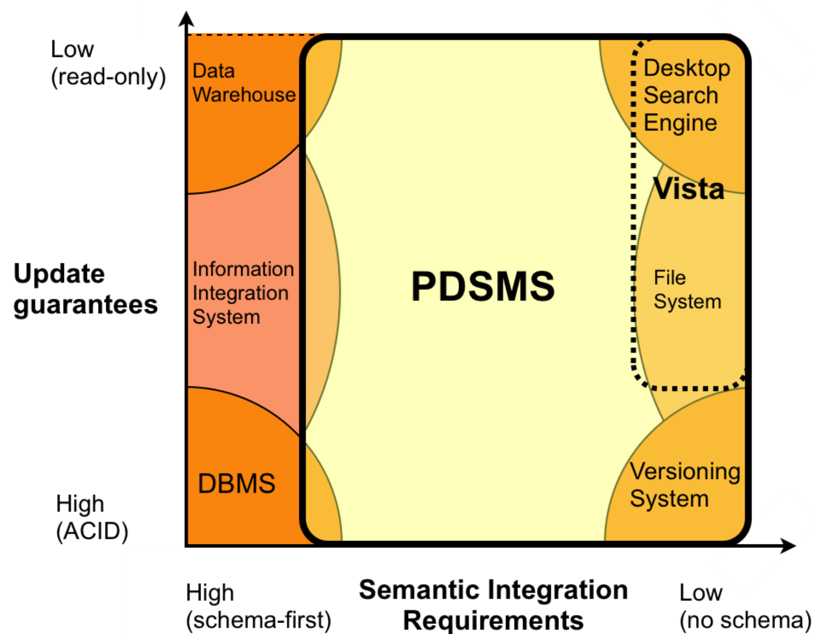


Figure 5.4: Design space of state-of-the-art information management systems: PDSMSs fill the gap between existing specialized systems.

axis displays requirements for semantic integration, while the vertical axis, in contrast to [65], displays the degree of update guarantees provided by different systems. One crucial aspect of dataspace management systems is their need to provide a pay-as-you-go information-integration framework that allows users to integrate data without defining a global schema. For this reason, a PDSMS occupies the design space in-between the two extremes “high semantic integration” (schema-first) and “low semantic integration” (no schema).

On the lower left corner of the mentioned design space we find DBMSs, which require high semantic integration efforts (up-front investment for schemas), but provide strong update guarantees (ACID). Examples of systems that attempted to apply DBMS technology to personal information include WinFS [165], MyLifeBits [15], Personal Data Manager [114], and Rufus [149]. These solutions, however, incur high costs for semantic integration and require full control of the data.

Strictly opposed to solutions based on DBMS technology, a desktop search engine (DSE) requires neither semantic integration nor full control of the data. On the other hand, these systems do not provide any update guarantees and do not allow structural information to be exploited for queries. Examples of such systems are Google Desktop [71], Apple Spotlight [152], Beagle [14], and Phlat [42]. The upper left corner of Figure 5.4 is occupied by data warehouses: These systems are optimized for read-only access. Furthermore, they require very high semantic integration efforts (integration of multiple schemas). Figure 5.4 also shows traditional information-integration systems (middle-left): These systems require high semantic integration investments and vary in terms of their update guarantees. Some systems, such as SEMEX [53] and Haystack [102], extend data-warehouse and information-integration technologies.

They extract information from desktop data sources into a repository and represent that information in a domain model (ontology). The domain model is a high-level mediated global schema over the personal information sources. Although systems such as SEMEX introduce techniques to deal with uncertainty in the mappings to this mediated schema [57], this schema-first approach makes it hard to integrate information in a pay-as-you-go fashion as required by a dataspace management system. In particular, extending the domain model is especially hard in those systems, while `iMeMex` offers an intuitive mechanism based on trails that allows users to reach the level of description of their personal domain that they find most convenient. In fact, all of the mentioned systems may be considered as applications on top of a data managing platform such as the `iMeMex` PDSMS.

The lower right corner of Figure 5.4 is occupied by versioning systems (e.g., Subversion, Perforce), which provide strong update guarantees but do not require semantic integration. File systems occupy the region on the middle-right, providing weaker update guarantees than versioning systems (e.g., recovery on metadata for journaling file systems). The operating system Windows Vista is also displayed as it provides some basic information management capabilities (dotted box on the upper right corner), covering functionalities offered by file systems and DSEs.

Figure 5.4 shows that a huge design space between the different extremes (sitting in the corners and along the margins) is not covered by current information management solutions. However, in order to be able to manage the entire dataspace of a user that space has to be covered. PDSMSs fill that gap. These systems cover the entire design space of information systems requiring medium-to-low semantic integration efforts. Furthermore, PDSMSs occupy the middle-ground between a read-only DSE (without any update guarantees) and a write-optimized DBMS (with strict ACID guarantees).

5.8 Conclusions

This chapter has advocated the design of a single system to master the personal information jungle [51]. The `iMeMex` Personal Dataspace Management System introduces a logical layer on top of the data sources that provides for full physical and logical personal-information independence. Our PDSMS is based on the OSGi framework and thus can be extended at (almost) any granularity. We have shown how `iMeMex` can be used by a search-and-browse GUI client to provide the user with a global view of her personal dataspace. For that purpose, we have illustrated through a use case how the user would interact with that GUI in a typical session. As future work, we plan to provide the upcoming features as listed in Section 5.5.2.

Chapter 6

Conclusion

Personal Information Management (PIM) draws increasing attention from end-users as more aspects of their lives go digital. Similarly, with the advent of a multitude of social-networking sites, users invest significant effort into online social interactions. In spite of those trends, current tools do not offer users an integrated view of their personal and social dataspaces. These tools either offer very simple services, such as the keyword search service offered by search engines, or require significant up-front investment before services are provided on the data, such as with traditional information-integration systems. In this thesis, we have investigated a novel breed of information-integration architecture that stands inbetween search engines and traditional information-integration systems. Our work is inspired by the vision for dataspace systems [65]: Simple services should be provided on all of the data from the start; however, the system should enable users to increase the level of integration of their dataspace gradually, in a pay-as-you-go fashion. In the following, we summarize the main contributions this thesis makes towards the development of the novel architecture of Personal Dataspace Management Systems (PDSMSs) in Section 6.1. We then conclude this thesis by discussing ongoing and future work in Section 6.2.

6.1 Summary

In this thesis, we made four main contributions towards the development of Personal Dataspace Management Systems, a new type of system capable of managing the whole personal and social dataspace of a user:

1. **`iMeMex` Data Model (iDM).** The first requirement for a PDSMS is to offer basic query services on all the data in the data sources from the start. In a personal- and social-dataspace scenario, this requirement implies dealing with a highly heterogeneous data mix (e.g., files, folders, emails, contacts, music) distributed among a variety of data sources (e.g., filesystems, email servers, databases, web sites). We have presented the `iMeMex` Data Model (iDM) as a simple, yet powerful, graph-based data model capable of representing that heterogeneous data mix found in personal dataspaces. On top of iDM, we have proposed the search-and-query language iQL to allow users to write intuitive keyword

searches with structural restrictions. As iDM clearly separates the logical data model from physical representation concerns, iDM graphs may be lazily computed, meaning that a system implementing iDM may operate by directly accessing the sources (mediation) or by creating local index structures and replicas (warehousing). In this way, it is possible to provide interactive processing times on iQL queries over iDM graphs.

2. **iTrails.** The second requirement for a PDSMS is to provide ways for users to increase the level of integration of data in their dataspace over time. To address that challenge, we have proposed iTrails. iTrails is a technique for pay-as-you-go information integration of a user's dataspace. With iTrails, users may gradually provide to the system lightweight integration hints, termed trails, which are then exploited to improve the precision and recall of query results. A trail models a relationship between two arbitrary sets of items in the dataspace. Trails include several relationships used in IR and information integration as special cases, such as synonyms, thesauri, language-agnostic search, semantic attribute matches, and multiple hierarchies. In addition, trails may specify equivalences among any two queries that encode a combination of path expressions and keyword searches. In spite of the expressiveness of trails, at no point a complete global schema for the dataspace needs to be specified. In addition, trails may be uncertain and it is possible to exploit that fact to scale trail-based query rewrites to thousands of trail definitions.

3. **Association Trails.** In personal and especially social dataspaces, it is frequently necessary to model not only set-level relationships as specified by iTrails, but also fine-grained relationships among individual instances. In order to model these fine-grained relationships that enable users to increase even further the level of integration of their dataspaces, we have proposed association trails. Association trails allow users to model an intensional graph of associations among instances in a dataspace. The connections in this intensional graph are exploited when users pose exploratory queries to the dataspace, such as neighborhood queries. In this way, not only primary query results are computed, but also important elements that are part of the context of those primary query results. Association trails not only permit users to specify such context declaratively, but also enable a system to use a number of different processing strategies when answering queries over the intensional graph created by them. These strategies may achieve over an order-of-magnitude gain in processing cost when compared to naively materializing the full extensional graph and then processing queries over it.

4. **iMeMex PDSMS Architecture.** During the development of this dissertation, we have worked on the design of iMeMex, the first generally available PDSMS. The iMeMex PDSMS is based on a layered and extensible architecture. The core idea of iMeMex is to provide an optional layer inbetween applications and data sources. At its current stage, this layer offers the advanced querying and pay-as-you-go information-integration services summarized above. Our system supports a number of data sources (e.g., filesystems, email servers, databases, RSS/ATOM feeds) and formats (e.g., XML, LaTeX, ZIP, PDF) commonly found in personal dataspaces. Extending the system to support new sources, formats, and even different indexing schemes is facilitated by our use of the OSGi framework. The iMeMex prototype is open-source and available at `http:www.imemex.org`.

6.2 Ongoing and Future Work

Below, we outline several interesting topics of ongoing work and discuss possibilities for future research.

Semi-automatic Creation of Dataspace Relationships. We have shown in Chapter 3 how to perform pay-as-you-go information integration with iTrails and how to scale iTrails to thousands of trail definitions. One important research question that is only partially solved is how to semi-automatically create the relationships handled by iTrails. As relationships encode data semantics, sharp improvements in precision and recall may be achieved when iTrails is provided with high-quality relationships. Possible approaches that could be employed to obtain those relationships include machine learning techniques and relevance feedback. We have done some initial work in that topic, with results summarized by Schmidt [146]. Basically, we have investigated how to apply semantic clustering techniques, in particular Latent Dirichlet Allocation (LDA) [18], in order to create a set of keyword-to-keyword trails. The core idea behind LDA is to find latent topics in a document collection. Each topic is a cluster of related words from the vocabulary. We may use this cluster to derive trails among these keywords. LDA does not help us, however, to obtain structural or schema relationships that may also be latent in the dataspace. One interesting direction to continue that work would be to investigate how to take structural information into account.

Dataspaces and Web Search. Web search engines face considerable challenges to integrate and query the increasing amount of structured data on the web. Among these challenges are integrating structured information in search results (e.g., Google Base or Google Music Search), tapping deep web data sources, supporting personalization, and exploiting community corpora (e.g., Wikipedia or del.icio.us). Rather than providing specialized solutions and implementing separate code in search engines for each of those challenges, we should ideally have a single, declarative, and powerful framework to model all of these different needs. We have discussed initial ideas on how iTrails could be used to tackle those challenges in a position statement by Vaz Salles et al. [142]. In particular, we have presented three architectures to integrate iTrails with web search engines: (1) client-side integration, (2) meta-search integration, and (3) search-engine integration. In Option (1), queries are rewritten with iTrails at the client side and then dispatched to a search engine (and potentially other sources). Option (2) introduces a meta-search engine layer inbetween clients and sources. Option (3) consists of integrating iTrails into the query processing stack of a state-of-the-art web search engine. We have taken initial implementation steps towards Option (1) by implementing a Google data-source plugin in `iMeMex`. As a consequence, we may have queries rewritten in `iMeMex` with trails and then (partially) dispatched to Google. However, challenges remain concerning the limited query model of Google, which implies that advanced structural constraints expressible in iQL cannot be directly translated to the search engine. One option here would be to provide direct access in `iMeMex` to deep-web front-ends that would be able to process these more complex query conditions. The reverse should also be considered: When remote sources have a query model more complex than iQL (e.g. XQuery), we could also investigate how to extend our pay-as-you-go information-integration techniques and our language to deal with these more sofisticated capabilities. How to do that in a scalable way, in particular, remains to be explored. Another direction of future work is to explore how to provide pay-as-you-go information integration for Options (2) and, especially, (3).

Dataspace Updates and Streaming. One important functionality that should be provided by a PDSMS, but was not explored in this dissertation, is the support for dataspace updates. As a first step, a PDSMS should be able to detect updates independently made to the data sources by applications bypassing the PDSMS. That functionality is similar to what is done today by search engines. In contrast, however, we would like the PDSMS not only to update its internal index structures, but also to provide some level of update guarantees on the data it manages. For example, a PDSMS could provide a weakened form of durability by monitoring and versioning portions of the dataspace. As a second step, the PDSMS could open interfaces to accept updates directly from applications and route those updates to the appropriate sources. The level of update guarantees that could be offered here depends on how much control data sources are willing to relinquish on their data.

Another interesting aspect related to dataspace updates is the handling of streaming sources. As we have seen, iDM is able to model content and data streams as infinite data. However, the pay-as-you-go information-integration techniques developed in this thesis to model dataspace relationships have assumed finite dataspaces. It would be interesting to investigate how to extend those techniques to infinite data and continuous queries. In addition, dataspace updates made directly to the sources may themselves be modeled as a stream from the point of view of the PDSMS monitoring these updates. It would also be interesting to investigate how to integrate the mechanisms used to process data streams and these dataspace updates.

Distributed and Intuitive Personal and Social Dataspaces. In this thesis, we have taken the perspective of a user wishing to manage her personal and social dataspace. When we have a community of users, however, we start having the notion of *nested* dataspaces. Portions of the dataspaces of separate users may be combined into group dataspaces and further combined into community dataspaces. Several issues arise in this scenario. One can argue which would be the most appropriate architecture to implement nested dataspaces. While a network of P2P instances is an option [72], there may be interesting possibilities coming from the movement towards cloud computing [5, 23]. These architectures bring challenges related to how to trade between consistency and performance, how to manage views at large scale, and how to protect privacy. Another interesting area for future research is on the design of intuitive visualizations and user interfaces for personal and social dataspaces. As recently argued for by Haas and Cousins [83], great benefits may result from the combination of effective data management and compelling human-computer interaction metaphors.

List of Tables

2.1 Important Resource View Classes to represent files and folders, relations, XML, data streams, and RSS . . . 19

2.2 Syntax of query expressions (Core grammar) . . . 28

2.3 Semantics of query expressions . . . 28

2.4 Characteristics of the personal dataset used in the evaluation . . . 29

2.5 Index sizes for the personal dataset . . . 31

2.6 iQL queries used in the evaluation . . . 32

3.1 Restrictions on content and group components of a resource view V_i. . . . 42

3.2 Logical algebra for query expressions. . . . 43

3.3 Datasets used in the experiments [MB]. . . . 59

3.4 Trails used in Scenario 1. . . . 60

3.5 Queries for trail evaluation in Scenario 1. . . . 60

3.6 Scenario 1 – Execution times of queries [sec]: iTrails with and without trail materialization compared to perfect query. . . . 62

4.1 Parameter settings used in the experiments . . . 101

4.2 iMeMex Basic Indexes: size and creation time. . . . 102

List of Figures

1.1 State-of-the-art Information-Integration Architectures 3

1.2 Architectural overview of a Personal Dataspace Management System (PDSMS). 6

1.3 Schematic depiction of the main contributions of this dissertation. 7

2.1 iDM represents heterogeneous personal information as a single resource view graph. The resource view graph represents the whole dataspace of a user 17

2.2 XML fragment represented as a resource view graph 22

2.3 A data stream is represented as a resource view graph 23

2.4 Indexing times [min] 31

2.5 Query-response times for queries Q1–Q8 [sec] 32

3.1 Running Example: a Dataspace consisting of four heterogeneous data sources. The data provided by the data sources is represented using the graph model of Chapter 2. The four components of the graph are disconnected. 39

3.2 Canonical form of `//home/projects//*["Mike"]`. 44

3.3 Trail Processing: (a) canonical form of Q_1 including matched subtrees $Q^M_{1\,\psi_7}$, $Q^M_{1\,\psi_8}$, and $Q^M_{1\,\psi_9}$, (b) left side of ψ_8, (c) transformation $Q^T_{1\,\psi_8}$, (d) final merged query $Q^*_{1\,\{\psi_8\}}$. 50

3.4 Algorithm 1 applied to Example 3.1. (a) original query, (b) rewritten query after one level, (c) after two levels. 53

3.5 Scenario 1 – Recall and Top-20 precision of query results: iTrails compared to baseline. . 61

3.6 Scenario 2 – Scalability of Trail Pruning: Effect and Sensitivity 63

3.7 Scenario 2 – Quality of Trail Pruning: Impact and Sensitivity 64

4.1 Running Example: (a) social networks today offer an extensional graph of users and manually-defined connections; (b) with association trails, a richer graph is defined intensionally. 72

4.2 Canonical query plan to process neighborhood queries 84

4.3 N-semi-join query plan to process neighborhood queries 86

4.4 Materialization of $A_*^{L.Q}$ and $A_*^{R.Q}$ with an OID List for association trails sameUniversity and graduatedSameYear. The materialization keeps a projection of the attributes referenced in the association trails' predicates. 88

4.5 Join materialization query plan to process neighborhood queries 89

4.6 Materialization of association-trail joins with an inverted list for association trails sameUniversity and graduatedSameYear 90

4.7 Grouping-compressed materialization of association trail joins for association trails sameUniversity and graduatedSameYear 92

4.8 Query materialization index for association trails sameUniversity and graduatedSameYear. Only tokens `fred:name` and `piano:hobbies` are displayed. 96

4.9 Query-response time versus number of association trails for methods without indexing . 103

4.10 Number of query results versus number of association trails 104

4.11 Query-dependent in-degree versus number of association trails 105

4.12 Query-response time versus selectivity of original query for methods without indexing . 106

4.13 Query-response time versus number of association trails for methods with indexing . . . 107

4.14 Indexing time versus number of association trails for methods with indexing 108

4.15 Index size versus number of association trails for methods with indexing 109

4.16 Number of join tuples (edges) versus number of association trails for join and grouping-compressed materializations 110

4.17 Query-response time versus selectivity of original query for methods with indexing . . . 112

4.18 Query-response time versus number of association trails for the hybrid approach 113

4.19 Indexing time versus number of association trails for the hybrid approach 114

4.20 Index size versus number of association trails for the hybrid approach 114

4.21 Query-response time versus selectivity of original query for the hybrid approach 116

5.1 `iMeMex` improves the current state-of-the-art in PIM by introducing logical layers that abstract from underlying subsystems, data formats, and devices. 125

5.2 `iMeMex` Component Architecture 127

5.3 The `iMeMex` Dataspace Navigator displays a global view of the user's personal dataspace. It enables users to query that global view and to navigate on rich contextual information. 132

5.4 Design space of state-of-the-art information management systems: PDSMSs fill the gap between existing specialized systems. 134

Bibliography

[1] D. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. Zdonik. Aurora: a New Model and Architecture for Data Stream Management. *The VLDB Journal*, 12(2):120–139, 2003.

[2] S. Abiteboul. On Views and XML. In *PODS*, 1999.

[3] S. Abiteboul, R. Agrawal, P. Bernstein, M. Carey, S. Ceri, W. B. Croft, D. DeWitt, M. Franklin, H. Garcia-Molina, D. Gawlick, J. Gray, L. Haas, A. Halevy, J. Hellerstein, Y. Ioannidis, M. Kersten, M. Pazzani, M. Lesk, D. Maier, J. Naughton, H.-J. Schek, T. Sellis, A. Silberschatz, M. Stonebraker, R. Snodgrass, J. Ullman, G. Weikum, J. Widom, and S. Zdonik. The Lowell Database Research Self Assessment. *The Computing Research Repository (CoRR)*, cs.DB/0310006, 2003.

[4] S. Abiteboul, A. Bonifati, G. Cobena, I. Manolescu, and T. Milo. Dynamic XML Documents with Distribution and Replication. In *ACM SIGMOD*, 2003.

[5] R. Agrawal, A. Ailamaki, P. Bernstein, E. Brewer, M. Carey, S. Chaudhuri, A. Doan, D. Florescu, M. Franklin, H. Garcia-Molina, J. Gehrke, L. Gruenwald, L. Haas, A. Halevy, J. Hellerstein, Y. Ioannidis, H. Korth, D. Kossmann, S. Madden, R. Magoulas, B. C. Ooi, T. O'Reilly, R. Ramakrishnan, S. Sarawagi, M. Stonebraker, A. Szalay, and G. Weikum. The Claremont Report on Database Research, 2008. Available at `http://db.cs.berkeley.edu/claremont`.

[6] S. Agrawal, S. Chaudhuri, and G. Das. DBXplorer: A System for Keyword-Based Search over Relational Databases. In *ICDE*, 2002.

[7] Y.-Y. Ahn, S. Han, H. Kwak, S. Moon, and H. Jeong. Analysis of Topological Characteristics of Huge Online Social Networking Services. In *WWW*, 2007.

[8] S. Amer-Yahia, L. V. S. Lakshmanan, and S. Pandit. FleXPath: Flexible Structure and Full-Text Querying for XML. In *ACM SIGMOD*, 2004.

[9] S. Amer-Yahia and M. Lalmas. XML Search: Languages, INEX and Scoring. *SIGMOD Record*, 36(7):16–23, 2006.

[10] Y. Arens, C. Y. Chee, C.-N. Hsu, and C. A. Knoblock. Retrieving and Integrating Data from Multiple Information Sources. *International Journal of Cooperative Information Systems*, 2(2):127–158, 1994.

[11] R. Baeza-Yates and B. Ribeiro-Neto. *Modern Information Retrieval*. Addison-Wesley, 1999.

[12] L. Ballesteros and W. B. Croft. Phrasal Translation and Query Expansion Techniques for Cross-language Information Retrieval. In *ACM SIGIR*, 1997.

[13] A.-L. Barabási and R. Albert. Emergence of Scaling in Random Networks. *Science*, 286(5439):509–512, 1999.

[14] Beagle. `http://beaglewiki.org`.

[15] G. Bell. Keynote: MyLifeBits: a Memex-Inspired Personal Store; Another TP Database. In *ACM SIGMOD*, 2005.

[16] O. Benjelloun, A. D. Sarma, A. Halevy, and J. Widom. ULDBs: Databases with Uncertainty and Lineage. In *VLDB*, 2006.

[17] D. Bhagwat, L. Chiticariu, W. C. Tan, and G. Vijayvargiya. An Annotation Management System for Relational Databases. *The VLDB Journal*, 14(4):373–396, 2005.

[18] D. Blei, A. Ng, and M. Jordan. Latent Dirichlet Allocation. *Journal of Machine Learning Research*, 3:993–1022, 2003.

[19] 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.

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

[21] P. Bohannon, E. Elnahrawy, W. Fan, and M. Flaster. Putting Context into Schema Matching. In *VLDB*, 2006.

[22] V. Borkar, M. Carey, D. Lychagin, T. Westmann, D. Engovatov, and N. Onose. Query Processing in the AquaLogic Data Services Platform. In *VLDB*, 2006.

[23] M. Brantner, D. Florescu, D. Graf, D. Kossmann, and T. Kraska. Building a Database on S3. In *ACM SIGMOD*, 2008.

[24] M. Brantner, S. Helmer, C.-C. Kanne, and G. Moerkotte. Full-fledged Algebraic XPath Processing in Natix. In *ICDE*, 2005.

[25] S. Brin and L. Page. The Anatomy of a Large-scale Hypertextual Web Search Engine. In *WWW*, 1998.

[26] N. Bruno, N. Koudas, and D. Srivastava. Holistic Twig Joins: Optimal XML Pattern Matching. In *ACM SIGMOD*, 2002.

[27] V. Bush. As We May Think. *Atlantic Monthly*, 1945.

[28] D. Calvanese. Keynote: Data Integration in Data Warehousing. In *CAiSE Workshops*, 2003.

[29] D. Carmel, Y. S. Maarek, M. Mandelbrod, Y. Mass, and A. Soffer. Searching XML Documents via XML Fragments. In *ACM SIGIR*, 2003.

[30] R. Cattell, D. Barry, M. Berler, J. Eastman, D. Jordan, C. Russell, O. Schadow, T. Stanienda, and F. Velez, editors. *The Object Data Standard: ODMG 3.0*. Morgan Kaufmann, 2000.

[31] D. Chakrabarti and C. Faloutsos. Graph mining: Laws, Generators, and Algorithms. *ACM Computing Surveys*, 38(1), 2006.

[32] S. Chakrabarti, S. Srivastava, M. Subramanyam, and M. Tiwari. Memex: A Browsing Assistant for Collaborative Archiving and Mining of Surf Trails. In *VLDB*, 2000. Demo Paper.

[33] K. C.-C. Chang and H. Garcia-Molina. Mind Your Vocabulary: Query Mapping Across Heterogeneous Information Sources. In *ACM SIGMOD*, 1999.

[34] L. Chen, A. Gupta, and M. E. Kurul. Stack-based Algorithms for Pattern Matching on DAGs. In *VLDB*, 2005.

[35] E. Chávez, G. Navarro, R. Baeza-Yates, and J. L. Marroquín. Searching in Metric Spaces. *ACM Computing Surveys*, 33(3):273–321, 2001.

[36] E. Codd. A Relational Model of Data for Large Shared Data Banks. *Communications of the ACM*, 13(6), 1970.

[37] S. Cohen, J. Mamou, Y. Kanza, and Y. Sagiv. XSEarch: A Semantic Search Engine for XML. In *VLDB*, 2003.

[38] W. Cohen, V. Carvalho, and T. Mitchell. Learning to Classify Email into "Speech Acts". In *EMNLP*, 2004.

[39] W. W. Cohen. Data Integration Using Similarity Joins and a Word-Based Information Representation Language. *ACM Transactions on Information Systems (TOIS)*, 18(3):288–321, 2000.

[40] G. P. Copeland and S. Khoshafian. A Decomposition Storage Model. In *ACM SIGMOD*, 1985.

[41] A. Culotta, R. Bekkerman, and A. McCallum. Extracting Social Networks and Contact Information from Email and the Web. In *CEAS*, 2004.

[42] E. Cutrell, D. Robbins, S. Dumais, and R. Sarin. Fast, Flexible Filtering with Phlat — Personal Search and Organization Made Easy. In *CHI*, 2006.

[43] U. Dayal. Processing Queries Over Generalization Hierarchies in a Multidatabase System. In *VLDB*, 1983.

[44] DBLP Dataset. `http://dblp.uni-trier.de/xml`.

[45] Apache Derby. http://db.apache.org/derby.

[46] D. DeWitt, J. Naughton, and J. Burger. Nested Loops Revisited. In *PDIS*, 1993.

[47] R. Dhamankar, Y. Lee, A. Doan, A. Halevy, and P. Domingos. iMAP: Discovering Complex Mappings between Database Schemas. In *ACM SIGMOD*, 2004.

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

[49] J.-P. Dittrich, P. M. Fischer, and D. Kossmann. AGILE: Adaptive Indexing for Context-Aware Information Filters. In *ACM SIGMOD*, 2005.

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

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

[52] A. Doan, R. Ramakrishnan, and S. Vaithyanathan. Managing Information Extraction. In *ACM SIGMOD*, 2006. Tutorial.

[53] X. Dong and A. Halevy. A Platform for Personal Information Management and Integration. In *CIDR*, 2005.

[54] X. Dong and A. Halevy. Malleable Schemas: A Preliminary Report. In *WebDB*, 2005.

[55] X. Dong and A. Halevy. Indexing Dataspaces. In *ACM SIGMOD*, 2007.

[56] X. Dong, A. Halevy, J. Madhavan, and E. Nemes. Reference Reconciliation in Complex Information Spaces. In *ACM SIGMOD*, 2005.

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

[58] S. T. Dumais, E. Cutrell, J. Cadiz, G. Jancke, R. Sarin, and D. Robbins. Stuff I've Seen: a System for Personal Information Retrieval and Re-Use. In *ACM SIGIR*, 2003.

[59] R. Elmasri and S. Navathe. *Fundamentals of Database Systems*. Addison-Wesley, 2000. Third Edition.

[60] Enron Dataset. http://www.cs.cmu.edu/∼enron.

[61] Facebook social networking site. http://www.facebook.com.

[62] R. Fagin. Combining Fuzzy Information: an Overview. *SIGMOD Record*, 31(2):109–118, 2002.

[63] D. Florescu and D. Kossmann. Storing and Querying XML Data using an RDMBS. *IEEE Data Engineering Bulletin*, 22(3):27–34, 1999.

[64] D. Florescu, D. Kossmann, and I. Manolescu. Integrating Keyword Search into XML Query Processing. *Computer Networks*, 33(1-6):119–135, 2000.

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

[66] M. Franklin, B. Jónsson, and D. Kossmann. Performance Tradeoffs for Client-Server Query Processing. In *ACM SIGMOD*, 1996.

[67] M. Franklin and S. Zdonik. "Data In Your Face": Push Technology in Perspective. In *ACM SIGMOD*, 1998.

[68] E. Freeman and D. Gelernter. Lifestreams: A Storage Model for Personal Data. *SIGMOD Record*, 25(1):80–86, 1996.

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

[70] V. Gaede and O. Günther. Multidimensional Access Methods. *ACM Computing Surveys*, 30(2):170–231, 1998.

[71] Google Desktop. `http://desktop.google.com`.

[72] R. Geambasu, M. Balazinska, S. Gribble, and H. Levy. HomeViews: Peer-to-Peer Middleware for Personal Data Sharing Applications. In *ACM SIGMOD*, 2007.

[73] F. Geerts, A. Kementsietsidis, and D. Milano. MONDRIAN: Annotating and Querying Databases through Colors and Blocks. In *ICDE*, 2006.

[74] O. Girard. Query Processing and Pay-as-you-go Information Integration in iMeMex. Master's thesis, ETH Zurich, 2007.

[75] R. Goldman and J. Widom. DataGuides: Enabling Query Formulation and Optimization in Semistructured Databases. In *VLDB*, 1997.

[76] Google. `http://www.google.com`.

[77] G. Graefe. Volcano - An Extensible and Parallel Query Evaluation System. *IEEE Transactions on Knowledge and Data Engineering*, 6(1):120–135, 1994.

[78] J. Graupmann, R. Schenkel, and G. Weikum. The SphereSearch Engine for Unified Ranked Retrieval of Heterogeneous XML and Web Documents. In *VLDB*, 2005.

[79] J. Gray, P. Sundaresan, S. Englert, K. Baclawski, and P. Weinberger. Quickly Generating Billion-Record Synthetic Databases. In *ACM SIGMOD*, 1994.

[80] R. Grishman. Information Extraction: Techniques and Challenges. In *SCIE*, 1997.

[81] H. Gupta, V. Harinarayan, A. Rajaraman, and J. Ullman. Index Selection for OLAP. In *ICDE*, 1997.

[82] Google Web Toolkit (GWT). `http://code.google.com/webtoolkit`.

[83] L. Haas and S. Cousins. Keynote: Information for People. In *ICDE*, 2007.

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

[85] A. Halevy. Answering Queries Using Views: A Survey. *The VLDB Journal*, 10(4):270–294, 2001.

[86] A. Halevy, N. Ashish, D. Bitton, M. Carey, D. Draper, J. Pollock, A. Rosenthal, and V. Sikka. Enterprise Information Integration: Successes, Challenges and Controversies. In *ACM SIGMOD*, 2005.

[87] A. Halevy, O. Etzioni, A. Doan, Z. Ives, J. Madhavan, L. McDowell, and I. Tatarinov. Crossing the Structure Chasm. In *CIDR*, 2003.

[88] A. Halevy, M. Franklin, and D. Maier. Principles of Dataspace Systems. In *PODS*, 2006.

[89] A. Halevy, A. Rajaraman, and J. Ordille. Data Integration: The Teenage Years. In *VLDB*, 2006. Ten-year best paper award.

[90] V. Harinarayan, A. Rajaraman, and J. Ullman. Implementing Data Cubes Efficiently. In *ACM SIGMOD*, 1996.

[91] S. Harizopoulos, V. Liang, D. Abadi, and S. Madden. Performance Tradeoffs in Read-Optimized Databases. In *VLDB*, 2006.

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

[93] Y. Huang, D. Govindaraju, T. Mitchell, V. Carvalho, and W. Cohen. Inferring Ongoing Activities of Workstation Users by Clustering Email. In *CEAS*, 2004.

[94] R. Huebsch, J. M. Hellerstein, N. Lanham, B. T. Loo, S. Shenker, and I. Stoica. Querying the Internet with PIER. In *VLDB*, 2003.

[95] iMeMex Project Web Site. `http://www.imemex.org`.

[96] E. H. Jacox and H. Samet. Spatial Join Techniques. *ACM Transactions on Database Systems (TODS)*, 32(1), 2007.

[97] H. V. Jagadish, L. V. S. Lakshmanan, M. Scannapieco, D. Srivastava, and N. Wiwatwattana. Colorful XML: One Hierarchy Isn't Enough. In *ACM SIGMOD*, 2004.

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

[99] W. Jones and H. Bruce. A Report on the NSF-Sponsored Workshop on Personal Information Management, Seattle, WA, 2005. `http://pim.ischool.washington.edu/final PIM report.pdf`.

[100] V. Kacholia, S. Pandit, S. Chakrabarti, S. Sudarshan, R. Desai, and H. Karambelkar. Bidirectional Expansion For Keyword Search on Graph Databases. In *VLDB*, 2005.

[101] J. Kang and J. F. Naughton. On Schema Matching with Opaque Column Names and Data Values. In *ACM SIGMOD*, 2003.

[102] D. R. Karger, K. Bakshi, D. Huynh, D. Quan, and V. Sinha. Haystack: A Customizable General-Purpose Information Management Tool for End Users of Semistructured Data. In *CIDR*, 2005.

[103] R. Kaushik, R. Krishnamurthy, J. F. Naughton, and R. Ramakrishnan. On the Integration of Structure Indexes and Inverted Lists. In *ACM SIGMOD*, 2004.

[104] M. Kersten, G. Weikum, M. Franklin, D. Keim, A. Buchmann, and S. Chaudhuri. Panel: A Database Striptease or How to Manage Your Personal Databases. In *VLDB*, 2003.

[105] J. Kleinberg. Authoritative Sources in a Hyperlinked Environment. *Journal of the ACM*, 46(5):604–632, 1999.

[106] D. Kossmann. The State of the Art in Distributed Query Processing. *ACM Computing Surveys*, 32(4):422–469, 2000.

[107] L. V. S. Lakshmanan, N. Leone, R. Ross, and V. S. Subrahmanian. ProbView: A Flexible Probabilistic Database System. *ACM Transactions on Database Systems (TODS)*, 22(3):419–469, 1997.

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

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

[110] N. Li, J. Hui, H.-I. Hsiao, and K. Beyer. Hubble: An Advanced Dynamic Folder Technology for XML. In *VLDB*, 2005.

[111] Y. Li, C. Yu, and H. V. Jagadish. Schema-Free XQuery. In *VLDB*, 2004.

[112] Apache Lucene. `http://lucene.apache.org/java/docs`.

[113] G. Luo. Efficient Detection of Empty-Result Queries. In *VLDB*, 2006.

[114] P. Lyngbaek and D. McLeod. A Personal Data Manager. In *VLDB*, 1984.

[115] J. Madhavan, S. Cohen, X. Dong, A. Halevy, S. Jeffery, D. Ko, and C. Yu. Web-Scale Data Integration: You Can Only Afford to Pay as You Go. In *CIDR*, 2007.

[116] D. Maier and L. Delcambre. Superimposed Information for the Internet. In *WebDB*, 1999.

[117] A. McCallum, A. Corrada-Emmanuel, and X. Wang. The Author-Recipient-Topic Model for Topic and Role Discovery in Social Networks: Experiments with Enron and Academic Email. Technical report, University of Massachusetts, Amherst, 12/2004.

[118] G. Miklau and D. Suciu. Containment and Equivalence for an XPath Fragment. In *PODS*, 2002.

[119] T. Milo, S. Abiteboul, B. Amann, O. Benjelloun, and F. D. Ngoc. Exchanging Intensional XML Data. In *ACM SIGMOD*, 2003.

[120] T. Milo and D. Suciu. Index Structures for Path Expressions. In *ICDT*, 1999.

[121] P. Mishra and M. H. Eich. Join Processing in Relational Databases. *ACM Computing Surveys*, 24(1):63–113, 1992.

[122] T. Mitchell. Keynote: Computer Workstations as Intelligent Agents. In *ACM SIGMOD*, 2005.

[123] M. Najork, H. Zaragoza, and M. Taylor. HITS on the Web: How Does It Compare? In *ACM SIGIR*, 2007.

[124] F. Naumann, U. Leser, and J. C. Freytag. Quality-driven Integration of Heterogenous Information Systems. In *VLDB*, 1999.

[125] T. Neumann. *Efficient Generation and Execution of DAG-Structured Query Graphs*. PhD thesis, University of Mannheim, 2005.

[126] T. Neumann and G. Weikum. RDF-3X: a RISC-style Engine for RDF. *JDMR (formerly Proc. VLDB)*, 1, 2008.

[127] W. S. Ng, B. C. Ooi, K.-L. Tan, and A. Y. Zhou. PeerDB: A P2P-based System for Distributed Data Sharing. In *ICDE*, 2003.

[128] Oscar: OSGi Implementation. `http://oscar.objectweb.org`.

[129] OSGi Alliance. `http://www.osgi.org`.

[130] Web Ontology Language (OWL). `http://www.w3.org/2004/OWL`.

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

[132] SIGIR PIM 2006. `http://pim.ischool.washington.edu/pim06home.htm`.

[133] CHI PIM 2008. `http://www.pim2008.org`.

[134] H. Pirahesh, J. M. Hellerstein, and W. Hasan. Extensible/Rule Based Query Rewrite Optimization in Starburst. In *ACM SIGMOD*, 1992.

[135] Y. Qiu and H.-P. Frei. Concept Based Query Expansion. In *ACM SIGIR*, 1993.

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

[137] R. Rashid, D. Reed, E. Felten, H. Schmidt, and E. Lawley. Panel: The Cyberspace Connection — Impact on Individuals, Society, and Research. In *Microsoft Research Faculty Summit*, 2008. Reported at: `http://www.pcworld.com/article/149028/2008/07/.html?tk=rss_news` and `http://research.microsoft.com/workshops/fs2008/agenda_mon.aspx`.

[138] Resource Description Framework (RDF). `http://www.w3.org/RDF`.

[139] Reiser FS. `http://www.namesys.com`.

[140] S. Robertson and S. Walker. Some Simple Effective Approximations to the 2-poisson Model for Probabilistic Weighted Retrieval. In *ACM SIGIR*, 1994.

[141] S. Robertson, H. Zaragoza, and M. Taylor. Simple BM25 Extension to Multiple Weighted Fields. In *CIKM*, 2004.

[142] M. A. V. Salles, J. Dittrich, and L. Blunschi. Adding Structure to Web Search with iTrails. In *IIMAS Workshop*, 2008.

[143] 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.

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

[145] R. Schenkel and M. Theobald. Feedback-Driven Structural Query Expansion for Ranked Retrieval of XML Data. In *EDBT*, 2006.

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

[147] D. Severance and G. Lohman. Differential Files: Their Application to the Maintenance of Large Databases. *ACM Transactions on Database Systems (TODS)*, 1(3), 1976.

[148] B. Shneiderman. Response Time and Display Rate in Human Performance with Computers. *ACM Computing Surveys*, 16(3):265–285, 1984.

[149] 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.

[150] C. Silverstein, M. Henzinger, H. Marais, and M. Moricz. Analysis of a Very Large AltaVista Query Log. Technical report, digitalSRC Technical Note 1998-014, 1998.

[151] V. Soloviev. A Truncating Hash Algorithm for Processing Band-Join Queries. In *ICDE*, 1993.

[152] Apple Mac OS X Spotlight. `http://www.apple.com/macosx/features/spotlight`.

[153] D. Srivastava and Y. Velegrakis. Intensional Associations Between Data and Metadata. In *ACM SIGMOD*, 2007.

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

[155] TelSearch. `http://tel.search.ch`.

[156] M. Theobald, H. Bast, D. Majumdar, R. Schenkel, and G. Weikum. TopX: Efficient and Versatile Top-k Query Processing for Semistructured Data. *The VLDB Journal*, 17(1):81–115, 2008.

[157] M. Theobald, R. Schenkel, and G. Weikum. An Efficient and Versatile Query Engine for TopX Search. In *VLDB*, 2005.

[158] S. Trissl and U. Leser. Fast and Practical Indexing and Querying of Very Large Graphs. In *ACM SIGMOD*, 2007.

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

[160] P. Valduriez. Join Indices. *ACM Transactions on Database Systems (TODS)*, 12(2):218–246, 1987.

[161] H. Wang, H. He, J. Yang, P. Yu, and J. X. Yu. Dual Labeling: Answering Graph Reachability Queries in Constant Time. In *ICDE*, 2006.

[162] Windows Desktop Search. `http://www.microsoft.com/windows/desktopsearch`.

[163] G. Weikum. An Experiment: How to Plan it, Run it, and Get it Published. In *CIDR*, 2005. Presentation.

[164] C. Weiss, P. Karras, and A. Bernstein. Hexastore: Sextuple Indexing for Semantic Web Data Management. *JDMR (formerly Proc. VLDB)*, 1, 2008.

[165] WinFS. `http://msdn.microsoft.com/data/WinFS`.

[166] Wikipedia Database Dump. `http://download.wikimedia.org`.

[167] Windows XP. `http://www.microsoft.com/windows/windows-xp`.

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

[169] Wordtracker. `http://www.wordtracker.com`.

[170] XQuery 1.0 and XPath 2.0 Data Model (XDM). `http://www.w3.org/TR/xpath-datamodel`.

[171] XML Information Set (Second Edition), 2004. `http://www.w3.org/TR/xml-infoset`.

[172] L. L. Yan, R. Miller, L. Haas, and R. Fagin. Data-Driven Understanding and Refinement of Schema Mappings. In *ACM SIGMOD*, 2001.

[173] C. Yu and H. V. Jagadish. Querying Complex Structured Databases. In *VLDB*, 2007.

[174] J. Zobel and A. Moffat. Inverted Files for Text Search Engines. *ACM Computing Surveys*, 38(2), 2006.

Curriculum Vitae: Marcos Vaz Salles

Member of the Association for Computing Machinery (ACM), of the Institute of Electrical and Electronics Engineers (IEEE), and of the Brazilian Computing Society (SBC)

Research Interests

- Pay-as-you-go Information Integration
- Dataspaces
- Personal Information Management

Affiliation during Doctoral Studies

Research Assistant (Ph.D. Student) in Computer Science
ETH Zurich
May, 2005 - November, 2008
Ph.D. Thesis: "Pay-as-you-go Information Integration in Personal and Social Dataspaces"

Education

M.S. in Informatics
PUC-Rio - Catholic University of Rio de Janeiro
March, 2002 - July, 2004
GPA: 1.0 / 1.0
Master's Thesis: "Autonomic Index Creation in Databases", supervised by Prof. Dr. Sérgio Lifschitz. The thesis involved implementation work in C/C++ with direct modifications to the PostgreSQL open-source DBMS.

B.S. in Computer Science
UNICAMP - State University of Campinas
February, 1995 - December, 1998
GPA: 0.92 / 1.0 (best GPA of his class)

Publications

Conference and Journal Papers

1. Jens Dittrich, Lukas Blunschi, Marcos Antonio Vaz Salles. Dwarfs in the Rearview Mirror: How Big are they really? International Conference on Very Large Databases (VLDB), August 2008, Auckland, New Zealand.
2. Marcos Antonio Vaz Salles, Jens Dittrich, Lukas Blunschi. Adding Structure to Web Search with iTrails. Workshop on Information Integration Methods, Architectures, and Systems (IIMAS; in conjunction with ICDE), April 2008, Cancún, México.
3. Jens Dittrich, Marcos Antonio Vaz Salles, Lukas Blunschi. Managing Personal Information Using iTrails. Personal Information Management Workshop (PIM; in conjunction with CHI), April 2008, Florence, Italy.
4. Marcos Antonio Vaz Salles, Jens-Peter Dittrich, Shant Kirakos Karakashian, Olivier René Girard, Lukas Blunschi. iTrails: Pay-as-you-go Information Integration in Dataspaces. International Conference on Very Large Databases (VLDB), September 2007, Vienna, Austria.
5. Cristian Duda, Jens-Peter Dittrich, Björn Jarisch, Donald Kossmann, Marcos Antonio Vaz Salles. Bringing Precision to Desktop Search: A Predicate-based Desktop Search Architecture (Poster Paper). IEEE International Conference on Data Engineering (ICDE), April 2007, Istanbul, Turkey.
6. Jens-Peter Dittrich, Lukas Blunschi, Markus Färber, Olivier René Girard, Shant Kirakos Karakashian, Marcos Antonio Vaz Salles. From Personal Desktops to Personal Dataspaces: A Report on Building the iMeMex Personal Dataspace Management System. GI-Fachtagung für Datenbanksysteme in Business, Technologie und Web (BTW), March 2007, Aachen, Germany.
7. Marcos Antonio Vaz Salles, Sérgio Lifschitz, Eduardo Maria Terra Morelli. Towards Autonomic Index Maintenance. Brazilian Symposium on Databases (SBBD), October 2006, Florianópolis, Brazil.
8. Marcos Antonio Vaz Salles, Jens-Peter Dittrich. iMeMex: A Platform for Personal Dataspace Management. VLDB Ph.D. Workshop, September 2006, Seoul, South Korea.
9. Jens-Peter Dittrich, Marcos Antonio Vaz Salles. iDM: A Unified and Versatile Data Model for Personal Information Management. International Conference on Very Large Databases (VLDB), September 2006, Seoul, South Korea.
10. S. Lifschitz, M. Salles. Autonomic Index Management (Poster Paper). IEEE International Conference on Autonomic Computing (ICAC), June 2005, Seattle.
11. R. Costa, S. Lifschitz, M. Noronha, M. Salles. Implementation of an Agent Architecture for Automated Index Tuning. International Workshop on Self-Managing Database Systems (SMDB; in conjunction with ICDE), April 2005, Tokyo, Japan.
12. R. Costa, S. Lifschitz, M. Salles. Index Self-tuning with Agent-based Databases. CLEI Electronic Journal - Special Issue of Best Papers presented at CLEI'2002, December 2003, Volume 6, Number 1, 22 pages, http://www.clei.cl/cleiej/paper.php?id=88.

13. M. Salles, S. Lifschitz. An Agent-based Architecture for the Index Self-tuning Problem in Relational Databases (in Portuguese), Ph.D. and M.S. Workshop, Brazilian Symposium on Databases (SBBD), October 2003, Manaus, Brazil.
14. M. Salles, C. Medeiros, F. Pires, J. Oliveira. Development of a Computer Aided Geographic Database Design System, Brazilian Symposium on Databases (SBBD), October 1998, Maringá, Brazil.
15. M. Salles, C. Medeiros, F. Pires, J. Oliveira. Development of a Computer Aided Geographic Database Design System (in Portuguese), GIS Brazil 98, May 1998, Curitiba, Brazil.

Demos

1. Lukas Blunschi, Jens-Peter Dittrich, Olivier René Girard, Shant Kirakos Karakashian, Marcos Antonio Vaz Salles. A Dataspace Odyssey: The iMeMex Personal Dataspace Management System. Conference on Innovative Data Systems Research (CIDR), January 2007, Asilomar, USA.
2. Jens-Peter Dittrich, Marcos Antonio Vaz Salles, Donald Kossmann, Lukas Blunschi. iMeMex: Escapes from the Personal Information Jungle. International Conference on Very Large Databases (VLDB), August 2005, Trodheim, Norway.
3. M. Salles, S. Lifschitz. A Software Agent for Index Creation on PostgreSQL (in Portuguese). I Demo Session - Brazilian Symposium on Databases (SBBD), October 2004, Brasília, Brazil. This demo was awarded the 2nd place prize among other 10 demos presented at the session.

Tech Reports

1. Jens-Peter Dittrich, Cristian Duda, Björn Jarisch, Donald Kossmann, Marcos Antonio Vaz Salles. Bringing Precision to Desktop Search: A Predicate-based Desktop Search Architecture. Tech Report, ETH Zurich, November 2006.
2. S. Lifschitz, A. Milanés, M. Salles. State of the Art in Relational DBMS Self-tuning (in Portuguese). Tech Report PUC-RioInf.MCC35/04, PUC-Rio, October 2004.

Selected Presentations

1. Pay-as-you-go Information Integration in Personal and Social Dataspaces. PhD Defense, November 2008, Zurich, Switzerland.
2. iTrails: Pay-as-you-go Information Integration in Dataspaces. International Conference on Very Large Databases (VLDB), September 2007, Vienna, Austria.
3. From Personal Desktops to Personal Dataspaces: A Report on Building the iMeMex Personal Dataspace Management System. GI-Fachtagung für Datenbanksysteme in Business, Technologie und Web (BTW), March 2007, Aachen, Germany.
4. iMeMex: A Platform for Personal Dataspace Management. VLDB Ph.D. Workshop, September 2006, Seoul, South Korea.

5. Autonomic Index Creation in Databases. DBIS/IKS Groups' Research Seminar at ETH Zurich, May 2005, Zurich, Switzerland.

Previous Work Experience

- Software Architect
  GAPSO Advanced Planning
  A software house specialized in Advanced Planning Systems (APS)
  August, 2004 - March, 2005
  Started a software development process improvement program in the company. Acted as technical lead on a helicopter logistics planning project for Petrobras, Brazil's largest oil company.
  Technologies used: Oracle Database, ERWin, Rational Rose, Java, XML, C/C++.

- Founding Partner
  INMETRICS
  A start-up specialized in IT systems monitoring
  January, 2002 - May, 2004
  Acted on the design and implementation of monitoring tools for databases and operating systems. Worked on adapting the company's technology to the IT infrastructure used in Brazilian financial institutions, with an emphasis on settlement systems. Worked on the adoption of the Rational Unified Process (RUP) for the company's software development process. Participated on an Oracle Forms migration and development project for a large telecom firm. Worked on pre-sales of HP and Oracle solutions on Cobra Technologies, a systems integrator owned by one of Brazil's largest public banks.
  Technologies used: Oracle Database, Oracle Forms, Oracle Reports, Rational Rose, Java, XML.

- Systems Analyst
  Software Design Informática
  A medium-sized systems integrator and Oracle Partner
  March, 1999 - January, 2002
  Lead the company's Financial Market Solutions area in the Rio de Janeiro's office. Planned, installed and configured systems for savings accounts, loans, open market, payments, transfers and settlements in several investment banks. Acted on process redefinition to improve maintenance and installation activities for the company's banking systems. Participated on the development of Internet and Email Banking systems. Lead a team of analysts on the installation of a settlement solution in response to changes on the sector's legal requirements.
  Technologies used: Oracle Database, Oracle Designer, Oracle Forms, Oracle Reports, ERWin, Java, XML.

- Programmer
  Software Design Informática

A medium-sized systems integrator and Oracle Partner
June, 1998 - March, 1999
Developed tailor-made systems for clients such as HBO Brazil and Eaton Transmissions.
Technologies used: Oracle Database, Oracle Designer, Oracle Forms, Oracle Reports.

Teaching and Mentoring Experience

- Co-supervision of ten semester theses and six master theses
  ETH Zurich, 2005 - 2008

- Teaching Assistant of the Information Systems Laboratory lecture
  ETH Zurich, FS 2007

- Teaching Assistant of the Data Warehouses lecture
  ETH Zurich, SS 2007 and SS 2006

- Teaching Assistant of the Architecture and Implementation of Database Systems lecture
  ETH Zurich, WS 2006/2007 and WS 2005/2006

- Lecturer and Coordinator of the Database Tuning and Administration I and II courses
  Extension Program (CCE) - PUC-Rio, 2002 - 2005

- Lecturer of the "Oracle Administration" and "Oracle Tuning" courses
  Petrobras Corporate University - in partnership with PUC-Rio
  These courses were repeated four times during 2004 and once in 2005

Other Professional Activities

- External reviewer for the conferences: ICSOC 2005, ICDE 2006, ICDCS 2006, SIGMOD 2006, CIKM 2007, SIGMOD 2007, VLDB 2007 - Demo Track, EDBT 2008, SIGMOD 2008 - Repeatability Committee, EDBT 2009.

- PC member for IIMAS 2008.

Selected Courses and Certifications

- Didactic Workshop
  This three day course included video footage aiming to improve teaching skills.
  ETH Zurich, October 2006

- Introduction to CMMI: Staged and Continuous
  This SEI-certified course enables participation in CMMI appraisal teams.
  Rio de Janeiro, January 2005

Languages

- Portuguese: Native
- English: Proficient
  Obtained certificates from Michigan University, USA, and from Cambridge University, England (CAE).
- German (A2-B1), French, Spanish (A1): Beginner