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

Marcos Antonio Vaz Salles Jens-Peter Dittrich Shant Kirakos Karakashian
Olivier René Girard Lukas Blunschi
ETH Zurich
8092 Zurich, Switzerland
dbis.ethz.ch | iMeMex.org

ABSTRACT

Dataspace management has been recently identified as a new agenda for information management [17, 22] and information integration [23]. 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 [17], as time and effort (pay) are needed over time (go) to provide integration semantics. The benefits are better query results (gain). This paper is the first to explore 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 efficiently implemented 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.

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 [23, 27, 34, 32], data warehousing [7], enterprise information integration [24], and also P2P systems [39, 33]. 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 [10, 40]). 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] [41]). □

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 [6]), there is still significant work to be done to devise good ranking methods for a wider range of information integration scenarios.

In this paper, we explore a new point in the design space in-between 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' (*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. Our work is an important step towards realizing the vision of a new kind of information integration architecture called *dataspace management system* [17]. □

1.1 Motivating Example

Figure 1 shows the dataspace scenario used throughout this paper as a running example. The example consists of four data sources:

*This work was partially supported by the Swiss National Science Foundation (SNF) under contract 200021-112115.

[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.

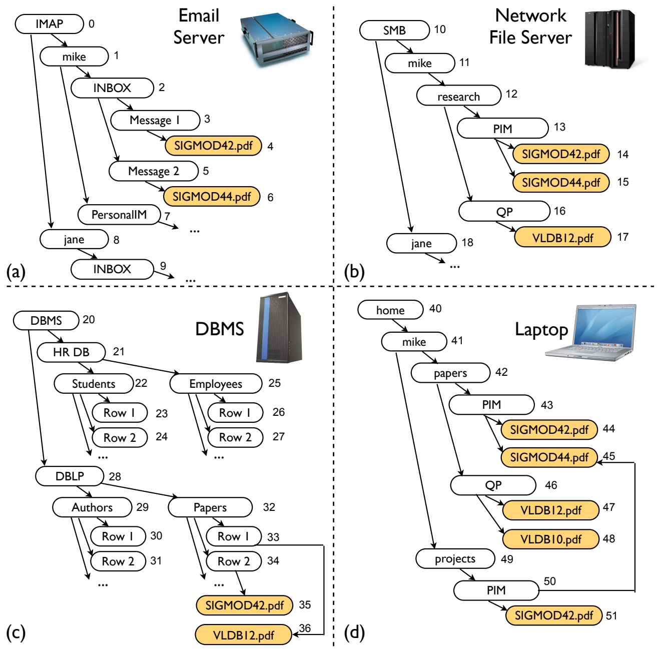


Figure 1: Running Example: a Dataspace consisting of four heterogeneous data sources. The data provided by the data sources is represented using our graph model proposed in [11]. The four components of the graph are disconnected.

(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 a graph model as proposed in [11]. For instance, in Figure 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 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 1(b,c,d), files, folders, databases, tables, and rows are represented by separate nodes. 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 1.

EXAMPLE 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 above schema knowledge.
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 paper 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 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. □

1.2 Contributions

In summary, this paper 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 a complete global schema for the dataspace needs 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 rewrite 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 [4]. 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 paper is structured as follows. Section 2 introduces the data and query model as well as the query algebra used throughout this paper. Section 3 presents the iTrails technique and discusses how to define trails. Section 4 presents iTrails query processing algorithms and a complexity analysis. Section 5 presents trail-rewrite pruning techniques. Section 6 presents the experimental evaluation of our approach. Section 7 reviews related work and its relationship to iTrails. Finally, Section 8 concludes this paper.

2. DATA AND QUERY MODEL

2.1 Data Model

The data model used in this paper is a simplified version of the generic iMeMex Data Model (iDM) [11]. As illustrated in the introduction and in Figure 1, iDM represents every structural component of the input data as a node. Note that, using iDM, all techniques presented in this paper can easily be adapted to relational and XML data.

DEFINITION 1 (DATA MODEL) *We assume that all data is represented by a logical graph G. We define $G := (RV, E)$ as a set of nodes $RV := \{V_1, .., V_n\}$ where the nodes $V_1, .., V_n$ are termed resource views. E is a sequence of ordered pairs (V_i, V_j) of resource views representing directed edges from V_i to V_j. A resource view V_i has three components: name, tuple, and content.*

Component of V_i	Definition
V_i`.name`	Name (string) of the resource view
V_i`.tuple`	Set of attribute value pairs $(\langle att_0, value_0\rangle, \langle att_1, value_1\rangle, \dots)$.
V_i`.content`	Finite byte sequence of content (e.g. text)

Table 1: Components of a resource view V_i

If a resource view V_j is reachable from V_i by traversing the edges in E, we denote this as $V_i \rightsquigarrow V_j$. □

As shown in [11], we assume that the graph G may represent a variety of different data including XML, Active XML [31], relational data, file&folder graphs, and structural elements inside files (e.g., LaTeX, XML). How this graph of data is generated or materialized by a system is orthogonal to the work presented here. Please see [11] for details.

2.2 Query Model

DEFINITION 2 (QUERY EXPRESSION) *A query expression Q selects a subset of nodes $R := Q(G) \subseteq G.RV$.* □

DEFINITION 3 (COMPONENT PROJECTION) *A component projection $C \in \{$`.name`, `.tuple.`$<att_i>$, `.content`$\}$ obtains a projection on the set of resource views selected by a query expression Q, i.e., a set of components $R' := \{V_i.C \mid V_i \in Q(G)\}$.* □

For instance, in Figure 1(c), if a query expression selects $R = \{33\}$, then R`.name` returns the string "Row 1".

Our query expressions are similar in spirit to NEXI [41]. Tables 2 and 3 show the syntax and semantics of the query expressions used throughout this paper.

2.3 Query Algebra

Query expressions are translated into a logical algebra with the operators briefly described in Table 4. As the query language we use is close in spirit to NEXI [41] (and therefore, XPath), our algebra resembles a path expression algebra [5]. In contrast to [5],

```
QUERY_EXPRESSION := (PATH | KT_PREDICATE) (UNION 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: Syntax of query expressions (Core grammar)

Query Expression	Semantics
`//*`	$\{V \mid V \in G.RV\}$
`a`	$\{V \mid V \in G.RV \wedge \text{‘}a\text{’} \subseteq V.content\}$
`a b`	$\{V \mid V \in G.RV \wedge \text{‘}a\text{’} \subseteq V.content \wedge \text{‘}b\text{’} \subseteq V.content\}$
`//A`	$\{V \mid V \in G.RV \wedge V.name = \text{‘}A\text{’}\}$
`//A/B`	$\{V \mid V \in G.RV \wedge V.name = \text{‘}B\text{’}$ $\wedge(\exists(W,V) \in G.E : W.name = \text{‘}A\text{’}\}$
`//A//B`	$\{V \mid V \in G.RV \wedge V.name = \text{‘}B\text{’}$ $\wedge(\exists(W,Z_1),(Z_1,\dots),..,(..,Z_n),(Z_n,V) \in G.E : W.name = \text{‘}A\text{’}\}$
`b=42`	$\{V \mid V \in G.RV \wedge \exists V.tuple.b : V.tuple.b = 42\}$
`b=42 a`	:= `b=42` ∩ `a`
`//A/B[b=42]`	:= `//A/B` ∩ `b=42`

Table 3: Semantics of query expressions

Operator	Name	Semantics
G	All resource views	$\{V \mid V \in G.RV\}$
$\sigma_P(I)$	Selection	$\{V \mid V \in I \wedge P(V)\}$
$\mu(I)$	Shallow unnest	$\{W \mid (V,W) \in G.E \wedge V \in I\}$
$\omega(I)$	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 4: Logical algebra for query expressions

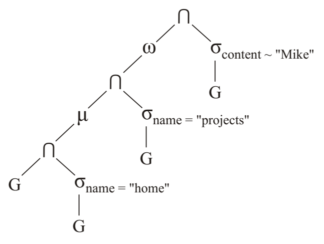
our algebra operates not only on XML documents but on a general graph model that represents a heterogeneous and distributed dataspace [11]. We will use the algebraic representation to discuss the impact of trails on query processing. We represent every query by a *canonical form*.

DEFINITION 4 (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. 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 on the right. 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. iTRAILS

In this section, we present our iTrails technique. We first present a formal definition of trails. Then we present several use cases (Section 3.2). After that, we discuss how to obtain trails (Section 3.3). Finally, we show how to extend trails to probabilistic trails (Section 3.4) and scored trails (Section 3.5).

3.1 Basic Form of a Trail

In the following, we formally define *trails*.

DEFINITION 5 (TRAIL) *A unidirectional trail is denoted as*

$$\psi_i := Q_L[.C_L] \longrightarrow Q_R[.C_R].$$

This means that the query (resp. component projection) on the left $Q_L[.C_L]$ induces the query (resp. component projection) on the right $Q_R[.C_R]$, i.e., whenever we query for $Q_L[.C_L]$ we should also query for $Q_R[.C_R]$. A bidirectional trail is denoted as

$$\psi_i := Q_L[.C_L] \longleftrightarrow Q_R[.C_R].$$

The latter also means that the query on the right $Q_R[.C_R]$ induces the query on the left $Q_L[.C_L]$. The component projections C_L and C_R should either appear on both sides of the trail or on none. □

Before we formally define the impact of a trail on a query Q (see Section 4), we present some trail use cases.

3.2 Trail Use Cases

USE CASE 1 (FUNCTIONAL EQUIVALENCE) Assume the following trail definitions:

$\psi_1 :=$ `//*.tuple.date`$\longrightarrow$`//*.tuple.modified`
$\psi_2 :=$ `//*.tuple.date`$\longrightarrow$`//*.tuple.received`
$\psi_3 :=$ `yesterday`$\longrightarrow$`date=yesterday()`
$\psi_4 :=$ `pdf`$\longrightarrow$`//*.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 [1], we do the opposite: we extend the original query to a more complex one. This solves Example 1. Note that in contrast to [28] we do not depend on heuristics to detect meaningfully related substructures. Instead, we explore the hints, i.e., trails, that were defined by the user or administrator in advance. □

USE CASE 2 (TYPE RESTRICTION) Assume a trail definition

$\psi_5 :=$ `email`$\longrightarrow$`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 (e.g. [39]), 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 (SEMANTIC SEARCH) Semantic search is just one special case of iTrails. Semantic search includes query expansions using dictionaries, e.g., for *language agnostic search* [2], using *thesauri* like wordnet [44], or *synonyms* [35]. The benefit of our technique is that it is not restricted to a specific use case. For example, a trail `car`$\longrightarrow$`auto` could be automatically generated from [44]. 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 4 (HIDDEN-WEB DATABASES) If data sources present in the dataspace are mediated, e.g. hidden-web databases, then trails may be used to integrate these sources. Consider the trail:

$\psi_6 :=$ `train home` $\longrightarrow$
```
        //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 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 taylor 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&drop frontend, (2) Create trails based on user-feedback in the spirit of relevance feedback in search engines [38], (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 like wordnet [44] to extract equivalences among keyword queries (see Use Case 3). A second example is to leverage information extraction techniques [20] 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 [36]. As an example of (4), sites in the style of bookmark-sharing sites like `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, blogs, etc), 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 paper 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 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.

3.4 Probabilistic Trails

To model trail quality we relax Definition 5 by assigning a probability value p to each trail definition.

DEFINITION 6 (PROBABILISTIC TRAIL) *A probabilisitc trail assigns a probability value $0 \leq p \leq 1$ to a trail definition:*

$$\psi_i := Q_L[.C_L] \xrightarrow[p]{} Q_R[.C_R].$$ □

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 analog to the problem of obtaining *data* probabilities [26] in probabilistic databases [3]. Therefore, we believe that techniques from this domain could be adapted to obtain trail probabilities. Furthermore, methods from automatic schema matching [36] could also be leveraged for that purpose. In Section 5 we will show how probabilistic trail definitions help us to accelerate the trail rewrite process.

3.5 Scored Trails

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

DEFINITION 7 (SCORED TRAIL) *A scored trail assigns a scoring factor $sf \geq 1$ to a trail definition:*

$$\psi_i := Q_L[.C_L] \xrightarrow[sf]{} Q_R[.C_R]. \qquad \Box$$

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. Thus, scored trails will 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 are obtained in the same way as trail probabilities (see Section 3.4).

4. iTRAILS QUERY PROCESSING

This section discusses how to process queries in the presence of trails. After that, Section 5 extends these techniques to allow pruning of trail applications.

4.1 Overview

iTrails query processing 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_i by ψ_i^L and ψ_i^R, respectively. The corresponding query and component projections are denoted by $\psi_i^{L.Q}$, $\psi_i^{L.C}$, and $\psi_i^{R.Q}$, $\psi_i^{R.C}$, respectively. If a query Q matches a trail ψ_i we denote the match as $Q^M_{\psi_i}$.
2. **Transformation.** If the left side of a trail ψ_i was matched by a query Q, the right side of that trail will be used to compute the transformation of Q. It is denoted $Q^T_{\psi_i}$.
3. **Merging.** Merging refers to composing the transformation $Q^T_{\psi_i}$ with the original query Q into a new query. The new query $Q^*_{\{\psi_i\}}$ *extends* the semantics of the original query based on the information provided by the trail definition.

4.2 Matching

We begin by defining the conditions for matching trails with and without component projections to a query. For the remainder of this paper, query containment is understood as discussed in [30].

DEFINITION 8 (TRAIL MATCHING) *A trail ψ_i matches a query Q whenever its left side query, $\psi_i^{L.Q}$, is contained in a query subtree Q_S of the canonical form $\Gamma(Q)$ (see Section 2.3). We denote this as $\psi_i^{L.Q} \subseteq Q_S$. Furthermore, Q_S must be maximal, i.e., there must be no subtree $Q^{\hat{S}}$ of Q, such that $\psi_i^{L.Q} \subseteq Q^{\hat{S}}$ and Q_S is a subtree of $Q^{\hat{S}}$. If ψ_i does not contain a component projection, we require Q_S not to contain $\psi_i^{R.Q}$, i.e., $\psi_i^{R.Q} \not\subseteq Q_S$. We then take $Q^M_{\psi_i} := Q_S$. On the other hand, i.e., if ψ_i contains a component projection, we require that the component projection $\psi_i^{L.C}$ be referenced in the query in a selection by an operator immediately after Q_S in $\Gamma(Q)$. The matching subtree $Q^M_{\psi_i}$ is then obtained by extending Q_S by the portion of the query referencing the component projection $\psi_i^{L.C}$.* $\Box$

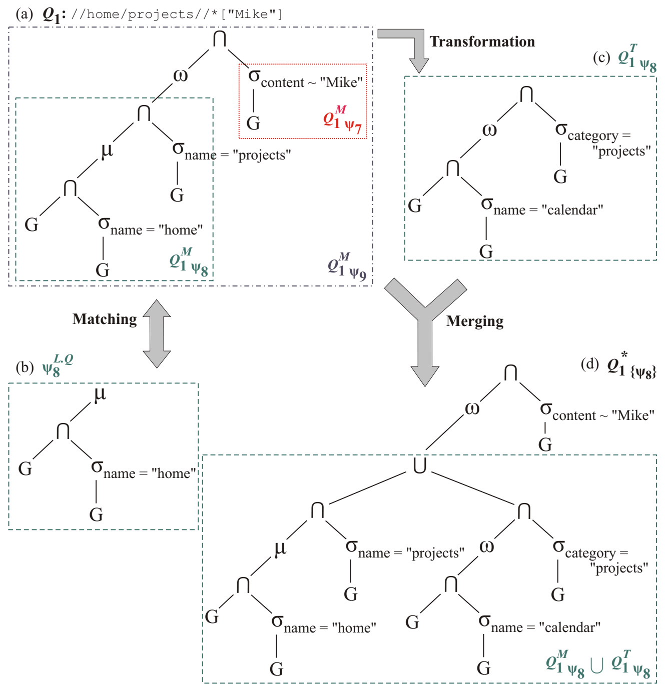

(a) Q_1: //home/projects//*["Mike"]
Transformation
(c) $Q^T_{1\ \psi_8}$
$\cap$
ω
$\sigma_{\text{content ~ "Mike"}}$
G
$Q^M_{1\ \psi_7}$
μ
$\sigma_{\text{name = "projects"}}$
$\sigma_{\text{name = "home"}}$
$Q^M_{1\ \psi_8}$
$Q^M_{1\ \psi_9}$
$\sigma_{\text{category = "projects"}}$
$\sigma_{\text{name = "calendar"}}$
Matching
Merging
(b) $\psi_8^{L.Q}$
(d) $Q^*_{1\ \{\psi_8\}}$
$\cup$
$Q^M_{1\ \psi_8} \cup Q^T_{1\ \psi_8}$

Figure 2: 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 $Q^*_{1\ \{\psi_8\}}$.

EXAMPLE 3 Consider query $Q_1 :=$ `//home/projects//*["Mike"]`. Its canonical form is shown in Figure 2(a). Suppose we have a set of four trails, $\Psi = \{\psi_7, \psi_8, \psi_9, \psi_{10}\}$ with

$\psi_7 :=$ `Mike`$\longrightarrow$`Carey`

$\psi_8 :=$ `//home/*.name`$\longrightarrow$`//calendar//*.tuple.category`

$\psi_9 :=$ `//home/projects/OLAP//*["Mike"]`$\longrightarrow$
`//imap//*["OLAP" "Mike"]`

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

Here, 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 $Q^M_{1\ \psi_7}$ and $Q^M_{1\ \psi_8}$ in Figure 2(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 is illustrated by ψ_9, which matches Q_1 as $\psi_9^{L.Q} \subseteq Q_1 = Q^M_{1\ \psi_9}$. On the other hand, 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 $\psi_{10}^{R.Q}$. $\Box$

In order to test whether a trail matches, we have to test path query containment. A sound containment test algorithm based on tree pattern homomorphisms is presented in [30] and we followed their framework in our implementation. The algorithm is polynomial in the size of the query trees being tested.

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 9 (TRAIL TRANSFORMATION) *Given a query expression Q and a trail ψ_i without component projections, we compute the transformation $Q^T_{\psi_i}$ by setting $Q^T_{\psi_i} := \psi_i^{R.Q}$. For a trail ψ_j with component 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 $\psi_j^{L.C}$ for $\psi_j^{R.C}$.* □

EXAMPLE 4 To illustrate Definition 9, consider again the trail ψ_8 and query Q_1 of Example 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 2(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 2(c) shows the resulting transformation $Q^T_{1\,\psi_8}$ = `//calendar//*[category="projects"]`. □

4.4 Merging

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

DEFINITION 10 (TRAIL MERGING) *Given a query Q and a trail ψ_i, the merging $Q^*_{\{\psi_i\}}$ is given by substituting $Q^M_{\psi_i}$ for $Q^M_{\psi_i} \cup Q^T_{\psi_i}$ in $\Gamma(Q)$.* □

EXAMPLE 5 Recall that we show the match of trail ψ_8 to query Q_1, denoted $Q^M_{1\,\psi_8}$, in Figure 2(a). Figure 2(c) shows the transformation $Q^T_{1\,\psi_8}$. If we now substitute $Q^M_{1\,\psi_8}$ for $Q^M_{1\,\psi_8} \cup Q^T_{1\,\psi_8}$ in $\Gamma(Q_1)$, we get the merged query $Q^*_{1\,\{\psi_8\}}$ as displayed in Figure 2(d). The new query corresponds to:

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

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 $Q^*_{\{\psi_i\}}$ always contains Q, then, if ψ_i originally matched Q, this means that ψ_i must match $Q^*_{\{\psi_i\}}$ (appropriately rewritten to canonical form). If we then merge $Q^*_{\{\psi_i\}}$ to obtain $Q^*_{\{\psi_i,\psi_i\}}$, once again ψ_i will match $Q^*_{\{\psi_i,\psi_i\}}$ and so on indefinitely. That also happens for different trails with mutually recursive patterns.

Multiple Match Colouring 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 already touched those nodes.

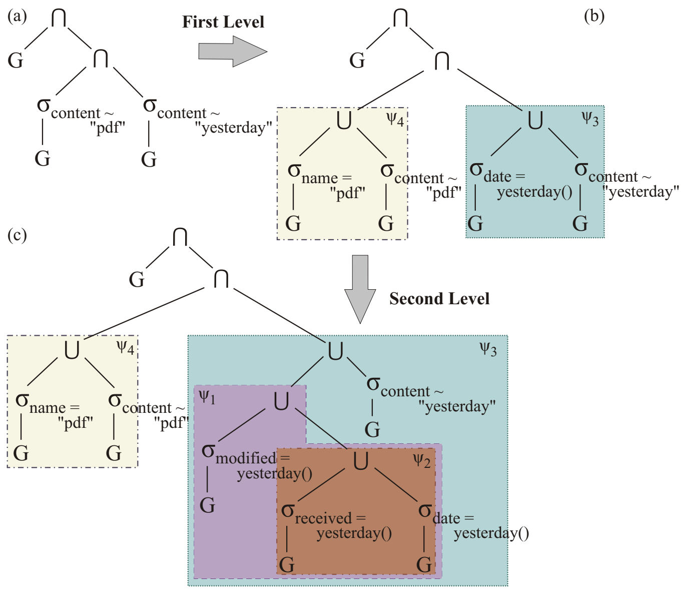


Figure 3: Algorithm 1 applied to Example 1. (a) original query, (b) rewritten query after one level, (c) after two levels.

EXAMPLE 6 (MULTIPLE TRAILS REWRITE) Recall the query `pdf yesterday` and the trails introduced in Use Case 1 of Section 3.2. Figure 3(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(b). Now, all nodes in the merged subtrees have been colored according to the trails applied. 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(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 a trail from Use Case 1 is matched and that is not already colored by that trail. Thus, the algorithm terminates. □

4.6 Trail Rewrite Analysis

THEOREM 1 (WORST CASE OF MMCA) *Let L be the total number of leaves in the query Q. Let M be the maximum number of leaves in the query plans introduced by a trail ψ_i. 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. This leads to 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 coloured 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 d-th level, this will lead to $L(M+1)^{d-1}$ trail applications and a total of $L(M+1)^d$ leaves.* □

COROLLARY 1 (MMCA TERMINATION) *MMCA is guaranteed to terminate in $O(L \cdot M^d)$ trail applications.* □

THEOREM 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:*

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

This means 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 5).

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 (see e.g. [25]). 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. This allows us to use a very space-efficient materialization that is similar to the document-ID list of inverted lists [45]. 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 Mat $:= \psi_i^{L.Q}$**.** Our first materialization option is to materialize the results of the query on the left side of a trail ψ_i as an inverted OID list. When $Q^M_{\psi_i} = \psi_i^{L.Q}$, the materialization of $\psi_i^{L.Q}$ may be beneficial. However, we may not always benefit from this materialization in general.

Materialize Mat $:= \psi_i^{R.Q}$**.** We may choose to materialize the query expression on the right side of a trail definition. That is beneficial as $\psi_i^{R.Q}$ is a subtree of $Q^T_{\psi_i}$.

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.

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:

Algorithm 1: Multiple Match Colouring Algorithm (MMCA)

```
Input: Set of Trails $\Psi = \{\psi_1, \psi_2, \ldots, \psi_n\}$
Canonical form of query $\Gamma(Q)$
Maximum number of levels $maxL$
Output: Rewritten query tree $Q_R$
Set $mergeSet \leftarrow <>$
Query $Q_R \leftarrow \Gamma(Q)$
Query $previousQ_R \leftarrow nil$
$currentL \leftarrow 1$
// (1) Loop until maximum allowed level is reached:
while $(currentL \leq maxL \wedge Q_R \neq previousQ_R)$ do
    // (2) Perform matching on snapshot of input query $Q_R$:
    for $\psi_i \in \Psi$ do
        if $(Q^M_{R\,\psi_i}$ exists $\wedge$ root node of $Q^M_{R\,\psi_i}$ is not colored by $\psi_i)$ then
            Calculate $Q^T_{R\,\psi_i}$
            Color root node of $Q^T_{R\,\psi_i}$ with color $i$ of $\psi_i$
            Node $annotatedNode \leftarrow$ root node of $Q^M_{R\,\psi_i}$
            Entry $entry \leftarrow mergeSet$.getEntry($annotatedNode$)
            $entry.transformationList$.append($Q^T_{R\,\psi_i}$)
        end
    end
    // (3) Create new query based on {node, transformationList} entries:
    $previousQ_R \leftarrow Q_R$
    for $e \in mergeSet$ do
        ColorSet $CS \leftarrow$ (all colors in $e.annotatedNode$) $\cup$
                (all colors in root nodes of $e.transformationList$)
        Node $mergedNode \leftarrow$
                $e.annotatedNode \cup Q^T_{R\,\psi_{i_1}} \cup \ldots \cup Q^T_{R\,\psi_{i_k}}$,
              for all colors $\{i_1, \ldots, i_k\}$ in $e.transformationList$
        Color all nodes in $mergedNode$ with all colors in color set $CS$
        Calculate $Q^*_{R\,\{\psi_{i_1},\ldots,\psi_{i_k}\}}$ by replacing $e.annotatedNode$ by
                $mergedNode$ in $Q_R$
        $Q_R \leftarrow Q^*_{R\,\{\psi_{i_1},\ldots,\psi_{i_k}\}}$
    end
    // (4) Increase counter for next level:
    $currentL \leftarrow currentL + 1$
    $mergeSet \leftarrow <>$
end
return $Q_R$
```

DEFINITION 11 (TRAIL WEIGHTING FUNCTION) *For every match $Q^M_{\psi_i}$ of a trail ψ_i the function $w(Q^M_{\psi_i}, \psi_i)$ computes a weighting for the transformation $Q^T_{\psi_i}$.* $\Box$

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 3.4 and 3.5).

5.1.1 Ranking Trails by Probabilities

The first strategy to rank trails is to consider the probabilities assigned to the trails (see Section 3.4).

DEFINITION 12 (PROBABILISTIC TRAIL WEIGHTING FUNCTION) *Let p be the probability assigned to a trail ψ_i. Then, $w_{prob}\big(Q^M_{\psi_i}, \psi_i\big) := p$ defines a probabilistic weighting function on $Q^T_{\psi_i}$.* $\Box$

Here, the probability p provides a *static* (query-independent) ranking of the trail set – analogue to PageRank [6], which provides a query independent ranking of Web sites. An interesting extension would be to rank the trail set *dynamically* (query-dependent), e.g., by considering how well a query is matched by a given trail. Dynamic trail ranking is an interesting avenue for future work.

5.1.2 Ranking Trails by Scoring Factors

The second strategy to rank trails is to also consider the scoring factors assigned to the trails (see Section 3.5).

DEFINITION 13 (SCORED TRAIL WEIGHTING FUNCTION) *Let sf be the scoring factor assigned to a trail ψ_i. Then, $w_{scor}\left(Q^M_{\psi_i}, \psi_i\right) := w_{prob}\left(Q^M_{\psi_i}, \psi_i\right) \times sf$ defines a scored weighting function on $Q^T_{\psi_i}$.* □

Ranking Query Results. The scoring factor determines how relevant the scores of results obtained by $Q^T_{\psi_i}$ 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 $Q^T_{\psi_i}$ by sf. In our implementation we achieve this by introducing an additional *scoring operator* on top of $Q^T_{\psi_i}$. We also exploit this for Top-K processing [16] (see experiments in Section 6.2).

5.2 Pruning Strategies

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

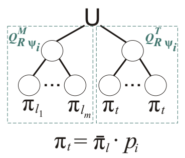
1. **Prune by Level,** i.e., punish recursive rewrites. The deeper the recursion, the further away the rewrite will be from the original query. We prune by choosing a value $maxL < N$ in Algorithm 1.
2. **Prune by Top-K Ranked Matched Trails.** Trails with the highest scores will provide the greatest 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.

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, this may come at the cost of precision.

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_i, with probability p_i, matches Q_R. The match $Q^M_{R\,\psi_i}$ includes leaves $l_1, \ldots, l_m$ of Q_R. This is visualized in the figure on the right.



$\pi_t = \bar{\pi}_l \cdot p_i$

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 $Q^T_{R\,\psi_i}$ by $\pi_t = \bar{\pi}_l \cdot p_i$. So, if p_i=1, we expect $Q^T_{R\,\psi_i}$ to contribute the same proportion of relevant results to the query as $Q^M_{R\,\psi_i}$. If $p_i <$ 1, then that proportion is affected by our certainty about ψ_i. Thus, every leaf in $Q^T_{R\,\psi_i}$ 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:

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

This means 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:

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

This means we sum the expected number of relevant results produced by each leaf.

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). 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 6.1. The results for Scenario 1 are presented in Sections 6.2 and 6.3. The results for Scenario 2 are presented in Sections 6.4 and 6.5.

6.1 Data and Workload

System. All experiments were performed on top of our `iMeMex` Dataspace Management System [4]. `iMeMex` 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[2]. The server in which the experiments were executed is a dual processed 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*.

Data. We conducted experiments on a dataspace composed of four different data sources similar to Figure 1. The first collec-

[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.

tion (Desktop) contained all files as found on the desktop of one of the authors. The second collection (Wiki4V) contained all English Wikipedia pages converted to XML [43] in four different versions, stored on an SMB share. The third collection (Enron) consisted of mailboxes of two people from the Enron Email Dataset [15], stored on an IMAP server. The fourth collection was a relational DBLP dataset [14] as of October 2006, stored on a commercial DBMS. Table 5 lists the sizes of the datasets.

	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 5: Datasets used in the experiments [MB]

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 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 the right. Furthermore, we defined a set of queries including keyword queries but also path expressions. These queries are shown in Table 6. 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.

```
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
```

Trails used in Scenario 1

No.	Query Expression	Original Tree Size	Final Tree Size	# 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 6: Queries for Trail Evaluation

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%. This means 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 ($skew = 1.12$).

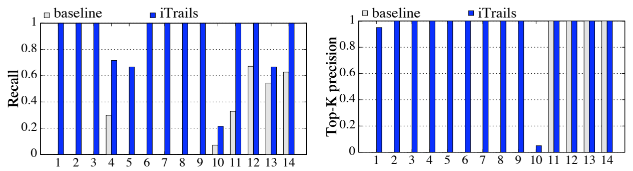


Figure 4: Trails: Recall and Top-K precision ($K = 20$) of query results: iTrails compared to baseline.

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 vs. Baseline. Figure 4 shows the total recall and Top-K 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, Q6–Q9. This happens because our approach exploits the trails to rewrite the query. For Top-K precision, the baseline approach fails to obtain any relevant result for queries 1–10, whereas it achieves perfect precision for queries 11–14. This happens because queries 11–14 are queries with partial schema knowledge that favor the baseline approach. In contrast, iTrails shows perfect precision for all queries except for Query 1, which still achieves 95% precision, and Query 10, 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 Query 10 could both be increased to 1. The same holds for Queries 4, 5, and 13, 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.

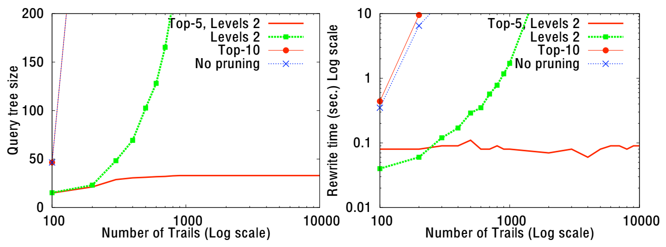
6.3 Query Performance

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

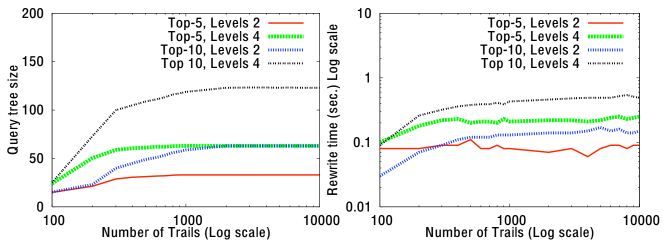
Trails vs. Perfect Query. We compared iTrails with the perfect query approach. Note that we did not compare against the original queries and keyword searches as they had unacceptably low precision and recall. Moreover, w.r.t. execution time it would be simply unfair to compare keyword searches against complex queries with structural constraints. The table on the right 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 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. This means that the overhead introduced by the trail rewrite is *not* negligible. This, however, can be fixed by exploiting trail materializations as presented in Section 4.7. The results in the third column of the table show that almost all queries benefit substantially from trail materializations.

Q. No.	Perfect Query with Basic Indexes	iTrails with Basic Indexes	iTrails with Trail Mat.
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

Execution times [sec]

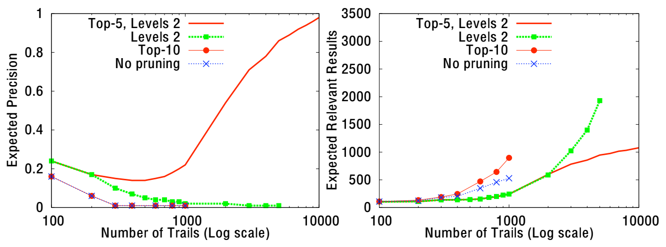


(a) Impact of pruning [query tree size & rewrite time]

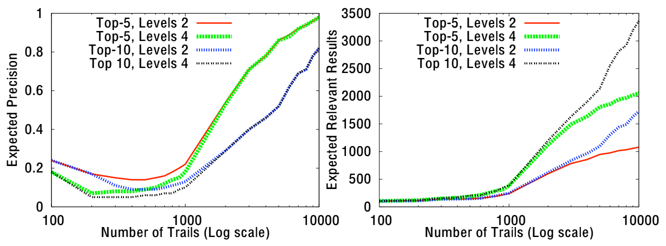


(b) Sensitivity of pruning [query tree size & rewrite time]

Figure 5: Scalability of Trail Pruning: Impact and Sensitivity



(a) Impact of pruning [precision (E_{Prec}) & relevant results (E_{Rel})]



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

Figure 6: Quality of Trail Pruning: Impact and Sensitivity

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 queries 1–14 are not known to the system beforehand, additional materialization for the perfect queries cannot be used. With trail materialization, Query 3 can now be executed in 0.39 seconds instead of 10.72 seconds (factor 27). For Query 9 the improvement is from 179.02 to 1.50 seconds (factor 119). In summary, using trail materializations all queries executed in less than 0.7 seconds, except for Query 9 which took 1.50 seconds. However, for this query the majority of the execution time was wasted in a deep unnest operator. The latter could be further improved, e.g., by adding a path index.

In summary, our evaluation shows that trail materialization significantly improves response time w.r.t. the perfect query approach.

6.4 Rewrite Scalability

In Figure 5(a) we show how the number of operators in the rewritten query grows as the number 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 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 5(a) we show only representative parameter choices for each of the strategies. As expected according to our rewrite analysis (see Section 4.6), the *No pruning* approach exhibits exponential growth in the query plan sizes, thus becoming inviable 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 5(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 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 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 Section 4.6). If we take the number of levels equal to 2 as shown in Figure 5(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 5(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 5(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 5(b), rewrite time has remained below 1 second. Rewrite time, just as tree size, also reaches a plateau. This is explained by the fact that our trail ranking scheme exploits 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 [16] 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.

6.5 Rewrite Quality

We evaluate the impact on expected quality as defined in Section 5.3 using the strategies studied in Section 6.4. Figure 6 shows the results. We computed expected relevant results (E_{Rel}) by taking c=100 (see Section 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 6(a) shows that expected precision drops significantly for the *No pruning*, *Top-10*, and *Levels 2* strategies as the number of trails increases. This 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 6.4, this strategy effectively limits the rewrite size, performing only up to a fixed number of trail applications. This means that, 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 6(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 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.

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.

Data Integration. Information integration systems (IIS) aim to provide a global, semantically integrated view of data. Classical systems either express the mediated schema as a view on the sources (GAV, e.g., TSIMMIS [34]), source schemas as views on the mediated schema (LAV, e.g. Information Manifold [27]), or a mix of these approaches (GLAV [18]). An extension to traditional IIS is quality-driven information integration [32]. 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 upfront effort to semantically integrate all source schemas and provide a global mediated schema. Only after this startup cost, queries may be answered using reformulation algorithms [21]. Peer-to-peer data management [39, 33] tries to alleviate this problem 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. In contrast to all of these approaches, dataspace management systems provide basic querying on all data from the start. iTrails comes into play as a declarative mechanism to enable semantic enrichment of a dataspace in a pay-as-you-go fashion.

Search Engines. Traditional search engines, such as Google, do not perform any semantic integration but offer a basic keyword search service over a multitude of web data sources. XML search engines, e.g. [10, 40], go one step further and provide queries combining structure and content over collections of XML documents. One important concern in these systems is ranking. The strategy chosen for ranking query answers is orthogonal to our approach for pay-as-you-go query evaluation. In fact, many ranking schemes may be used, including schemes that exploit scores assigned to trails. Unlike our approach, these systems do not provide integration semantics. The same holds for systems that use relevance feedback to improve search results. The focus of these approaches, e.g. [38], 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. Therefore, we consider all of the above search-based approaches variants of our *semi-structured search baseline* (No-schema).

Schema Matching. Much work has been done to semi-automatically generate schema matches and, ultimately, schema mappings. We may regard iTrails as a generalized method to specify equivalences among schema and instance elements. Unlike in previous work on schema matching [36], 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 [19] 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 [28] 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 [35]. 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 [13] 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 [9] 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 [8] 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 [37, 42] 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. iTrails, in contrast, 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 [17, 22] have proposed dataspaces as a new abstraction for information management. Dataspaces have been identified as one key challenge in information integration [23]. Recently, web-scale pay-as-you-go information integration for the Deep Web and Google Base is explored in [29]. 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, our paper is the first to provide an actual *framework* and *algorithms* for pay-as-you-go information integration. In previous work, we have proposed a unified data model for personal dataspaces [11] and the iMeMex Dataspace Management System [12, 4].

8. CONCLUSIONS

This paper is the first to explore 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 [17]. 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 [11], including personal, enterprise, or web sources.

As part of future work, we plan to explore other trail classes that allow us to express different integration semantics than the ones currently expressible by iTrails. In particular, we will explore in more detail the specification of join semantics. Furthermore, we plan to investigate techniques for (semi-)automatically mining trails from the contents of a dataspace. In that context, defining trails for mediated sources, such as hidden-web databases, is an especially interesting research challenge.

Acknowledgements. First of all, we would like to thank the anonymous reviewers for their helpful comments. In addition, we would like to thank our colleagues Nesime Tatbul, Peter Fischer, and Cristian Duda for comments on earlier versions of this paper.

9. REFERENCES

[1] S. Amer-Yahia, L. V. S. Lakshmanan, and S. Pandit. FleXPath: Flexible Structure and Full-Text Querying for XML. In *ACM SIGMOD*, 2004.

[2] L. Ballesteros and W. B. Croft. Phrasal Translation and Query Expansion Techniques for Cross-language Information Retrieval. In *ACM SIGIR*, 1997.

[3] O. Benjelloun, A. D. Sarma, A. Halevy, and J. Widom. ULDBs: Databases with Uncertainty and Lineage. In *VLDB*, 2006.

[4] 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 (Demo). In *CIDR*, 2007.

[5] M. Brantner, S. Helmer, C.-C. Kanne, and G. Moerkotte. Full-fledged Algebraic XPath Processing in Natix. In *ICDE*, 2005.

[6] S. Brin and L. Page. The anatomy of a large-scale hypertextual Web search engine. *Computer Networks*, 30(1-7), 1998.

[7] D. Calvanese. Data Integration in Data Warehousing (Keynote Address). In *CAiSE Workshops*, 2003.

[8] S. Chakrabarti et al. Memex: A Browsing Assistant for Collaborative Archiving and Mining of Surf Trails (Demo Paper). In *VLDB*, 2000.

[9] K. C.-C. Chang and H. Garcia-Molina. Mind Your Vocabulary: Query Mapping Across Heterogeneous Information Sources. In *ACM SIGMOD*, 1999.

[10] S. Cohen, J. Mamou, Y. Kanza, and Y. Sagiv. XSEarch: A Semantic Search Engine for XML. In *VLDB*, 2003.

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

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

[13] X. Dong and A. Halevy. Malleable Schemas: A Preliminary Report. In *WebDB*, 2005.

[14] DBLP Dataset. http://dblp.uni-trier.de/xml/.

[15] Enron Dataset. http://www.cs.cmu.edu/ enron/.

[16] R. Fagin. Combining Fuzzy Information: an Overview. *SIGMOD Record*, 31(2):109–118, 2002.

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

[18] M. Friedman, A. Levy, and T. Millstein. Navigational Plans For Data Integration. In *AAAI - Proceedings of the National Conference on Artificial intelligence*, 1999.

[19] R. Goldman and J. Widom. DataGuides: Enabling Query Formulation and Optimization in Semistructured Databases. In *VLDB*, 1997.

[20] R. Grishman. Information Extraction: Techniques and Challenges. In *SCIE*, 1997.

[21] A. Halevy. Answering queries using views: A survey. *The VLDB Journal*, 10(4):270–294, 2001.

[22] A. Halevy, M. Franklin, and D. Maier. Principles of Dataspace Systems. In *PODS*, 2006.

[23] A. Halevy, A. Rajaraman, and J. Ordille. Data Integration: The Teenage Years. In *VLDB*, 2006. Ten-year best paper award.

[24] A. Y. 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.

[25] V. Harinarayan, A. Rajaraman, and J. D. Ullman. Implementing data cubes efficiently. In *ACM SIGMOD*, 1996.

[26] 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.

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

[28] Y. Li, C. Yu, and H. V. Jagadish. Schema-Free XQuery. In *VLDB*, 2004.

[29] J. Madhavan, S. Cohen, X. L. Dong, A. Y. Halevy, S. R. Jeffery, D. Ko, and C. Yu. Web-Scale Data Integration: You can afford to Pay as You Go. In *CIDR*, 2007.

[30] G. Miklau and D. Suciu. Containment and Equivalence for an XPath Fragment. In *PODS*, 2002.

[31] T. Milo, S. Abiteboul, B. Amann, O. Benjelloun, and F. D. Ngoc. Exchanging Intensional XML Data. In *ACM SIGMOD*, 2003.

[32] F. Naumann, U. Leser, and J. C. Freytag. Quality-driven Integration of Heterogenous Information Systems. In *VLDB*, 1999.

[33] 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.

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

[35] Y. Qiu and H.-P. Frei. Concept Based Query Expansion. In *ACM SIGIR*, 1993.

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

[37] Resource Description Framework. http://www.w3.org/rdf. (rdf).

[38] R. Schenkel and M. Theobald. Feedback-Driven Structural Query Expansion for Ranked Retrieval of XML Data. In *EDBT*, 2006.

[39] I. Tatarinov and A. Halevy. Efficient query reformulation in peer data management systems. In *ACM SIGMOD*, 2004.

[40] M. Theobald, R. Schenkel, and G. Weikum. An Efficient and Versatile Query Engine for TopX Search. In *VLDB*, 2005.

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

[42] Web Ontology Language (OWL). http://www.w3.org/2004/owl..

[43] Wikipedia Database Dump. http://download.wikimedia.org.

[44] WordNet. http://wordnet.princeton.edu/.

[45] J. Zobel and A. Moffat. Inverted Files for Text Search Engines. *ACM Computing Surveys*, 38(2), 2006.