UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

SÉRGIO LUIS SARDI MERGEN

Indexing and Querying Dataspaces

Thesis presented in partial fulfillment
of the requirements for the degree of
Doctor of Computer Science

Dr. Carlos Alberto Heuser
Advisor

Porto Alegre, May 2011

CIP – CATALOGING-IN-PUBLICATION

Mergen, Sérgio Luis Sardi

Indexing and Querying Dataspaces / Sérgio Luis Sardi Mergen. – Porto Alegre: PPGC da UFRGS, 2011.

124 f.: il.

Thesis (Ph.D.) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, BR–RS, 2011. Advisor: Carlos Alberto Heuser.

1. Dataspaces. 2. Data Integration. 3. Search Engine. 4. Indexing. 5. Query Rewriting. I. Heuser, Carlos Alberto. II. Título.

AGRADECIMENTOS

Teses de doutorado trazem contribuições. Isso todos sabem. Talvez o que passe despercebido é a real extensão da afirmação. Por isso volto a afirmar, sem medo da repetição, pois foi repetindo que cheguei até aqui. Teses de doutorado trazem contribuições: as suas.

Dedico o trabalho a todos que me ajudaram de uma forma ou outra. Não vou citar nomes, pois de referências já bastam as do texto. Além do mais, vocês sabem quem são. Se não souberem fiquem tranquilos. Na próxima tese serei mais objetivo. ;-)

Obrigado a todos!

CONTENTS

LIST OF FIGURES

LIST OF TABLES

ABSTRACT

Over the Web, distributed and heterogeneous sources with structured and related content form rich repositories of information commonly referred to as dataspaces. To provide access to this heterogeneous data, information integration systems have traditionally relied on the availability of a mediated schema, along with mappings between this schema and the schema of the source schemas.

On dataspaces, where sources are plentiful, autonomous and extremely volatile, a system based on the existence of a pre-defined mediated schema and mapping information presents several drawbacks. Notably, the cost of keeping the mappings up to date as new sources are found or existing sources change can be prohibitively high.

We propose a novel querying architecture that requires neither a mediated schema nor source mappings, which is based mainly on indexing mechanisms and on-the-fly rewriting algorithms. Our indexes are designed for data that is represented as relations, and are able to capture the structure of the sources, their instances and the connections between them.

In the absence of a mediated schema, the user formulates structured queries based on what she expects to find. These queries are rewritten using a best-effort approach: the proposed rewriting algorithms compare a user query against the source schemas and produces a set of rewritings based on the matches found.

Based on this architecture, two different querying approaches are tested. Experiments show that the indexing and rewriting algorithms are scalable, i.e., able to handle a very large number of structured Web sources; and that support simple, yet expressive queries that exploit the inherent structure of the data.

Keywords: Dataspaces, Data Integration, Search Engine, Indexing, Query Rewriting.

1 INTRODUCTION

From data produced by Web services and published from online databases to Web tables, the volume of structured data on the Web has grown considerably in the recent past. Additionally, there is a lot of distributed resources related to the same universe of discourse, and they often present overlapping information about the same object.

Recent literature refers to these structured Web sources as Dataspaces (HALEVY; FRANKLIN; MAIER, 2006; FRANKLIN; HALEVY; MAIER, 2005). A simple way to define a dataspace is as a set of structured and autonomous information sources that address a specific need and contain (implicit/explicit) connections among them.

In contrast to unstructured (textual) documents, the presence of structure enables rich queries to be posed against dataspaces. This creates new opportunities for mining, correlating and integrating information from an unprecedented number of disparate sources (AGRAWAL et al., 2008).

Existing approaches to data integration, however, are very limited when it comes to dataspaces. Consider, for example, information mediators (WIEDERHOLD, 1997). These systems define an integrated global schema and pre-defined mappings that connect this schema with the ones of the sources. Based on the mappings, queries over the global schema are translated into queries over the information sources. This process is best known as query rewriting (ULLMAN, 1997; HALEVY, 2000; BOYD et al., 2004; FRIEDMAN; LEVY; MILLSTEIN, 1999).

Clearly, it would not be feasible to manually create and maintain such a system for thousands (or even hundreds) of sources. Because new sources are constantly added and existing sources modified (both content and structure), keeping track of sources as they change and updating mapping information and global schema can be prohibitively expensive. Besides, it is unlikely that a single integrated schema would be suitable for all users and information needs.

Although solutions have been proposed to amortize the cost of integration, either through mass collaboration (MCCANN et al., 2005) or by using a peer to peer architecture (TATARINOV et al., 2003), there is still a need to create and maintain mappings for data to be included in the integration system.

In an orthogonal direction, search engines are also considered for the dataspace environment. Search engines are highly used to perform keyword search over Web documents, where information is modelled as bag of words and queried through simple keyword-based interfaces (BRIN; PAGE, 1998).

When it comes to dataspaces, specialized search engines are required in order to handle structure. So far, related work accomplished to use search engines as a mean to reduce maintenance cost, which represents one of the greatest problems when dealing with a multitude of sources. However, they are still not able to use the structure meaningfully. For example, the query expressiveness is limited and tend to follow the keyword based model, in which queries are simple collections of terms.

In this thesis we propose a new approach to integration designed to deal with the scale and dynamic nature of structured Web sources. Our goal is to provide users the ability to query and integrate a large number of structured sources available on the Web on the fly, merging the query expressiveness of data integration systems to the flexibility of dataspace integration systems.

From this merging, three important features in our approach arise:

- There is no global schema: Instead of posing queries over a global schema, a user formulates queries based on her knowledge of the domain and on what she expects to find. The queries can be refined as the user explores and learns more about the information sources.

- Mappings are derived automatically: Instead of requiring mappings to be predefined, a user query is rewritten into queries over the sources based on correspondences (automatically) identified between attributes in the query and attributes in sources.

- Queries are structured: Queries are more than the simple lists of keywords from traditional search engines. By exploring the structure of the sources, our approach allows users to pose meaningful queries and consequently achieve higher quality answers.

To handle the problem properly, we have built the Eidos architecture. One of the main premisses of the architecture is to treat each individual data source as a relation. To support this idea, our underlying model is based on three interconnected entities: relations, attributes and values. Relationships between these entities are captured in specialized indexes. In addition, the indexes capture associations across relations, i.e., relations that share attributes and values.

Using proper wrappers, contents of the sources are extracted as relations and stored in the indexes. Furthermore, the associations between sources (relations) are discovered using a simplistic but straightforward approach - by linking relations that share homonym attributes, and having at least one value in common for these shared attributes.

Based on the indexes, it is possible to express queries with predicates. Rewriting mechanisms of Eidos are responsible for finding the sources that best fit the user requests, compute the answers and even produce results that correlate information from multiple sources, which enables on-the-fly integration at an unprecedented scale.

As expected, this best-effort approach cannot give guarantees of coverage (i.e., that all answers are retrieved) or even that the returned answers are ‘correct’. In this thesis, we investigate the consequences of working with unprecise integrations systems as a way to find structured information on the Web.

This work is structured as follows:

Chapter 2 discusses concepts that apply to data integration systems in general, such as mediators, wrappers, levels of heterogeneity, different mapping strategies and characterization of the data sources.

Chapter 3 focus on the problem of integrating dataspaces. First, it is shown why traditional data integration system fail to support a large corpus of volatile source efficiently. Next, we show approaches that do not depend on pre-defined mappings. Special attention is dedicated to works that are search-engine based.

Chapter 4 presents the proposed Eidos architecture. The components that compose the architecture are described, and those responsible for indexing and query rewriting are highlighted, as they represent the major contribution of the thesis. It is shown that the architecture is flexible, and allows components to be replaced in order to easily adapt the system to different realities, such as new types of data sources or new querying capabilities.

Chapter 5 shows one of the proposed querying approaches, in which the user is able to perform Select-Project-Join queries. We describe the indexing and querying components of the Eidos architecture necessary to support this type of query.

Chapter 6 shows the other proposed querying approach, in which the user specify selection predicates only - joins are not declared in the query. It is up to the querying algorithm to decide if the query can be answered by combining information from multiple sources. This chapter discusses how the indexing and rewriting components are adapted in order to support this type of query.

As a proof of concept, we built a system that supports queries over large volumes of (real) structured Web information sources. We focused on data that is structured as relations and whose schema (i.e., attribute names) and contents can be extracted automatically. Chapters 7 and 8 describe this implementation and discuss experimental results that indicate that our indexing and query rewriting algorithms are both effective and scalable. We also compare our work with one found in the literature that is most similar to ours in nature: use indexes to seamlessly query structured sources on the Web.

Finally, Chapter 9 brings the concluding remarks. It is important to state that the use of a best-effort approach represents a paradigm shift to data integration, which deserves a new look and the adaptation of concepts that so far applied to precise querying mechanisms only. Therefore, we finish this chapter discussing relevant topics that still need to be addressed in future work.

2 INFORMATION INTEGRATION

Until recently, information systems were mostly used by applications dedicated to handle information in a well-defined and controlled context. Examples include industrial applications (like ERP and CRM) and simple applications suitable for home users and small companies. In such cases, the relational database technology is able to handle most of the problems smoothly.

The advent of the Internet represents a mark as to what to expect from information systems. The volume of distributed information published on the Web by unrelated users introduced a challenge that so far did not received much attention: The problem of integrating information from independent data sources.

The change of perspective lead researches in the quest of new approaches that are able to handle distributed information, and terms like autonomy and heterogeneity were introduced as parts of the challenges. Heterogeneity refers to sets of schemas that are not designed according to a common model/schema. Autonomy refers to the source being capable of modifying itself without the permission of a central authority.

Autonomy can occur in different levels, and data integration systems can be classified according to the level of autonomy of its data sources. At one end are the integration systems that do not allow the sources to change neither its content nor its schema. This configuration is found mostly in data replication solutions, where data needs to be distributed in several nodes, and changes to a node need to be replicated (or at least accessible) to the others.

Communication occur between a master node and slave nodes. The master node (the central authority) is allowed to modify data and the slave nodes are periodically updated by the master. Some extensions consider nodes that act either as master and slaves (PACITTI; SIMON, 2000). This architecture provides better autonomy, since several nodes are allowed to modify its own data. Another advantage of having multiple masters is that nodes can operate off-line: the nodes are updated when the data link becomes available.

Data replication has many usages, such as improving availability of information, in the case of node failures, and achieving load balancing, in case of high number of concomitant accesses (ENSLOW, 1978). Moreover, replication solutions are generally applied over databases that are designed to solve a specific problem. For this reason, architectures that support replication usually work with homogeneous sources, where there is a common schema and access policies.

A higher degree of autonomy is achieved by federated databases (SHETH; LARSON, 1990). A database of this kind is composed by a federation of independent data sources, in which each of them operates according to its own policies, and there is no central authority responsible for accessing/updating information. The important aspect of this architecture is that the sources continue to operate the same way as when they were not part of the federation. Besides, data sources can be heterogeneous.

Federated databases can be loosely coupled or tightly coupled. In a loosely coupled setting, the location of the sources must be explicitly provided inside the command language responsible for accessing/updating the data. Additionally, the user must be aware that different sources may have different schemas, and construct the commands using the proper elements of each source. The term multi-database refers to this concept (KENT, 1991). Some databases management systems (like Oracle) have multi-database support, which enables the user to query and even update information from sources that are part of the federation. In order to access distributed sources, the user build SQL commands and indicates the location of each table. It is then up to the multi-database to coordinate the command execution, and to control operations like join (in query commands) and transaction atomicity (in update commands).

In tightly coupled federated databases, the local schemas are integrated as a unified global schema. Queries are performed over the global schema, regardless of the individual elements of the local schemas. This increases transparency, since the user does not need to know the location of the sources or the structure of the information. This kind of database is commonly referred to as information (or data) integration systems (CALVANESE et al., 1998).

Information integration systems must provide mechanism that allow users to access data sources in a transparent fashion, where the data sources can be autonomous and heterogeneous. There are several applications for data integration, such as corporative integration, data warehouses, data mining, distributed access to sources on the Web, Hidden Web, access to legacy systems, and data exchange, just to name a few (HALEVY, 2001; CALÌ et al., 2002).

Figure 2.1 shows, in a simplified manner, the typical design of a data integration architecture. This architecture is composed by a set of local schemas, a global (mediated) schema and mappings between the local schemas and the global schema (BOYD et al., 2004; LENZERINI, 2002). The global schema represents the querying interface, that is, the user interacts with the system by posing queries over the global schema (HALEVY, 2001; MANOLESCU; FLORESCU; KOSSMANN, 2001). Note the presence of wrappers in the architecture of the system. This component, as the name suggests, wraps a data source so all communications in or out the source are intercepted and adapted by it.

There are some variants of this design, such as mapping the local sources among themselves, but the principle remains the same. Given a query, the system uses the mappings between the global schema and the local schemas in order to send this global query to the relevant sources. The mappings are also used so this global query can be translated into a query that uses the constructs of the local schemas. The problem of translating the global query into sub-queries that use the schema of the sources is known as query rewriting. The terms query processing, query answering and query reformulation also refer to the same problem.

One of the main challenges in integration systems is precisely the one of integrating the sources properly. This particular problem comprises many related subjects that need to be handled, such as the development of wrappers, the implications of

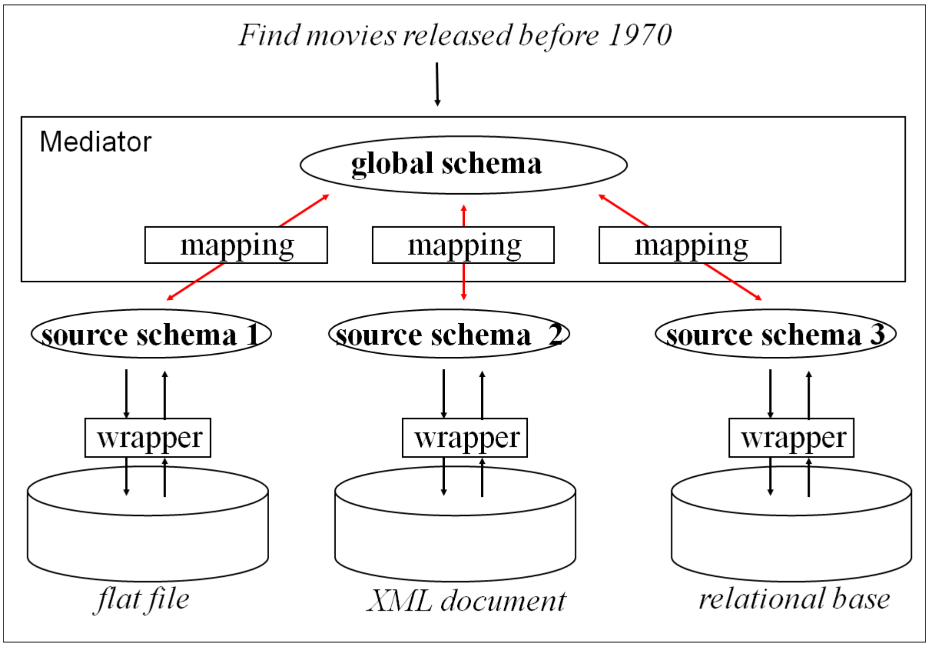


Figure 2.1: Typical architecture of data integration systems

having materialized views or not, the reconciliation of schemas (which involves problems such as schema matching and data cleaning), and the query processing itself. Putting it all together is a daunting task that very easily ends up with the creation of a complex architecture. In what follows, an overview of the most important subjects is presented.

2.1 Integration Architecture

2.1.1 Mediators/Topology

In the world of information integration, the mediator is the module which contains the administrative and technical knowledge to create integrated views of information that is distributed in several sources (WIEDERHOLD, 1997). From the architectural point of view, a mediator is composed by a global schema, which is the front end that expose the information to the outside world, and components that are dedicated to query processing.

Mediators can be classified as vertical or horizontal, according to the range of domains they support. Vertical mediators are specialized in a single domain, and thus provide access to a limited number of sources. On the other hand, horizontal mediators reach several different domains, and consequently provide access to a larger number of sources.

The obvious limitation of a mediator is that processes such as maintenance and query processing require more computational resources as the number of registered data sources increases. More recently, peer-to-peer (P2P) systems have been proposed as more scalable alternatives than the mediator architecture. Instead of using a centralized approach where data sources are mapped into a global schema, in a P2P architecture, there is a network of interconnected mediators (peers), where mappings are provided between the schemas of the peers (Figure 2.2).

In this approach, a user poses queries using the schema of its own peer. Queries

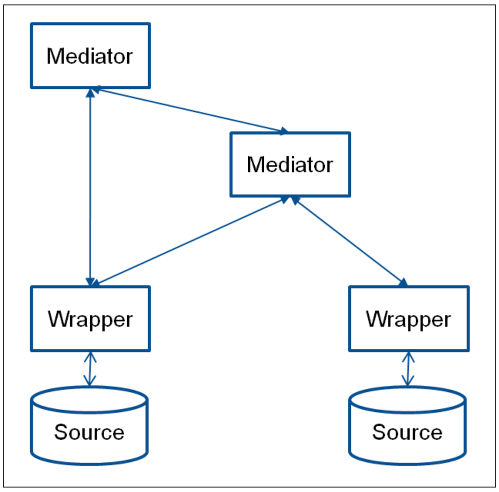


Figure 2.2: Network of interconnected mediators

over one peer can obtain relevant data from any reachable peer in the network. The mappings are traversed by reformulating a query at a peer into queries on its neighbors (TATARINOV et al., 2003; OOI et al., 2003). This is an interesting approach to build a horizontal service, in which the computational resources are distributed across the several mediators that compose the hierarchy.

2.1.2 Wrappers and Extractors

The concept of wrapper was first used in (WIEDERHOLD, 1997). As Figure 2.1 shows, a wrapper acts as an abstraction layer between the global schema and the sources, allowing the integration system to work with sources that are modelled using different paradigms. Given a user query, defined in terms of the integration system's query language, it is up to the wrapper to adapt the query considering the model of the local schema. The result of the query, described in terms of the source, also needs to be adapted to the format expected by the integration system. In software engineering, a similar concept is present under the name of the adapter design pattern.

The major benefit of wrappers is that they completely isolate the system from the sources. New types of data source are supported with no changes to the integration system, as long as it is possible to build a wrapper that maps the two languages properly.

Differently than wrappers, extractors are used to transform a data source into a different type of representation. They are usually employed to transform data sources that are unstructured (or whose structure is hard to understand) into a more interpretable structured data format.

For instance, some Web sites expose information in a tabular format. This is achieved by writing HTML code that is rendered by a browser in a tabular fashion. In some cases, this structure can be easily inferred by a extractor, when the tables are built using the proper HTML constructs, such as <tr> and <td>. However, when other constructs are used, the process of identifying rows and cells becomes more complex.

It is important to note the difference between an extractor and a wrapper. The wrapper resolves the structure differences between the communicating sides, and it assumes that the local data sources are structured. It is up to the extractor to reveal the structure of a data source, so the information underneath it can be processed. In some cases, extractors are built as part of wrappers (GRUSER et al., 1998). However, extractors can also be built to be used in isolation, if necessary.

2.2 Levels of Heterogeneity

There is a great number of data sources published over the Web, in many different formats. One of the primary goals of an integration system is to provide access to as many of them as possible. This goal is hard to achieve, considering the sources are highly heterogeneous. Several factors contribute to form heterogeneous data sources, such as the ones listed below:

Format/model: Data sources can expose information in different formats / models. Examples include media (MPEG, WMV,...), hyper-media (HTML, PDF), text (CSV, XML), and data storage models (relational, hierarchical, object-oriented, ...).

Structure: Even when the same model is used, differences can occur at the structural level. For instance, two documents could be described in XML model, and still, the structure of the elements can differ considerably, such as the name of the elements and the way the information is nested.

Semantic: This level of heterogeneity occurs when a universal concept is not uniformly represented across data sources. This category can be sub-divided as follows:

> **Naming Heterogeneity:** Comprises homonymous and synonymous problems. Homonymy occurs when the same name is used to represent different concepts, and synonymy occurs when different names are used to represent the same concept.
>
> **Structure Heterogeneity:** This problem happens when the relationships among concepts change from data source to data source. For example, when two concepts from different schemas contain different parents.
>
> **Domain Heterogeneity:** data sources use different values as possible instances for a given concept.

The issue of heterogeneity also reaches the query language, which represent the means by which access to the sources is granted. The language is usually a sub-product of the data model. For instance, SQL is the language adopted by relational databases, whereas OQL is the standard language for object-oriented databases.

Observe that the presented levels of heterogeneity exist not only among the sources, but also between an integration system and the sources. Naturally, the global schema differs than the local schemas, both structurally and semantically. Besides, integration systems are normally built based on a data model whose constructs follow a proprietary vocabulary. For instance, in the integration system Tsimmis (one of the earliest ever built), data is described in the proprietary OEM format, which is a self-descriptive format capable of modelling objects. Additionally, queries are defined in a proprietary rule-base language called MSL (CHAWATHE et al., 1994).

2.3 Mapping

2.3.1 Schema Matching

In order to create the mapping between a global and a local schemas, they first need to be matched to one another. In other words, it is necessary to determine which elements from one schema correspond to elements in the other schema.

The task of manually defining the matches can be too time consuming, when there is a magnitude of schemas that need to be integrated. To prevent this huge amount of labor, (semi)automatic match techniques are used. These techniques are generally called matchers. A matcher suggests a list of matches, along with a similarity score that determines the confidence of each match. It is up to the user to ultimately define which of the suggested matches are correct.

Matchers can be classified in several ways (RAHM; BERNSTEIN, 2001). In this thesis we divide them as coarse/fine grained, local/historical, schema/instance based and single/composed.

In fine grained matching, the matcher compares properties of the finer grained elements of the schemas, such as their names and data types. In coarse grained matching, the matcher compares elements according to the structure that surrounds them. The general idea is that two elements are similar if other elements in their surroundings are considered similar.

Schema based matchers are those that rely on schema information in order to find the matches. Conversely, instance based matchers analyze the instance level, that is, the values that are structured according to the underlying schema. This category can also be sub-divided into fine/coarse grained, where a fine grained instance matcher compares individual values, and a coarse grained instance matcher compares lists of values, such as records (BILKE; NAUMANN, 2005).

Historical matchers differ from local matchers in that they exploit information of previously stored matching information. In this sort of strategy, matches are considered transitive. For example, consider elements A,B and C, from three different schemas. If A matches B, and B matches C, there is a chance that A and C also match. Many historical matchers are based on machine learning techniques, where the learning algorithms are trained with a corpus of previously matched schemas (DOAN; DOMINGOS; HALEVY, 2003).

A matcher that implements one of the above described strategies is called a single matcher. Usually, the quality of the match is improved when a composition of single matchers is used. Inside a composition, weights indicate the relevance of a single matcher. The final similarity score is computed as the sum of the single matchers similarity score multiplied by their weights. Some works are dedicated into finding good matchers composition by automatically distributing weights. Usually the distribution is achieved by machine learning (MARIE; GAL, 2008).

The work of (DO; RAHM, 2002) presents a framework that handles matchers compositions. Inside the framework, it is possible to add matchers and to indicate how they are integrated in order to obtain a final similarity score. The tool is useful to visually test new matchers and new types of compositions.

2.3.2 Schema Mapping

The schema matching focus on the problem of finding which elements of different schemas correspond to each other. On the other hand, the schema mapping involves additional aspects that are not addressed by the matchers, mainly those that are present in ETL (Extract-transform-load) processes.

An ETL process comprehends three different phases (YAN et al., 2001): Extracting information from a source, transforming the extracted information according to a target format, and loading the transformed data as instances of the target schema. Common usages of ETL is to load information into a datawarehouse and to exchange information in e-commerce applications.

Considering the general integration system's architecture, the steps of ETL can be divided into those that take place inside the wrapper / extractor (the extraction) and those that take place inside the mediator (the transformation and load).

In the extraction phase, as already stated, one of the problems is to recover the data as a readable format. Another interesting problem is the automatic generation of queries, when a query language is available. For example, the work of Miller, Haas and Hernández shows that, when extracting information from relational databases, there can be multiple ways to join relations in a SQL query. They also present heuristics that help determining how to choose the better joining path (MILLER; HAAS; HERNáNDEZ, 2000).

In the transformation phase, information from the source need to be adapted to fit the format expected in the output. For one-to-one mappings, the most common transformations involves the applications of masks, such as when date/monetary values need to be represented in a different format. For one-to-many and many-to-one mappings, the transformations usually involve splitting a source value into multiple targets and aggregating multiple source values into a single target, respectively. While some of the transformations rules are easy to discover automatically, such as conversion masks, more complex rules, such as those based on conditions, require human intervention in order to be defined.

The loading phase faces the same challenges encountered in the extraction phase. For example, considering the target is a relational database, and that multiple target relations need to be feeded by source data, one of the problems is to correctly determine which target paths should be used, that is, which foreign key relations need to be created. In a integration system, the target is the format used by the system to deliver the results of a query to the user.

2.3.3 Mapping strategies

A mapping strategy defines the direction in which two schemas are mapped to each other. Considering the scenario in which a global schema (the schema of the mediator) is mapped to the local schemas, two main mapping strategies emerge: Global as View (GAV (ULLMAN, 1997)) and Local as View (LAV (HALEVY, 2000)).

2.3.3.1 GAV

In GAV, the local schemas are considered the source of the information. Elements of the global schema contain a script (GAV mapping) that indicates how to extract, from the local schemas, information that conform to the global definitions. In other words, elements of the global schema are views over elements of the local schemas.

Figure 2.3 illustrates an hypothetical use of GAV mappings. In the general case, information is represented as relations, in both local and global schemas. Observe that each relation in the global schema can map to one or more relations in the local schemas.

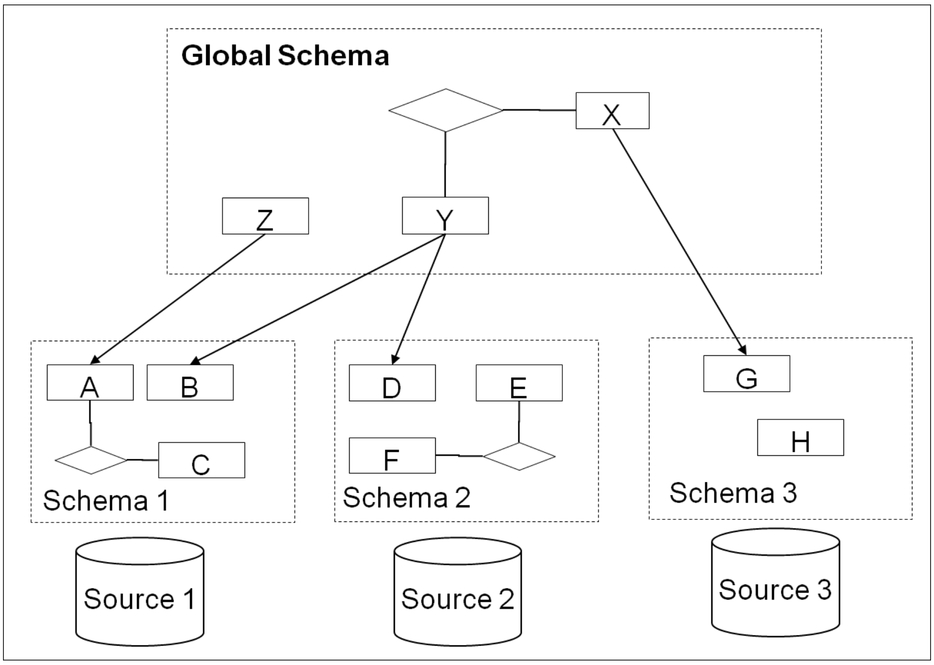


Figure 2.3: Hypothetical GAV mappings

Figure 2.4 shows a less abstract example, in which GAV mappings associate the global relations *movie* and *director* with the local sources $schema_1$ and $schema_2$. To simplify, the mappings are described as conjunctive queries, using the Datalog notation (CERI; GOTTLOB; TANCA, 1989).

The first mapping creates a one to one relationship between relations. Conversely, the second mapping shows a more elaborated case. In this case, the global relation *director* is given by a join between two local relations.

movie(title, year)	:-	$schema_1$(title, year, director)
director(title, name, country)	:-	$schema_1$(title, year, name), $schema_2$(name, country)

Figure 2.4: GAV mappings described as datalog rules

Given a query over the global schema, GAV mappings are used in order to brake the query down into local queries that conform to the local schemas. The process responsible for finding the local queries that correspond to the global query is known as query rewriting (HALEVY, 2001).

Query rewriting under GAV is relatively easy. Basically, elements of the query are replaced by their respective mapping script, in a operation called unfolding. For example, Figure 2.5 describes the query rewriting process, considering the mappings described in Figure 2.4. At the top, the query over the global schema appears, described in datalog notation. The unfolded version of the query is described below. The underscore identifies variables that are not part of the mappings and are therefore not relevant to the query.

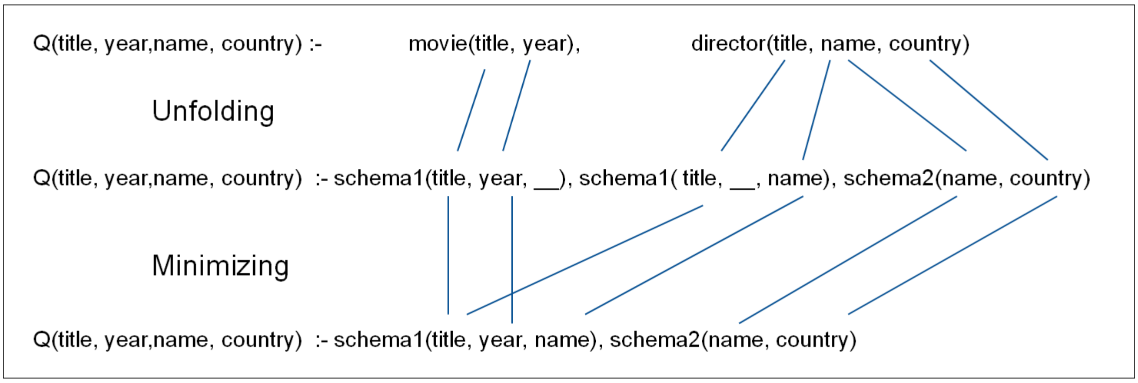

Q(title, year,name, country) :-
movie(title, year),
director(title, name, country)
Unfolding
Q(title, year,name, country) :- schema1(title, year, __), schema1(title, __, name), schema2(name, country)
Minimizing
Q(title, year,name, country) :- schema1(title, year, name), schema2(name, country)

Figure 2.5: Unfolding of a query using GAV mappings

Observe that the source relation *schema*2 appears twice, in a redundant self-join. In this case, it is possible to apply a minimization technique to simplify the query, as indicated in the Figure.

Query minimization involves removing subgoals of a query while keeping the same set of answers. It was proven that finding a minimal query is an NP-complete problem (CHANDRA; MERLIN, 1977) with respect to the number of subgoals. Since queries usually contain few subgoals, the time required to perform the minimization is not significant.

2.3.3.2 LAV

In LAV (Local as View), the global schema is considered the source of the information. Elements of the local schemas contain a script (LAV mapping) that indicates how to extract, from the global schema, information that conforms to the local definitions. In other words, elements of the local schema are views over elements of the global schema.

Figure 2.6 illustrates an hypothetical use of LAV mappings. Observe that each relation in local schemas can map to one or more relations in the global schema.

The figure 2.7 shows a less abstract example, in which LAV mappings associate the global relations *movie* and *schedule* with the local sources *schema*1, *schema*2 and *schema*3.

Note that in the LAV definition of *schema*1 the distinguished variable *director* is not used in the body of the view. In LAV, this only means that the source exposes more attributes than the ones described in the global schema.

In the LAV definition of *schema*2, there is a selection predicate indicating that this source contains movies released after 2000. When choosing the views that can answer a user query, this knowledge allows the rewriting algorithm to filter out views if the selection predicates of the query are inconsistent with the predicates of the view. This type of inference could not be made in a pure GAV approach.

The third view definition uses two global relations. This kind of construction makes the query rewriting problem harder to solve than in GAV. Frequently a user query can be rewritten in many ways, which can be exponential to the number of subgoals of the query and to the number of views that are mapped to the subgoals of the query (HALEVY, 2001).

Some rewriting algorithms are known for their ability to handle a large number of views. One of them, called Bucket, is used in Information Manifold sys-

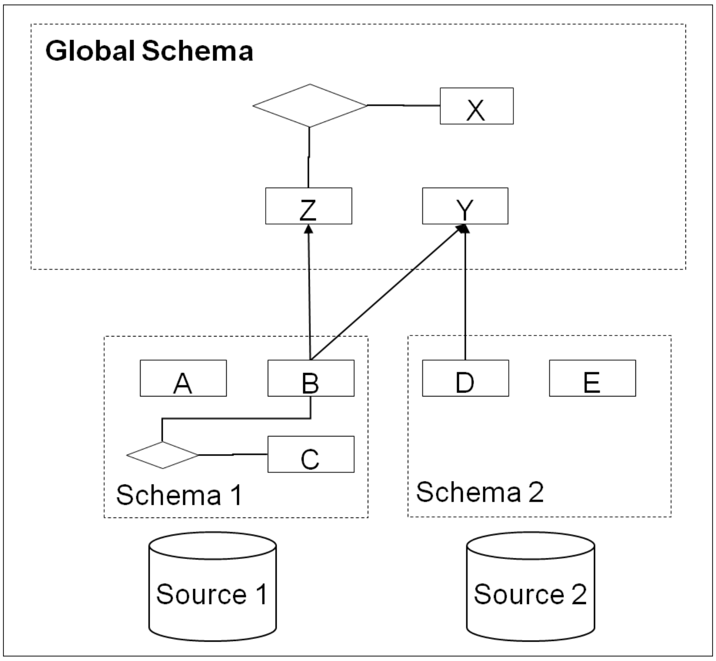


Figure 2.6: Hypothetical LAV mappings

$schema_1$(title, year, director, genre)	:-	movie(title, year, genre)
$schema_2$(title, year, genre)	:-	movie(title, year, genre), $year >= 2000$
$schema_3$(title, year, room, time)	:-	movie(title, year, genre), schedule(title, room, time)

Figure 2.7: LAV mappings described as datalog rules

tem (LEVY; RAJARAMAN; ORDILLE, 1996). The algorithm Inverse Rules was developed and used in the InfoMaster system (DUSCHKA; GENESERETH, 1997). In (POTTINGER; HALEVY, 2001) another algorithm is presented(MiniCon), but not applied to any particular system. Experiments show that the performance of the MiniCom algorithm is superior than the other two (POTTINGER; HALEVY, 2001).

As mentioned, query rewriting in LAV is harder than query rewriting in GAV. On the other hand, LAV favors the extensibility of the system: adding a new source requires only the addition of a rule to define the source, and the global schema needs no change. In such a scenario, the maintenance of the integration system can be decentralized, since the owner of the sources is in charge of providing the required LAV mappings, without ever caring about changes in the global schema.

Observe that a mapping strategy is a way to define ETL steps. In the provided examples, it is easy to identify the extraction and the loading parts. The transformation phase is suppressed, since no transformation rule was provided. However, transformation rules can be defined inside a mapping.

It is important to remark though that some transformations are irreversible. For instance, target values that are the result of a multiplication of two source values cannot be decomposed into the original source values. With this respect, GAV strategy is said to be more expressive, since there is no need to reverse the computation of a target value.

2.3.3.3 Other Mapping Strategies

From the GAV and LAV approaches, several others have emerged, like BAV and GLAV. The new strategies were created as an effort to combine the best of both worlds: the query unfolding capability of GAV and the flexible maintenance of LAV.

Both as View(BAV (BOYD et al., 2004)) combines the GAV and LAV approaches by using reversible mappings. The mappings are created as a consequence of a step that transforms one schema into another. The originality of this proposal is that the global schema relations can be defined as views over the sources and the source relations can be defined as views over the global schema.

One of the key benefits of BAV is that any querying mechanism that is used in GAV or LAV can also be used in BAV. Besides, this approach overcomes the limitations that are inherent to GAV, such as the inability to handle the local schemas evolution, and the limitations inherent to LAV, such as the inability to handle the global schema evolution.

Global Local as View(GLAV (FRIEDMAN; LEVY; MILLSTEIN, 1999; MADHAVAN; HALEVY, 2003)) is an extension of LAV where the header of the views may be a conjunction of predicates. Additionally, the header may contain variables that are not in the body of the view.

In Both Global Local as View(BGLaV (XU; EMBLEY, 2004)), the approach is to map the sources to a pre-defined global schema, which is specified regardless of the sources. The query rewriting is based on unfolding techniques.

2.4 Ontologies and Catalogs

There has been a lot of research on the information integration area with a leading focus of bringing more semantics to the mappings between the global schema and the mediator. Specially important is the use of ontologies to this end. Ontologies are special types of schemas that not only capture the structure of the data, but provide additional/hidden knowledge related to the data (NOY, 2004).

By defining the global schema and/or local schemas with ontologies, it is possible to create more meaningful mapping definitions, which leverages the source selection during query rewriting (SICILIA, 2005). Besides, ontologies allow users to submit requests using richer query languages.

In parallel with ontologies, source catalogs also improve source selection (LEVY; RAJARAMAN; ORDILLE, 1996). Information from a catalog guide the query rewriting algorithms by stating if a given source is relevant to a specific query. Common meta-information found in a catalog include:

Logical contents informs which domains are encompassed by the source (movies, books) and which ranges of values can be found (e.g. year $<$ 2000)

Capabilities informs which kinds of query are supported, and which attributes accept (or even demand) filters. For example, some Web Services can only be accessed if determined filters are provided.

Completeness informs if the source contains all existing information.

Reliability informs if the source can be trusted.

Additionally, statistics about the sources give approximate indicators for items like update frequency, availability and latency. Besides, these items enable the use of more concrete measurements than statistics. For example, availability and latency can be measured by the number of mirrors and the locality, respectively.

2.5 Open World Assumption and Closed World Assumption

Data sources can be classified according to the amount of data that they cover. There are three possibilities: they can be exact, complete or sound (incomplete). A data source is exact if it contains exactly all data that is part of the universe, not less and not more. A data source is complete if it contains at least all data that is part of the universe. The data source is sound if it does not contain all data that is part of the universe (ABITEBOUL; DUSCHKA, 1998).

The universe of data that is reached by an integration system can be classified as Open World Assumption (OWA) and Closed World Assumption (CWA).

In the scenario known as closed world, it is supposed that all sources are exact or complete. When a closed world is assumed, the answer to a query contains all data that is expected, at least for the context of the given application. This is typically the case of database systems, in which the data that is stored is the only information that matters.

Conversely, if a system is running in an open world, a query may not return all data that is expected. This is typically the case of integration systems that allow the access to arbitrary, and therefore sound, data sources. For instance, if a query asks for movies released before 1950, the answer obtained by an integration system is just a subset of all possible answers.

The querying of incomplete(sound) databases is a subject of research, in which the goal is to determine for which types of queries there exist a precise answer (IMIELINSKI; LIPSKI JR., 1984). To exemplify, non-monotonic queries are untractable if the sources are incomplete. Non-Monotonic queries are those in which tuples can be removed from the answer by the addition of a tuple in one of the relations used in the query, such as queries that use the negation operator.

For example, a query that joins tables *movie* and *rate* in order to discover which movies are not ranked with 5 stars is non-monotonic. If the movie "Star Wars"exists the *movie* relation and does not exist in the *rate* relation, it would be part of the answer. However, it cannot be assumed that it is not a 5 star movie just because this information is not stated in the *rate* relation.

The notion of certain answers is used as a formalism to determine which tuples are relevant as answers of a query, according to the available sources (ABITEBOUL; DUSCHKA, 1998). Given a set of sources and a query over this set, a tuple is a certain answer if it is a certain answer to any possible extension of each source. In the example above, Star Wars in an uncertain answer, since it would not be part of the answer if the relation *rate* contained a tuple that rates the movie with 5 stars.

It is important to remark that incomplete databases refer to missing information in general. The example above shows a case in which a tuple is not stated in a relation. Other cases are possible, such as tuples with null values for one of its attributes. Under Open World, it is impossible to say if the null value indicates that there exist no information for the attribute or if the information is unknown.

2.6 Query Containment

The concepts of query containment and equivalence (CHEKURI; RAJARAMAN, 1997) are important in the context of LAV integration systems, where queries are answered using rewritings over views (HALEVY, 2001). In this case, it is important to check if a rewriting brings more or less tuples than the query asked for.

We say that a query Q_1 is contained in the Q_2, denoted by $Q_1 \subseteq Q_2$, if the answers to Q_1 are a subset of the answers to Q_2 for any database instance. The query Q_1 is equivalent to Q_2, denoted as $Q_1 \equiv Q_2$, if and only if $Q_1 \subseteq Q_2$ and $Q_2 \subseteq Q_1$. In such cases, the answers of Q_1 are equal the answers of Q_2.

The problem of finding equivalent rewritings is mainly addressed in database systems processes, such as query optimization (GOLDSTEIN; LARSON, 2001). In this case, the purpose is to check if there are views that can substitute relations of the query, as a way to speed up the query execution. Another usage of equivalent rewritings is to keep the data physical layer independent from the data logical layer (TSATALOS; SOLOMON; IOANNIDIS, 1996).

Observe that equivalent rewritings only exist in closed worlds, like databases, since this is the only scenario in which it can be assumed that all (relevant)data is reachable. In open worlds, rewritings can only be contained, never equivalent. Additionally, information is spread over a multitude of autonomous sources, and there is the need to build rewritings that reach all such sources. This problem is known in the literature as the Maximally contained Rewriting Problem (HALEVY, 2001).

2.7 Chapter Remarks

This chapter describes an overview of the data integration field. Several topics related to this problem are presented, including conceptual definitions, architectural issues and general challenges.

Among other things, it is shown that integration systems are build under a solid foundation formed by mediated schemas and pre-defined mappings. Conversely, the work proposed in this thesis follows the opposite direction, in which mapping information cannot be taken from granted. Nevertheless, the notions related to data integration are useful for understanding the problems that are inherent to the querying of structured and heterogeneous sources on the Web, and may also suggest possible solutions for the common challenges.

For example, schema matching techniques used to consolidate mappings between global and local schemas provide opportune insights about how to automatically discover data sources that may contain answers to user queries. Similarly, the existence of consolidated mappings lead to quality related terms, such as query containment and certain answers. When mappings are absent, caring about these issues is at least not as important, since query answers are unprecise by nature. The awareness of this lack of precision is important, as it enables the creation of proper algorithms, whose aim is not at enforcing that answers are correct, but rather delivering a large number of answers as efficiently as possible.

Through the rest of the thesis, we show how the fundamentals introduced in this chapter participate in the creation of our querying mechanisms, from the definition of the overall architecture to the implementation of the indexing and rewriting algorithms. In addition, data integration topics are included in several possibilities of future work.

3 THE NO-SCHEMA APPROACH TO DATA INTEGRATION

The previous chapter presents a conceptual overview of the data integration subject, and briefly discusses the challenges that arise in this context. In this chapter, we introduce a novel aspect of integration, which appears in the world of dataspaces.

The problem of providing seamless access to structured data has already been addressed by a number of different works. Salles et al. (VAZ SALLES et al., 2007) classifies integration systems into two different categories: schema-first approach (SFA) and no-schema approach (NSA).

SFA assumes that the sources of information are integrated into a global schema along with mappings that connect this schema to the schema of the sources, as discussed in the previous chapter. In the other direction, NSA assumes that a global schema is not available.

Schema-first approaches are suitable to handle controlled data sources. However, they are limited when considering dataspaces. A dataspace can be seen as a collection of distributed, heterogeneous and partially unstructured data (HALEVY; FRANKLIN; MAIER, 2006), like those that are integrated through a global schema. What distinguishes dataspaces is the large number of sources available. This term become popular very recently, when researches started to notice that the Web is full of structured and autonomous sources that publish information about the same concepts of the world.

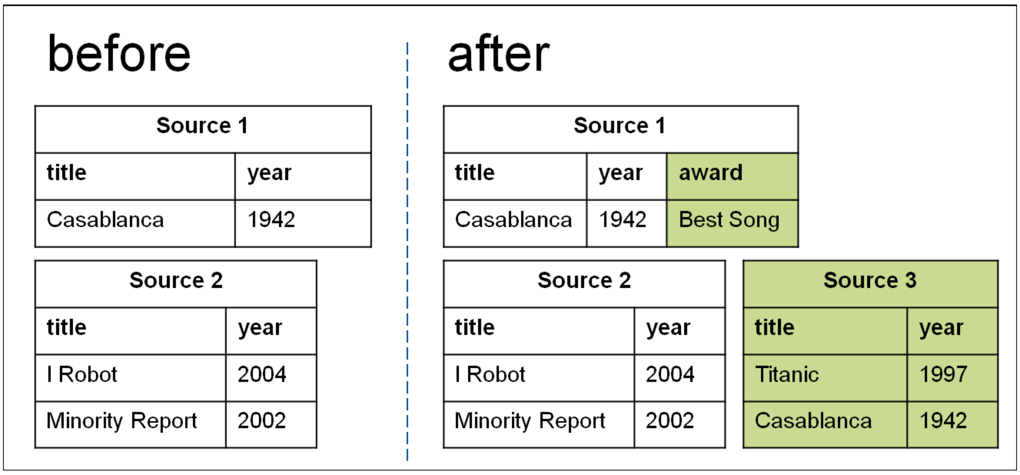
Next section shows why SFA is limited when it comes to dataspaces. Next, we present NSA works that are able to accommodate the particularities of dataspaces more smoothly.

3.1 Schema-First Approaches - Motivating Example

Traditionally, SFA considers a global schema mapped to the sources, using GAV/LAV rules or variations. To illustrate how a dataspace is handled by a system based on pre-defined mappings, consider the data sources described in Figure 3.1. Suppose that initially only the relations on the left are available.

A global schema could be easily created in order to integrate these sources. Say this schema is initially composed by a single relation, formed by the relation *movie* with attributes *title* and *year*.

Considering the GAV strategy, the mapping of this relation is represented as a union of queries, as demonstrated below:


before
after

before

Source 1	
title	year
Casablanca	1942

Source 2	
title	year
I Robot	2004
Minority Report	2002

after

Source 1		
title	year	award
Casablanca	1942	Best Song

Source 2	
title	year
I Robot	2004
Minority Report	2002

Source 3	
title	year
Titanic	1997
Casablanca	1942

Figure 3.1: Available data sources in two different moments

$$\begin{aligned} \text{movies(title, year)} &\ \text{:-}\ \ Source_1\text{(title, year)} \\ \text{movies(title, year)} &\ \text{:-}\ \ Source_2\text{(title, year)} \end{aligned}$$

As already mentioned, the benefit of GAV is that the rewriting is just a matter of unfolding (ULLMAN, 1997). However, adding new sources may require changes to the definition of the global schema. For instance, if the new relation $Source_3$ becomes available, the definition of *movies* needs to be modified to include the new source.

In contrast to GAV, LAV favors the extensibility of the system. In order to map source $Source_3$, all it takes is to create a new LAV mapping, as described below. Additionally, the task of creating the mapping can be delegated to the administrators of the sources.

$$Source_3\text{(title, year) :- movies(title, year)}$$

It is true that the LAV strategy is easier to maintain. However, this kind of mapping does not entirely prevent modifications to the global schema. For example, consider the new state of $Source_1$ (Figure 3.1). Observe that a new attribute is represented (*award*), and it is currently not supported by the global schema. Consequently, it is not possible to reach information about the awards received by movies.

For exploring and integrating information sources on the Web, expecting the existence of a pre-defined global schema is not practical. To define a global schema, one first needs to know about which sources are available. And on dataspaces, there are too many of them. It is unfeasible to model a universal global schema that contains constructs for every possible concept of the real world.

Besides, creating mappings and maintaining them as sources evolve and the global schema changes can be expensive and time consuming. This effort is required and justified for applications that are mostly static, cater to well-defined information needs, and integrate a relatively small number of sources. This is not the case on the Web, where the sources are not only plentiful, but also too volatile, which would require the mappings to be constantly updated.

In an attempt to improve scalability of schema-first approaches, recent approaches have been proposed to amortize the cost of integration, such as for example, peer-to-peer systems (TATARINOV et al., 2003) and community-based information integration systems (MCCANN et al., 2005). In the former, users provide mappings between peers, whereas in the latter, users collaborate in the creation of mappings between the sources and the global schema. However, their efficacy to the problem of integrating dataspaces is not yet conclusive. Additionally, the necessity to maintain the global schema remains in both of them.

3.2 NSA (No-Schema Approaches)

NSA assumes that a global schema is not available. In other words, the user needs to formulate queries with no knowledge of which concepts are modelled and how they are connected. The challenge is to translate user queries into queries that conform to the local schemas without the help of predefined mappings. The cost to maintain mappings is the greatest problem when dealing with integration systems. Therefore, the absence of mappings makes of NSA an interesting idea to the world of dataspaces.

Next sections present recent works that follow this direction. Section 3.2.1 shows approaches that perform keyword search over relational databases, where the database schema is unknown. Section 3.2.2 shows an approach that simulates a global schema by grouping the sources as a universal relation. Pay as you go approaches are discussed in Section 3.2.3. Finally, Section 3.2.4 presents approaches that replace the global schema by indexes that capture the structure of the sources.

3.2.1 Keyword Search over Relational Databases

This section presents solutions that adopt keyword search to retrieve information stored in relational databases. It is assumed that the user has no knowledge about the underlying database. When the user poses a query, the system derives an SQL query from the keywords provided. It is possible to apply this idea to dataspaces, if we consider Web sources as collections of relational databases.

In general, these approaches represent the database as a graph whose nodes are connected based on foreign key relationships. Two possibilities emerge: instance-level graph (HULGERI; NAKHE, 2002) and schema-level graph (HRISTIDIS; PAPAKONSTANTINOU, 2002).

In the instance-level graph, the nodes are tuples of the database. Given a query, keywords are mapped to nodes in the graph. Each subset of the graph that links all of the selected tuples become an individual answer. Results are then presented as nested information, according to the connections between the tuples.

In the schema-level graph, the nodes refer to elements of the database schema. Like the instance level counterpart, keywords of a query map to nodes in the graph. Then, each subset of the graph that links all of the selected nodes derives a query that contains some answers.

In BANKS, for example, information is viewed as a graph of tuples connected by foreign key relationships (HULGERI; NAKHE, 2002). The keywords of the queries are matched against the tuples, and the subgraphs that contain all matched tuples are present as possible answers.

Originally designed to relational databases(in the close-world-assumption sense), these approaches could be extended to Web information. However, it is not clear if and how they would be suitable to the large amount of information available on the Web, specially considering that foreign key relationships are unknown.

According to (HULGERI; NAKHE, 2002), it is possible to keep only a lightweight graph in main memory, delegating the remaining information to secondary storage. However, they do not report any kind of experiments over large data sets.

3.2.2 Universal Relation

The universal relation was also considered for the problem of querying multiple structured Web sources. Generally speaking, a database is virtually flattened as a single relation with a large number of attributes. This relation provides a user friendly interface where queries can be expressed without navigation (no joins need to be provided). It is up to the universal relation to infer which of the original relations need to be accessed, and how they have to be joined (MAIER; ULLMAN; VARDI, 1984).

Implementations of the universal relation, such as SUR (Structured Universal relation) proposes interfaces to query dynamic Web content from multiple sources (e.g., content publish by Web services and online databases). SUR was developed as the infrastructure to support webbases that are designed for well-defined domains (e.g., cars, jobs, houses). SUR requires experts in each domain to specify compatibility constraints for relations as well as concept hierarchies for attributes (DAVULCU et al., 1999).

Querying the universal relation is simpler than querying a database scheme, since it is not required to inform foreign key predicates when querying two or more related relations. However, when asking for attributes from different source relations, there may be more than one possible way to produce valid queries, each using different joins between the relations. This problem is formally defined in (FAGIN; MENDELZON; ULLMAN, 1982) as the presence of cycles in the hyper-graph that represents the join dependencies of the relations. Another drawback of universal relations is that they assume that every attribute has a unique name. Queries over the universal relation using non-unique attribute names would be ambiguous otherwise.

Moreover, it is expected that experts help in the process of mapping the SUR to the sources. Clearly, this is hard to achieve if the system needs to support a magnitude of sources that needs to be added on the fly.

3.2.3 Pay as You Go

There are also pay-as-you-go approaches that lie in-between the SFA and the NSA. Works of this kind start with few mapping information and evolve over time, taking user feedback into account (TALUKDAR; IVES; PEREIRA, 2010). Madhavant et al (MADHAVAN et al., 2007) states some architectural issues that need to be addressed in such a scenario.

One of the ideas behind pay-as-you-go strategies is to use probabilistic models, where an automatically created global schema is connected to the sources through probabilistic mappings. These mappings (and their probabilities) evolve as the user interacts with the system, which is why architectures of this kind are considered as self-configuring data integration systems (DAS SARMA; DONG; HALEVY, 2008; KHALID et al., 2010).

Salles et al. (VAZ SALLES et al., 2007) provides a concrete framework that supports dataspace enhancing by adding relationships among the data. Madhavant et al (MADHAVAN et al., 2007) also discuss how the index of a dataspace can be improved in a pay as you go fashion, where user feedback plays an important role to enrich the information of the indexes. The same ideas could be applied to our work as well. As will be discussed later, we present rewritings to a query individually. Hence, it would be possible for the user to mark an incorrect rewriting, and leave it up to the system to learn from that information.

3.2.4 Query Answering through Indexing

Some approaches use specialized indexes that store information about the Web sources. Considering a source as a collection of columns and records, the source indexing is responsible to store both data (the records) and the metadata (the columns). During querying, the terms of a query are identified in the indexes. Next, a correlation algorithm is used to locate the sources that might provide relevant answers.

Compared to the previous ideas presented in this chapter, these approaches are more dataspace oriented, since they are designed to handle the volatile nature of Web data - the maintenance cost is reduced by replacing the manual work of experts by a more automatic process of indexing.

In what follows, we present two works related to ours. Both treat Web sources as relations and propose indexing strategies to capture the structure of the sources. Additionally, the proposed indexes are accompanied by algorithms that find relevant answers to user queries.

3.2.4.1 WebTables

To support queries over relations on the Web, (CAFARELLA et al., 2008) proposed a search engine called WebTables. This search engine is based on an inverted list that connects keywords to the tables in which they appear. Queries are expressed by keywords and the result of the search are pages whose inner tables contain terms that appear as keywords in the query. In order to effectively find the most relevant information first, the results are ranked according to a set of features, as described later in the section.

Figure 3.2 shows the index populated with the contents of Table_1 and Table_2. The square brackets indicate the offset of the keyword inside the table.

There are some similarities between this index and the ones used by traditional search engines. The later also relies on inverted lists, which connect keywords to the document id where the keywords can be found. Besides, there is an offset along with the document id. This offset determine the distance of the keyword from the beginning of the document. In the work of Cafarella, the offset is a two-dimensional value that describes where in the table the keyword can be found. For instance, according to Figure 3.2, the keyword '2004' can be found in the third row/third column of Table_1 and the forth row/second column of Table_2.

Not only values, but metadata is stored in the index. Observe that some of the entries of Figure 3.2 refer to the names of the column tables. For example, according to the index, the keyword 'DATE'(a column name) can be found in the first row/second column of Table_2.

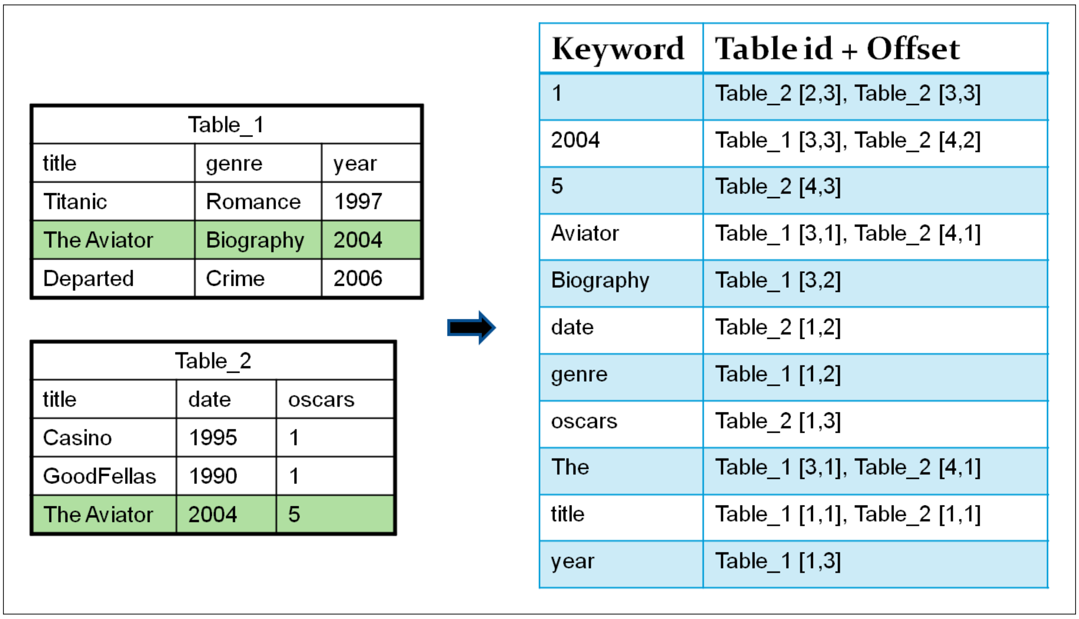


Table_1		
title	genre	year
Titanic	Romance	1997
The Aviator	Biography	2004
Departed	Crime	2006

Table_2		
title	date	oscars
Casino	1995	1
GoodFellas	1990	1
The Aviator	2004	5

Keyword	Table id + Offset
1	Table_2 [2,3], Table_2 [3,3]
2004	Table_1 [3,3], Table_2 [4,2]
5	Table_2 [4,3]
Aviator	Table_1 [3,1], Table_2 [4,1]
Biography	Table_1 [3,2]
date	Table_2 [1,2]
genre	Table_1 [1,2]
oscars	Table_2 [1,3]
The	Table_1 [3,1], Table_2 [4,1]
title	Table_1 [1,1], Table_2 [1,1]
year	Table_1 [1,3]

Figure 3.2: WebTables index populated with the contents of Table_1 and Table_2

Searching and Ranking

The search algorithm uses the same principles of searching in regular search engines. For conjunctive queries with multiple keywords, the search engine steps through the keywords inverted lists in parallel, testing to see where they share a table. For disjunctive queries with multiple keywords, the result is the union of tables found inside the keywords inverted lists.

The offset can also be explored in a similar manner. While regular search engines use the offset to find adjacent terms, the WebTables offset opens the possibility to use spatial operators. For example, operators like samecol and samerow could be used in order to return results only if the search terms appear respectively in cells in the same column or in cells of the same row of the table.

The ranking of the results is based on the set of features described below.

- Number of rows in table
- Presence of a table header
- Number of null values inside table
- Presence of searched keywords on header (column names)
- Presence of searched keywords on leftmost column
- Presence of searched keywords on second-to-leftmost column
- Presence of searched keywords on any table cell

The relevance of each feature was estimated through experimentation. The results show that the best features are 1) presence of searched keywords on header and 2) Presence of searched keywords on leftmost column. The former indicates that attribute labels are a strong indicator of the tables meaning. The latter indicates that values in the leftmost column are usually the most representative, providing a useful summary of the contents of a data row.

During the keyword search, information about these two features can be extracted from the index offset. Offsets of the format [1,*] indicate the keyword appears on the header, while offsets of the format [*,1] indicate the keyword appears in the leftmost column.

Besides the mentioned set of ranking features, the authors propose a metric based on the schema coherence score. A schema is considered coherent if its attributes are all tightly related to one another. The experiments show that using this information for ranking outperforms the approach that is based only on the features presented above.

The coherence score between two attributes is given by the Pointwise Mutual Information (PMI), which is a information often used in computational linguistics and web text research (CHURCH; HANKS, 1990; ETZIONI et al., 2004; TURNEY, 2001). To compute the PMI between attributes a and b, some probabilistic information are required: p(a) and p(b), which indicate the probability of finding the attributes a and b in the schema corpus, respectively; and p(a,b), which indicate the probability of finding both attributes together in the schema corpus.

These probabilities are easily derived from the attribute correlation statistics database (ACSDb). The ACSDb is simple. It contains a set of pairs of the form(R,c), where R is a schema and c counts the number of times this schema appears in the corpus.

As detailed by the authors, information from the ACSDb is useful for many applications. In the paper, besides ranking of results, they also show applications for schema auto completion, attribute synonym finding and navigation through related schemas. As a drawback of this approach, structure is not taken into account for the purpose of answering structured queries. Consequently, selection predicates are not allowed, and the results are limited to information encountered on single tables.

3.2.4.2 ATIL and AAIL

Dong and Halevy (DONG; HALEVY, 2007) propose a solution that is able to perform queries with structure. The indexing is based on keywords that point to a list of instances where the keywords can be found. The index can be enriched with hierarchical information (terms that can substitute a keyword) and association labels, so that information from multiple sources can be combined in the result. The following sections further describe this approach.

Data Model

Data is modelled as a set of triples. Each triple is either of the form (instance, attribute, value) or of the form (instance, association, instance). An instance is a real-world object composed by attributes (and its respective values). An association is a relationship between two instances, and it is represented by a role. Also, associations are assumed to be directional.

Figure 3.3 illustrates how information sources Table_1 and Table_2 are represented in this model. The information of the sources is structured as tables, but the model can be applied over any type of schema composed by attributes and values.

The set of triples (the bottom of Figure 3.3) was extracted from the information

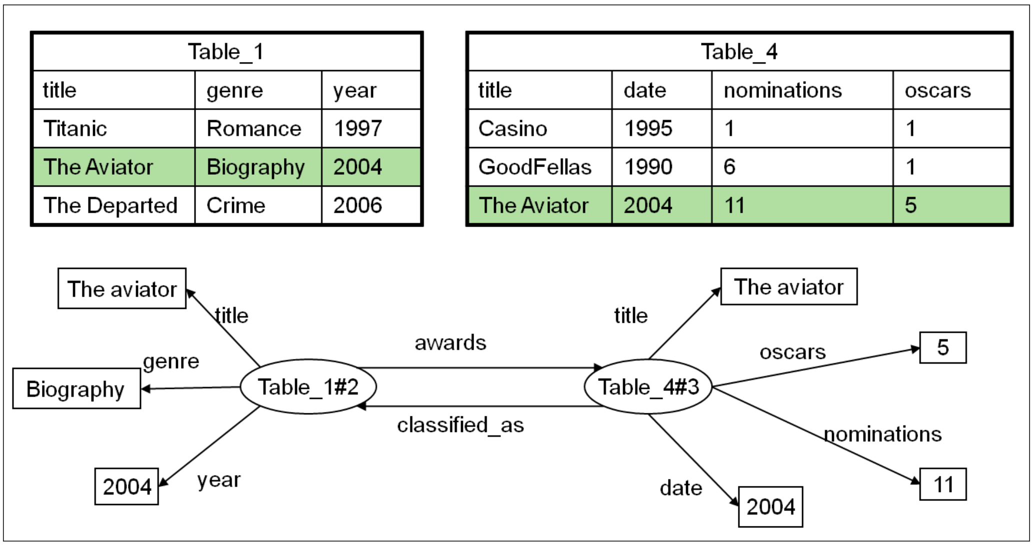


Table_1

title	genre	year
Titanic	Romance	1997
The Aviator	Biography	2004
The Departed	Crime	2006

Table_4

title	date	nominations	oscars
Casino	1995	1	1
GoodFellas	1990	6	1
The Aviator	2004	11	5

Figure 3.3: Example that illustrates how relations are modelled in (DONG; HALEVY, 2007)

sources using techniques adapted from (GUBANOV; BERNSTEIN, 2006; ETZIONI et al., 2008). We only illustrate the triples that correspond to the marked rows of the tables. Note that each row represents an instance. For clarity, we identified the instances by the concatenation of the source name and position inside the source.

Besides the relation between attributes and values, the extraction is also responsible for detecting the associations between the instances, which may be hard to derive automatically. In the example given, it is hard even for a human to determine the association between Table_1 and Table_2, since the tables are not normalized with respect to each other. The limitations of automatically inferring the associations is not discussed in the paper.

Query Language

Two types of queries are supported: Predicate Queries and Neighborhood Queries.

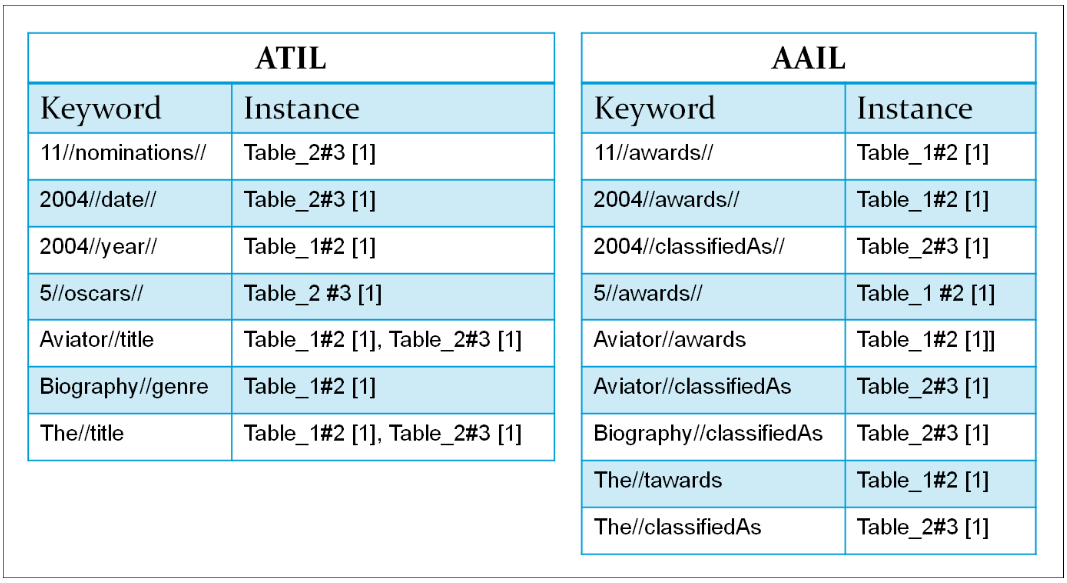
A predicate query contains a list of predicates, where a predicate is composed by a verb (either an attribute name or an association name) and a list of keywords. For instance, a query asking for Tarantino movies in 2005 can be represented as: (Director 'Tarantino'), (Year '2005').

A neighborhood query is simply a list of keywords. A neighborhood query is just a fancy name for plain keyword queries, and they indicate that the language support queries without predicates.

Indexing and Querying

Both values and structure of the sources are captured using extended inverted lists. Two main inverted lists were proposed (ATIL and AAIL), each designed for a specific kind of query. In general, they connect keywords to the list of instances where the keywords can be found.

In ATIL, the keyword is a concatenation of a value and its attribute. Figure 3.4 shows how this index works. In the keyword values, the string "//" is a marker reserved for indexing purposes. The square brackets that accompany each instance indicate the number of occurrences of the keyword inside the instance. In the example given, none of the indexed keywords contain a number of occurrences greater than one. Just to demonstrate a different situation, say that the title attribute of instance Table_1#2 contains 'The Aviator - The Movie'. In this case, the keyword 'the//title' would have a number of occurrences equal to two for that instance.



ATIL	
Keyword	Instance
11//nominations//	Table_2#3 [1]
2004//date//	Table_2#3 [1]
2004//year//	Table_1#2 [1]
5//oscars//	Table_2 #3 [1]
Aviator//title	Table_1#2 [1], Table_2#3 [1]
Biography//genre	Table_1#2 [1]
The//title	Table_1#2 [1], Table_2#3 [1]

AAIL	
Keyword	Instance
11//awards//	Table_1#2 [1]
2004//awards//	Table_1#2 [1]
2004//classifiedAs//	Table_2#3 [1]
5//awards//	Table_1 #2 [1]
Aviator//awards	Table_1#2 [1]]
Aviator//classifiedAs	Table_2#3 [1]
Biography//classifiedAs	Table_2#3 [1]
The//tawards	Table_1#2 [1]
The//classifiedAs	Table_2#3 [1]

Figure 3.4: The indexes ATIL and AAIL populated with the contents of Table_1 and Table_2

To answer a predicate query, each predicate is looked up in the ATIL. The returned instances need to satisfy at least one predicate in the query. An instance satisfies a predicate if it contains at least one of the keywords of the predicate. For example, the query (title 'The Aviator'), needs two index lookups (The//title and Aviator//title). The returned instances would be Table_1#2 (there is a match for keywords 'The' and 'Aviator'), Table_1#3(there is a match for keyword 'The') and Table_2#3 (there is a match for keywords 'The' and 'Aviator'). Answering a neighborhood query uses the same principle, only the attribute part in the ATIL is ignored.

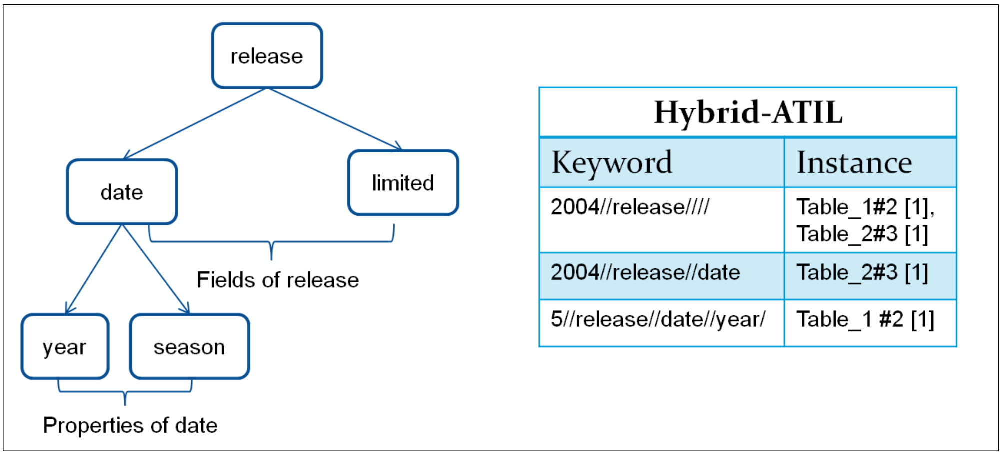
The authors comment about the possibility of dealing with conjunction of predicates, but do not explore this feature in the paper. Considering conjunctive queries, the query (genre 'Biography'),(title 'The Aviator') would return only Table_1#3, since it is the only one that satisfies both predicates.

The AAIL is an extension of the ATIL that includes keywords formed by a concatenation of a value and its association (the "role"in the model). Figure 3.4 shows the additional entries that are considered by the AAIL.

With this extension, a larger number of queries can be answered, i.e. those that are described using the association names represented in the model, such as (genre 'Biography'), (Awards '5').

Another interesting idea is to think of the indexed terms as part of an hierarchy of concepts, and to use this information in the index. Two types of hierarchies are considered: sub-property type (e.g., director being a sub-property of crew) and sub-field type (e.g., year being a sub-field of movie). No distinction is made about the types, they can be mixed in the same hierarchy. Several hierarchy-aware indexes were proposed (Dup-ATIL, Hier-ATIL, Hybrid-ATIL). In here, we demonstrate the Hybrid-ATIL, which subsumes the others as for the optimizations/benefits introduced.

Figure 3.5 shows part of the Hybrid-ATIL on the right. The year attribute is indexed according to the hierarchy in the left of the Figure.



Hybrid-ATIL

Keyword	Instance
2004//release////	Table_1#2 [1], Table_2#3 [1]
2004//release//date	Table_2#3 [1]
5//release//date//year/	Table_1 #2 [1]

Figure 3.5: The Hibrid Atil populated with the contents of Table_1 and Table_2

In Hybrid-ATIL, the hierarchy becomes part of the keyword, as the second and third row indicate. This makes it possible for hierarchical queries to be answered, such as (release/date ’2004’).

The first row is called a summary row, because it shadows rows 2 and 3. Note the extra "//"at the end. Summary rows improve index search for queries like (release ’2004’), by stoping the search after a summary row is found (no additional entries need to be accessed). Note that the second row could also be a summary row by shadowing the third row. In this case, the summarization would start at the second hierarchy level. The number of hierarchy level that can contain summaries is a parameter of the indexation algorithm.

3.3 Chapter Remarks

This chapter discusses ways to implement no-schema approaches, where the ultimate goal is to integrate data sources with the less possible effort. We call attention to the fact that some of these approaches do contain a global schema, like the universal relation and pay-as-you-go approaches. Still they are classified as no-schema attempts because the schema is generated automatically.

According to current works, the universal relation was not considered as a general purpose solution for querying on the Web. The application of this solution seems to be appropriate for single domains, with a small number of relations. As for pay-as-you-go approaches, experiments are also restricted to single domains, and mostly dedicated to discovering one-to-one mappings - cases involving joins are not considered. Keyword search over relations is also limited, as the original idea applies to relational databases, which contains a restricted number of relations.

Indexes based on inverted list appears as a recent attempt to handle querying over large and distribute structured data sets. Usually explored in the context of keyword based search engines, indexes are easy to build, maintain and access. Besides, they are suitable data structures able to accommodate the diversity and volume of data found on the Web. Along this path, related work reaches a point where rich models of interconnected sources are possible to represent as indexes. However, it is still not clear how these models are created automatically. Moreover, querying capabilities still need improvements, such as the ability to answer predicate queries.

4 SYSTEM ARCHITECTURE

In order to query dataspaces, we envision structured search engines that support simple, yet expressive queries, which allows users to explore Web information, discover useful information sources and their relationships. The no-schema approach is adopted. Therefore, considering that data relationships are most of the time implicit (they are not declared anywhere), part of the problem involves finding these associations in order to deliver more useful information to the user.

Figure 4.1 shows our architecture, which we named Eidos[1]. Three layers compose the architecture. The data layer intermediates communications between the sources and the rest of the system. The Mediator layer contains components responsible for the integration itself. It is sub-divided into the Rewriter Layer, which subsumes the components dedicated to the rewriting process.

The components depicted in the Figure are presented in the next sections. Special attention is dedicated to the components of the Rewriter Layer, which represent the focus of this thesis.

4.1 Crawler

The Crawler is a Mediator Layer component responsible for finding relevant structured Web sources. Similar to a traditional search engine, such as Google and Yahoo!, we use a Web crawler to locate structured information on the Web and index them into the Eidos repository. The system can be set to periodically crawl the Web to update the index as well as to discover new information sources. The information will then be immediately available to the users.

Because structured information is sparsely distributed on the Web, we use a focused crawler for this task (BARBOSA; FREIRE, 2007a). The crawler uses templates to guide the crawling. Templates indicate whether a given document should be considered for indexing and even as a source of additional documents to crawl (such as page links).

Several templates are possible. For instance, the crawler can focus on files of a given type, like HTML or XML. Also, more sophisticated templates can be provided, such as those that are based on the contents of documents. Details about the patterns that guide the focused crawling can be found in the Experiments Chapter.

[1] The name is a reference to the term used by Plato to indicate that the shape of things is not always associated to their outside appearance.

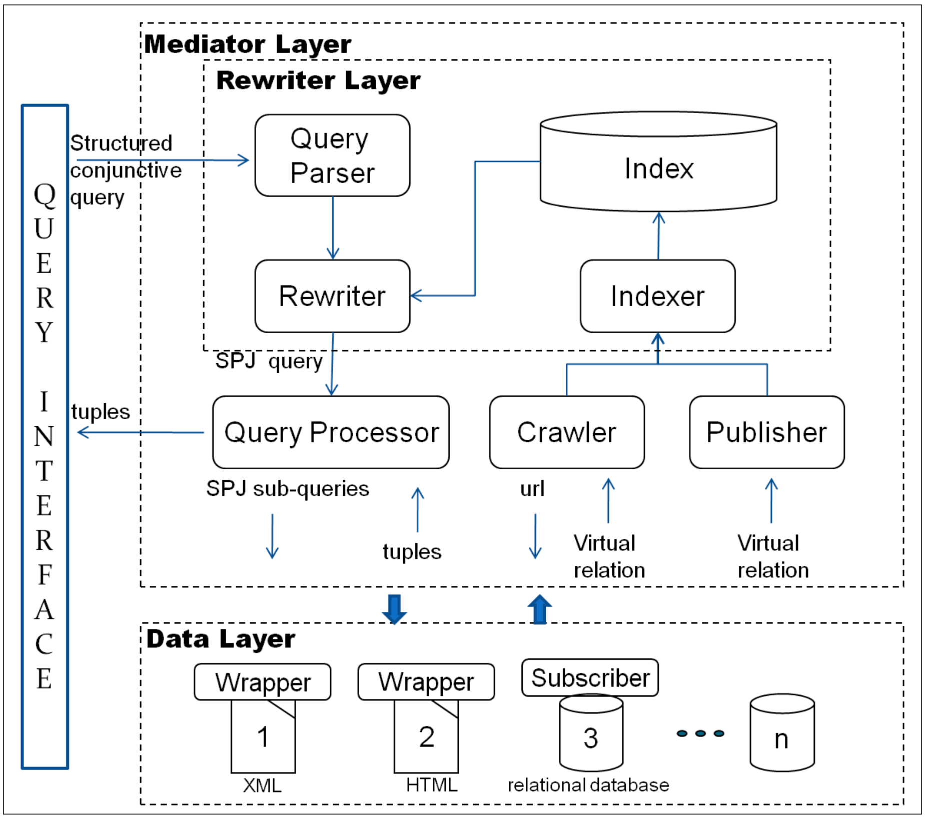


Figure 4.1: High-level architecture of Eidos

4.2 Wrappers and Sources

The internal data model used by the Eidos architecture follows the relational paradigm. Requests that start from the Rewriter or the Query Processor are expressed as Select-Project-Join SQL queries. Besides, answers are delivered as tuples.

Given this, and considering that sources may be heterogeneous (and even semi-structured), wrappers are necessary in order to make the data available to the system, by abstracting each source as one or more virtual relations.

Wrappers are activated in two different moments: indexing and querying. During indexing, the crawler selects a source and asks for the wrapper to compute virtual relations out of it. The computed relations are then stored in the repository. During querying, the query processor asks the wrapper to answer an SQL query over a selected source. The wrapper then computes the virtual relations in order to answer the issued query, and delivers the tuples obtained as answer.

A virtual relation is composed by a schema t, a list of attributes $\{c_1, ..., c_n\}$ and a list of tuples. To understand the concept of tuple, consider that the extension of a virtual relation $t = \{c_1, ..., c_n\}$ is $E(t) = \{DOM(c_1), ..., DOM(c_n))\}$, where $DOM(c)$ denotes the domain of attribute c. Given this, a tuple τ of t is a list of domain values $\{d_1, ..., d_n\}$ such that $d_i \in DOM(c_i)$, where $c_i \in t$. To simplify, we use the terms wrapped relations(or wrapped sources) and wrapped attributes to refer to the virtual relations and attributes that are constructed by a wrapper.

The relational model was chosen to be part of the architecture because relations provide a natural representation for a significant number of structured Web sources. One particularly interesting example are HTML Tables. A recent study reports that there are over 144 million relations published as HTML tables in Google's index.

Besides, several works in the literature show how to extract lists of records that are present in HTML tables (COHEN; HURST; JENSEN, 2002) and even other kinds of HTML constructs. Other sources(e.g. web services, free text, XML,...) can also be wrapped as relations, using correct and sufficient abstractions (LAENDER et al., 2002; DALVI; KUMAR; SOLIMAN, 2011). For example, a simple approach to wrap XML is to consider that every non-leaf element is a relation, and the hierarchy of non-leaf elements indicates foreign key constraints (KAPPEL; KAPSAMMER; RETSCHITZEGGER, 2004). Observe that additionally to other (semi-)structured sources, relational databases are natively supported by Eidos. In this case, the wrapper serves merely as an interface to the underlying database. The wrapping of relational databases is discussed in the next section.

4.3 Subscriber and Publisher

Similarly to wrappers, subscribers are data layer components. They are able to wrap a source as one or more virtual relations, which enables the subscribed source to communicate with the mediator. The difference is that subscribed sources register themselves into the system, while wrapped sources need to be crawled. This type of component is useful for sources that are not reachable by the crawler, such as relational databases.

Another difference is that wrappers are server-side components. Communication between wrappers and the sources occur over the Internet. On the other hand, a subscriber sits next to the registered source, which makes of it the only client-side component. Therefore, for the mediator to access the source, the subscriber needs to be installed and resident in the machine where the sources are located.

The publisher is the Mediator Layer component that registers the sources wrapped by a subscriber. The registration occurs by selecting the information that needs to be made available to others. To exemplify, in the case of relational databases, the user could publish the relations of interest only, instead of publishing all of them. Additionally, the subscriber can be set to periodically register itself, so the indexed information is more up to date.

Google's Fusion Tables is a product that uses a subscriber related approach. However, instead of keeping the source in the client, the user needs to upload it into Google's index, and associate it with keywords that indicate the contents of the source (GONZALEZ et al., 2010). Compared to this approach, our solution is more dataspace-oriented, since wrappers are able to capture the structure of the sources, which is more meaningful then keywords. Besides, it takes the concept of information sources one step further, by allowing anything to be published, as long as appropriated wrappers are available.

4.4 Query Processor

The query processor receives relational queries from the Rewriter Layer. Received queries are composed by a set of relations, where each relation may come from a different source. It is up to the query Processor to break the query into sub-queries according to the involved relations, forward the sub-queries to the identified sources, combine the tuples from each sub-query as a single result, and return the resulting tuples to the query interface.

This is very similar to the problem of answering queries over multi-databases, and faces the same challenges, such as those related to query plan optimizations. For instance, consider the query below:

SELECT
    s_1.title, s_1.genre, s_1.director, s_2.awards
FROM
    source1 s_1, source2 s_2
WHERE
    s_1.title = s_2.title and s_1.year $<$ 2000

Sources 1 and 2 are distributed, so it is not possible to answer the query with a single select. Instead, the query processor needs to separate the constructs related to attributes of source1 in one select statement, and constructs related to attributes of source2 in another statement. Query plans for distributed queries can be represented as a relational algebra tree that joins the statements that are particular to each distributed source. In order to build efficient plans, part of the problem involves pushing operators down in the tree and using advanced join algorithms to combine the data from different sources.

Multi-database querying is a classical problem in the field, with many years of research. Therefore, it is not our goal to develop new solutions to a well known problem. Instead, we employ the state of the art techniques, which have proven to be efficient (KOSSMANN, 2000). This enforces the decision of using the relational paradigm as our underlying data model. By treating queries as SQL and sources as relations, we start from a solid foundation with consolidated techniques, some of which are useful to our goal, such as those needed by the Query Processor component.

4.5 Query Rewriting

The Rewriter layer is the core of the Eidos architecture, as it contains the components that form the general rewriting mechanism. These components are isolated from the other Mediator components, which makes it possible to change the rewriting strategy without interfering with the rest of the architecture.

Three components are available. The Indexer is activated during indexing time. It receives information from the crawler and the publisher (virtual relations from the sources) and decides how to store it into the Index.

The other two components are activated during querying time. The Query parser receives a structured conjunctive query and transform it into a format expected by the Rewriter. On its turn, the Rewriter is responsible for finding queries that use the available sources. As presented earlier, the queries may englobe distributed wrapped relations. The relations are selected based on the information provided by the index.

The focus of this thesis is dedicated to the components of the rewriter layer. The next two chapters present details about two approaches we have proposed. Each approach handles the problem in different ways, providing specific languages for structured conjunctive queries, and complementary index and rewriting algorithms. These components are designed according to the architecture of Figure 4.1, so they can be easily coupled as part of the Mediator.

In this section, we present general information that apply to the rewriting idea as a whole. The two rewriting approaches detailed in the next chapters follow the principles stated below.

Considering that the sources of interest are distributed over the Web and belong to autonomous sites, our work falls into the Open World Assumption (OWA) category. Among other things, this means that the answer to a query may be spread over a large number of independent sources.

In this context, there may be many possible ways to answer a user query, each of them using a different set of sources and yielding different answers as the result. We call each individual way to answer a query a Rewriting.

Our goal is to reach the larger number of rewritings possible. One possibility would be to provide the answer as a union over all computed rewritings. However, we decided to present to the user each rewriting in isolation. Several reasons motivate this decision:

1. Depending on the size of the data set, the cost for computing all answers becomes prohibitive. Besides, the user would normally focus on the first results only.

2. It is easier to relate the resulting tuples to their respective sources. Knowing the provenance of the tuples helps to identify the best sources of information.

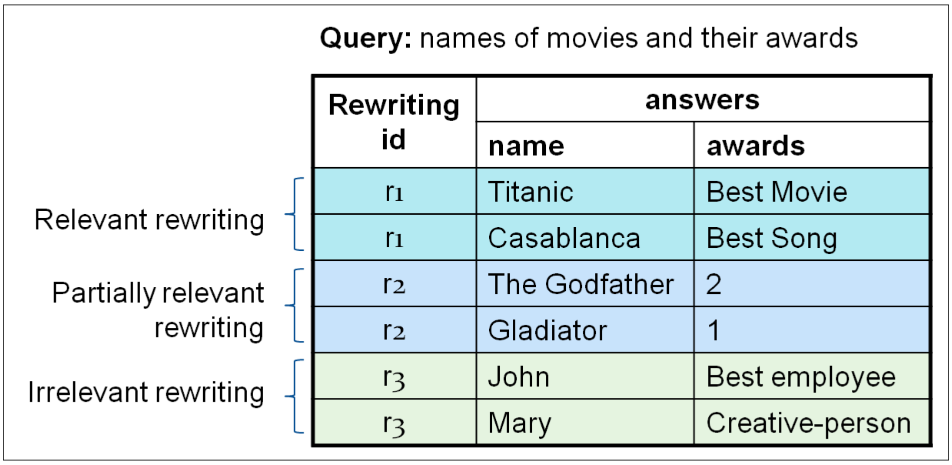
3. It is possible to associate a specific rewriting to answers understood as irrelevant by a user and ignore all answers that come from this rewriting.

4. It is possible to conceive a feedback mechanism, where the user is able to inform that the answers of a specific rewriting are not correct. Based on this information, the system could adapt the rewriting in order to enhance accuracy.

A example that helps to illustrate these reasons is shown in Figure 4.2. The query asks for names of movies and their awards (for clarity, the example shows a query described in natural language). The list shows answers from three rewritings, with two tuples each. Tuples from r_1 refer to known movies and awards, so the user may assume that all tuples from this rewriting can be trusted. Tuples from r_2 also refer to known movies. However, the award information is related to the amount, not the title. In such cases, the user may still be interested in this rewriting, since it is at least partially correct. On the other hand, tuples from r_3 clearly refer to another domain - the data shows rewards given to employees as a proof of recognition. Intuitively the user can ignore all other tuples that are grouped under this rewriting.

The main goal of our architecture is to integrate heterogeneous, autonomous and distributes sources, and to allow users to access them through a single querying interface, which are characteristics of a typical data integration system.

However, the example of Figure 4.2 shows an important aspect about the proposed system, and that we emphasize here: rewritings are found based on an inference mechanism, and not on pre-defined mapping information. Consequently, answers of queries are uncertain by nature.



Query: names of movies and their awards

	Rewriting id	answers	
		name	**awards**
Relevant rewriting	r1	Titanic	Best Movie
	r1	Casablanca	Best Song
Partially relevant rewriting	r2	The Godfather	2
	r2	Gladiator	1
Irrelevant rewriting	r3	John	Best employee
	r3	Mary	Creative-person

Figure 4.2: Abstract example that shows possible answers of rewritings

This fact represents a gap that distinguishes our architecture from traditional integration systems, which are built under the notion that all answers obtained are correct. Therefore, our approach cannot be classified as a pure integration system.

Instead, we consider our architecture as a mix of a data integration system and a search engine. In search engines, like Google, the users are aware that answers may be incorrect, and have to deal with it properly, by analyzing the results and adapting the queries if desired. This is exactly the expected behavior in our case too. The main difference is that traditional search engines aim at answering keyword based queries and present results as lists of documents, whereas we support structured queries and results as lists of tuples.

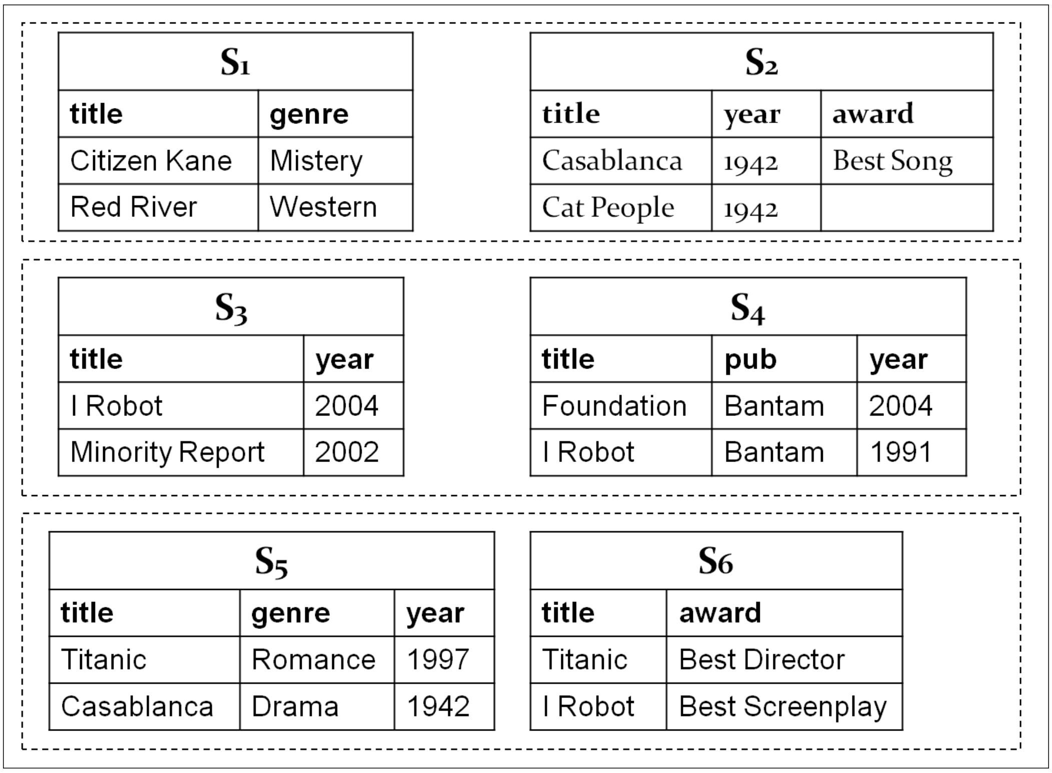
It is important to notice that the theory of query answering adopted by integration systems does not fully apply here. For instance, query containment is a subject that arises when LAV mappings are used. Since Eidos has no pre-defined mappings, it makes no sense to check for containment. When the systems computes n rewritings for a query, the union of these n rewritings does not represent a maximally contained rewriting. Instead, they solely represent the sum of collections of tuples, where one given collection may be more relevant to the user than the other.

The concept of certain answers is also irrelevant to our goal. It is useful to determine untractable classes of queries for incomplete databases. Integration systems use this knowledge to limit the support to queries that are tractable, such as conjunctive queries. In our cases, answers are already known to be unprecise. There is no need to limit the querying capability, since any type of query will deliver uncertain answers.

However, as Figure 4.1 shows, we did restrict the support to conjunctive queries. This decision was made just as a way to define the boundary of our work, and not based on tractability issues (as they do not apply). This is not a significant limitation though, since conjunctive queries are expressible enough, as they represent the most common form of relational queries (CHEKURI; RAJARAMAN, 1997).

4.6 Running Example

Figure 4.3 shows the data sources that will serve as the basis for running examples discussed throughout the thesis. The sources are presented as relations, where information is exposed as a set of tuples. Note that the original model of the sources is irrelevant to our discussion. We start from the assumption that wrappers are available to expose the concrete information as sets of relations.



S1	
title	genre
Citizen Kane	Mistery
Red River	Western

S2		
title	year	award
Casablanca	1942	Best Song
Cat People	1942	

S3	
title	year
I Robot	2004
Minority Report	2002

S4		
title	pub	year
Foundation	Bantam	2004
I Robot	Bantam	1991

S5		
title	genre	year
Titanic	Romance	1997
Casablanca	Drama	1942

S6	
title	award
Titanic	Best Director
I Robot	Best Screenplay

Figure 4.3: Data sources transformed into wrapped relations

The dotted rectangle groups sources that come from the same location. For instance, for HTML documents, the location could be the Web site, while for a relational table it corresponds to its respective relational database.

All sources contain information about movies, except S_4, which brings information about books. In fact, sources S_3 and S_4 belong to the same location and group information related to Isaac Asimov works in general (books and movies based on books). This variation in the domain shows one of the expectations of our work, which is to be able to integrate information from similar domains.

Also, observe that some objects have information spread over different sources, and some sources that contain complementary data come from different locations. For instance, some attributes of the movie *Casablanca* are exposed in S_2, while others are exposed in S_5. It is easy to see that the same object is represented in different wrapped relations, since they have the same value for the attribute *title*.

The existence of heterogeneous, distributed and correlated sources represent some of the aspects found on dataspaces. In the next two chapters we investigate approaches that are able to bring this information to the surface and allow users to express structured queries over it.

5 QUERY ANSWERING IN THE PRESENCE OF JOIN PREDICATES

In this chapter we show how to access dataspaces through an interface that accepts queries with joins. With this type of query, the user explicitly defines the relations of interest, along with the necessary join conditions. To support queries with joins, we create particular indexing and rewriting components of the Eidos architecture. Together, these components are responsible for two main processes: 1) find which wrapped sources map to the relations of the user queries, and 2) use them accordingly in order to find the answers.

The remainder of the chapter is organized as follows. In Section 5.1, we present the query language that allows users to ask queries with joins. In Section 5.2, we present the index component, which involves the structure of the index and the mapping functions that populate the index with information from the sources. Section 5.3 discusses several algorithms that belong to the rewriting component. These algorithms access the indexes in order to find rewritings to user queries.

5.1 Query Language

As explained earlier, we need a language that allows the user to explicitly declare joins. Also, Chapter 4 described that the Eidos architecture only accepts query languages restricted to conjunctions of predicates. Therefore, in order to satisfy the Eidos architecture restrictions, and to provide support for joins, we let the user pose select-project-join(SPJ) queries described in SQL.

This class of queries is relatively easy to express. Besides, they represent a large part of queries issued on relational databases. For instance, take the SQL query described below. This query is asking for movies in the twentieth century that received an award:

```
SELECT
    m.title, m.genre, a.award
FROM
    movie m, award a
WHERE
    m.title = a.title and m.year >= "1900"and m.year < "2000"
```

Although our query interface accepts conjunctive queries in SQL, for clarity we explain the concepts and components of our work using the Datalog notation. In

Datalog, a SPJ query has the following form:

$$h(X) : -g_1(X_1), ..., g_n(X_n), f_1, ..., f_m$$

A query has two parts, divided by the symbol ":-". The part in the left of the symbol is called header. The header, identified by the atom $h(X)$, represents the projection part of the query. The part in the right of the symbol is called body. The atoms in the body represented by each of the $g(X)$ are called relation subgoals.

A relation subgoal refer to a relation. The arguments of type $X = \{x_1, ..., x_n\}$ refer to the attributes of the relations, and are all variables (constants are irrelevant to our discussion).

In order for the query to be safe, it is required that $X \subseteq X_1 \cup \cdots \cup X_n$, that is, the variables of the header must be also in at least one of the subgoals.

To complete, each f_i is a comparison subgoal in the form $\alpha\ \theta\ \nu$, where α is a variable that appears in a relation subgoal, ν is a constant and θ is an operator that accepts $\neq$, =, >, >=, <, <=.

In datalog, the SQL query presented earlier is represented as:

```
h(title,genre,award)  :-  movie(title,genre,year),
                          award(title,award),
                          year >= 1900, year < 2000
```

The variables of the header represent the attributes of the select clause. The body is composed by four subgoals, where two of them correspond to relations and the others are comparison subgoals.

Variables that are present in multiple subgoals are called shared variables. They correspond to either join variables, when they appear in more than one relation subgoal (like *title*) or selection variables, when they appear in selection subgoals (like *year*). Variables that are not shared are called local.

Additionally, variables that appear in the header are called distinguished variables. Variables that are not distinguished are said to be existential.

5.2 Proposed Indexes

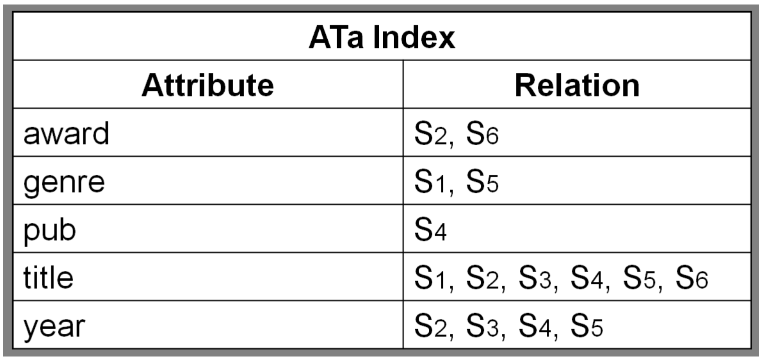
In this section we present several indexes that store information from the wrapped relations. In addition, we explain the mapping rules responsible for extracting information from the wrapped relations into the repository. The wrapped relations presented in Figure 4.3 (S_1 to S_6) are used to exemplify the mapping process.

5.2.1 A Index

The A index([A]ttribute index) is composed by a list of indexed attributes $A = \{a_1, ..., a_n\}$.

Alone, this index does not help in the rewriting process. All it can do is answer the question of weather the variables of the user queries exist in wrapped relations or not, without identifying which are the relations. However, its structure compose the foundation of more meaningful indexing structures, as will be described later.

A wrapped relation is mapped to the A index according to Definition 1.



ATa Index	
Attribute	**Relation**
award	S2, S6
genre	S1, S5
pub	S4
title	S1, S2, S3, S4, S5, S6
year	S2, S3, S4, S5

Figure 5.1: ATa index populated with the contents of sources S_1 to S_6

Definition 1 *(A Mapping)*
Consider the wrapped attribute $c \in t$. Given this, let $mapA(c) = a$ be a function that maps c to an indexed attribute a.

The function $mapA$ computes the map based on the name of the wrapped attribute. Several techniques can be used, such as a combination of string matching and lexicons. Our purpose is not to discuss the possibilities, but to show that they exist. For instance, one function could map the wrapped attributes *title* and $film$ to the same canonized indexed attribute *movie*. To simplify, in this work we determine that homonymous wrapped attributes map to the same indexed attribute.

5.2.2 ATa Index

The ATa index([A]ttribute-[Ta]ble index) extends the A index with the list of indexed relations $R = \{r_1, ..., r_n\}$. Furthermore, $rel(a) = R_x$ is the function that returns the list of relations that contain a, where $R_x \in R$.

The goal of the ATa is to allow relation lookups. Given a query, it provides the location of the relations that contain the variables of the relation subgoals.

A wrapped relation is mapped to the ATa index according to Definition 2.

Definition 2 *(ATa Mapping)*
The ATa Mapping extends the A Mapping to support the relation level of the ATa. Provided that, consider the wrapped attribute $c \in t$. Given this, $mapR(c, t) = r$ is the function that maps the wrapped relation t to the indexed relation r. Furthermore, considering the maps $mapA(c) = a$ and $mapR(c, t) = r$, then $r \in rel(a)$.

To illustrate the mapping, Figure 5.1 shows the ATa populated with the contents of the sources from S_1 to S_6. Basically, ATa is an inverted list that allows navigation from the attribute to the relation level. For clarity, the index is presented in the Figure as a table. The physical implementation can be achieved in multiple ways (BAEZA-YATES; RIBEIRO-NETO, 1999), such as a sorted array, a prefix B-tree or a Patricia trie. Alternatively, compression techniques could be used to shrink the index and reduce the storage requirements (HERSH, 2001).

5.2.3 ATaVa Index

The ATaVa index([A]ttribute-[Ta]ble-[Va]lue) extends the ATa with the triple (D, V, I).

The element $D = \{d_1, ..., d_n\}$ denotes the list of data types of each pair (a, r) such that $a \in A$ and $r \in rel(a)$. Complementarily, the function $dty(a, r) = d_x$, where $d_x \in D$, returns the data type for a specific pair (a, r).

In addition, $V = \{v_1, ..., v_n\}$ is the list of indexed values of each pair a, r. The function $val(a, r) = V_x$, where $V_x \in V$, is the function that returns the list of indexed values related to the pair (a, r).

Finally, $I = \{i_1, ..., i_n\}$ contains the tuple identification of each triple a, r, v. This identification is the vertical coordinate of a value v inside a indexed relation r, for attribute a. The function $ide(a, r, v) = i$, where $i \in I$, returns the respective tuple identification.

A wrapped relation is mapped to the ATaVa index according to Definition 3.

Definition 3 *(ATaVa Mapping)*
The ATaVa Mapping extends the ATa Mapping to support the values level of the ATaVa. Provided that, consider the wrapped attribute $c \in t$ and the domain value $d \in DOM(c)$ that belongs to the tuple τ of table t. Given this, let $mapV(c, t, d) = v$ be a function that maps the domain value d to the indexed value v. Furthermore, considering the maps $mapA(c) = a$, $mapR(c, t) = r$ and $mapV(c, t, d) = v$, then $v \in val(a, r)$. Finally, $ide(a, r, v)$ and $dty(a, r)$ are given by the functions $ide(\tau)$ and $dty(DOM(c))$, respectively.

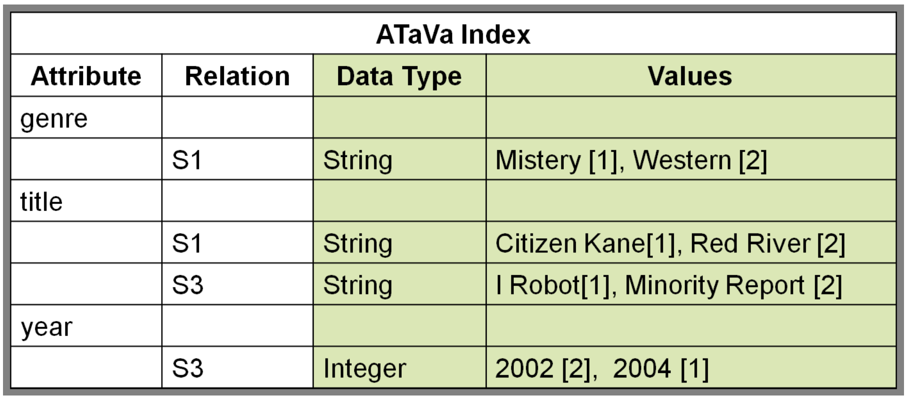
The computation of $ide(\tau)$ must be provided by the Wrapper module, since the result depends on how the wrapper extracts information from the underlying data model of the wrapped source. For instance, considering HTML tables, the wrapper could extract one tuple for each record enclosed by the tag <tr>. In this case, the function $ide(\tau)$ could return the order of the record inside the table as the identifier. A similar approach can be used for indexing relational tables. Even though relational tuples have no particular order inside a relation, queries over the relation will return the tuples in a specific order, even if arbitrary. In indexing time, relation wrappers can submits queries to obtain the tuples, and then use the result set order as the tuple identification.

The data type of an attribute $dty(DOM(c))$ is also related to the underlying data model of the wrapped source. For some sources, like relational tables, the data type can be obtained directly. For other sources, like Web data, this information is normally not defined, but can be easily discovered through a template based approach. For instance, FOCIH also has a library of regular-expression recognizers for values in common formats (EMBLEY et al., 1999).

To illustrate the mapping process, Figure 5.2 shows the ATaVa populated with the contents of the sources s_1 and s_3. The dashed part indicates information that is new, if compared to the ATa. As its predecessor, the ATaVa index also provides the location of the relations that contain the query variables. However, there is a new level of information concerning the tuples of the wrapped relations.

The value inside the square brackets identifies the tuple from which the specified value can be obtained. For this mapping, we assume that the wrapper uses the order presented in Figure 4.3 as the tuple identification.

As the Figure shows, two data types appear in the index (String and Integer), which can be easily inferred from the sources using a simple template. We do not constraint the list of data types allowed. Implementations of the index can define the data types as it see fits, and use them accordingly.



ATaVa Index			
Attribute	Relation	Data Type	Values
genre			
	S1	String	Mistery [1], Western [2]
title			
	S1	String	Citizen Kane[1], Red River [2]
	S3	String	I Robot[1], Minority Report [2]
year			
	S3	Integer	2002 [2], 2004 [1]

Figure 5.2: ATaVa index populated with the contents of the sources S_1 and S_3

The values level is useful for answering queries with arithmetic comparisons, since it helps locating relations that are able to satisfy the selection predicates. Additionally, the tuple identification can be used when the query has multiple selection predicates, as it enables the process of verifying whether any specific tuple of a relation is able to satisfy all predicates. Finally, data types add semantic to the process of resolving query predicates, by allowing the rewriting algorithms to compare values according to their respective data type.

5.2.4 TAt Index

The TAt index([T]able [At]tribute) is a tuple (R^{tat}, A^{tat}), such that $R^{tat} = \{r_1^{tat}, ..., r_n^{tat}\}$ is the list of indexed relations and $A^{tat} = \{a_1^{tat}, ..., a_n^{tat}\}$ is the list of indexed attributes.

Furthermore, $attr(r^{tat}) = A_x^{tat}$ is the function that returns the list of attributes of r^{tat}, where $A_x^{tat} \in A^{tat}$.

A wrapped source is mapped to the TAt index according to Definition 4.

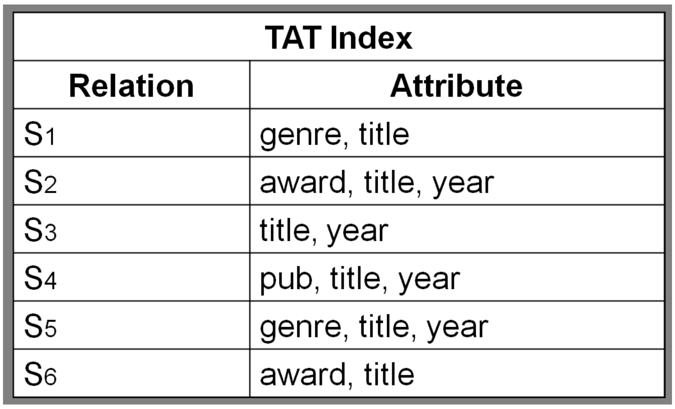
Definition 4 *(TAt Mapping)*
Consider the wrapped column $c \in t$. Given this, let $mapR(t) = r^{tat}$ be a function that maps t to an indexed relation r^{tat}. Additionally, the function $mapA^{tat}(t, c) = a^{tat}$ maps the wrapped attribute c to the indexed attribute a^{tat}, such that $a^{tat} \in attr(r^{tat})$.

To illustrate the mapping, Figure 5.3 shows the TAt populated with the contents of the sources s_1 to s_6. TAt is an inverted list that allows navigation from the relation to the attribute level. In this sense, it is orthogonal to ATa/ATaVa, which offers a navigation in the opposite direction.

TAt allows attribute lookups. Together with the relation lookups of ATa, these two indexes accelerate the process of finding rewritings, as the next Sections demonstrate.

It is possible to merge the ATa/ATaVa with the TAt as follows: Consider the column $c \in t$, and the maps: $mapA(c) = a$, $mapR(c, t) = r$, $mapR^{tat}(t) = r^{tat}$ and $mapA^{tat}(t, c) = a^{tat}$ and . Given this, the two entries a and a^{tat} are turned into one. Additionally, a direct association is created from r to r^{tat}.

Figure 5.4 shows the index merge of the sources s_1 and s_3, represented as a graph (information about the values are omitted for the sake of clarity). The index is a bipartite graph with directed edges connecting relations and attributes. The



TAT Index	
Relation	**Attribute**
S_1	genre, title
S_2	award, title, year
S_3	title, year
S_4	pub, title, year
S_5	genre, title, year
S_6	award, title

Figure 5.3: TAt index populated with the contents of sources S_1 to S_6

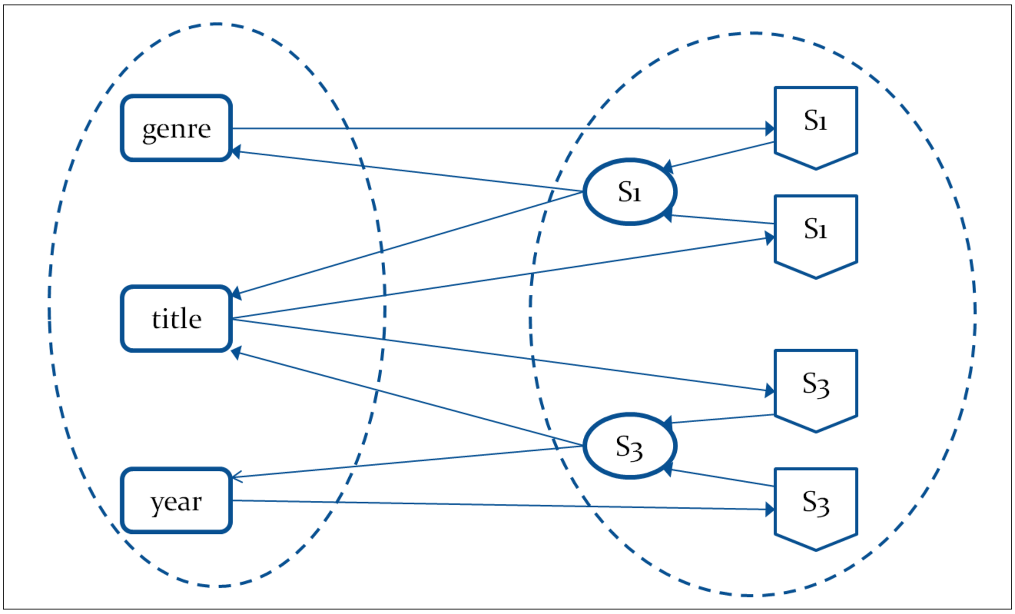


Figure 5.4: Merging the ATa/ATaVa with the TAt. The index is a bipartite graph connecting relations and attributes

advantage of this arrangement is that the lookups can start either at the attribute level or the relation level. Dashed nodes indicate the entry points of the merged index.

5.3 Query Rewriting

In what follows, we present a series of examples that illustrate the proposed rewriting algorithms. Given a query, we show how to access the indexes in order to find the sources that are able to provide meaningful answers.

To start, consider Example 1. The query is composed by a single subgoal with two variables.

Example 1: Find the title and year of movies.

Q_1**:h(title, year) :- movie(title, year)**

To answer this query, we need to find the possible rewritings. To our goal, a rewriting of a conjunctive query q

$$h(X) : -g_1(X_1), ..., g_n(X_n), f_1, ..., f_m$$

is another conjunctive query w

$$h^i(X^i) : -g^i_1(X^i_1), ..., g^i_n(X^i_n), f^i_1, ..., f^i_m$$

whose subgoals refer to wrapped relations.

Moreover, every subgoal of q must be mapped to a subgoal of w. In other words, every subgoal of the query needs to be covered by a relation found on the Web. We call this mapping Table Match. A table match indicates that the variables of the subgoal are covered by the attributes of the matched relation.

For queries with a single relation subgoal, such as in Example 1, there will be a rewriting for each table match found for the subgoal. Rewriting algorithms traverse the indexes in order to find the wrapped relations that are able to form table matches. We call this process Table Search.

Additionally, we make a distinction between complete and partial table matches. In a complete table match, every variable is covered. In a partial table match, some of the variables are left uncovered. First, we explain how to support rewritings that use complete table matches.

5.3.1 Table Search Algorithm: Table Scan

A naïve table search approach to find complete table matches is described in Algorithm 1. The Table Scan strategy performs a scan over all indexed relations, and creates table matches for every relation that covers all variables of the subgoal. TAt lookups indicate whether a query variable is indexed as an attribute of the given relation. Attributes that cover the subgoal variables are added into the corresponding table match.

In the example given, four table matches are created, for sources S_2, S_3, S_4 and S_5. Each of them forms a rewriting, as presented below.

Rewriting 1:	h(title, year)	:-	S_2(title, year, _)
Rewriting 2:	h(title, year)	:-	S_3(title, year)
Rewriting 3:	h(title, year)	:-	S_4(title, _, year)
Rewriting 4:	h(title, year)	:-	S_5(title, _, year)

For each rewriting, the subgoals of the original query are replaced by their respective matched relations, and the query variables are replaced by the matched attributes. Also, unmatched attributes of the wrapped relations become local variables (*award*, *pub* and *genre*). The underscore sign indicates this fact.

Observe that rewriting 4 is a false positive, since the answers reveal information about books instead of movies. This is a natural consequence of the lack of predefined mappings between a global schema and the sources. Index lookups can lead to false positives (in the case of homonyms) or false negatives (in the case of synonyms).

Although it is conceivable that a user may want to integrate information about books and movies, in some cases, these rewritings are not desirable. Tuning the

```
tableScan(Subgoal g)
//list of table matches
$TM \leftarrow \{\}$ //Initially empty
for every $r$ of $R^{tat}$ do {//iterating through relations using TAt}
    $findMatch(g, r, TM)$
end for
findMatch(Subgoal g, IndexedRelation r, TableMatches TM)
$tm \leftarrow \{\}$ //new table match
for every variable $x$ of $g$ do
    if $(x \in attr(r))$ then {//performing a TAt lookup}
        add into $tm$ the respective attribute that covers $x$
    else
        Return with no match between $g$ and $r$
    end if
end for
add $tm$ to $TM$
```

Algorithm 1: Table Scan Algorithm - finds wrapped relations that cover all variables of a relation subgoal

rewriting component to better match a user's preference is a problem we plan to study in future work.

5.3.2 Incomplete Rewritings

The previous algorithm is able to find table matches only if all variables of the subgoal are covered. When the subgoal contains many variables, the number of table matches tends to decrease, if we consider that the number of variables of a query is inversely proportional to the number of relations that are likely to provide a full coverage.

We address this problem by allowing incomplete rewritings. By incomplete we mean rewritings that do not cover all variables of the queries. In order to support this, the variables are classified as required or optional. Required variables always need to be covered whereas optional variables may remain uncovered. Given a subgoal g, the functions $req(g)$ and $opt(g)$ returns the required and optional variables of g, respectively, where $g = req(g) \cup opt(g)$.

The existence of optional variables demands the computation of partial table matches, where only the required variables need to be matched. Rewritings that are formed based on partial table matches are called incomplete.

To define which variables are required, one possibility is to let the user make this decision. In this case, the query language would have to be extended to support this feature. Instead, we propose an approach that defines required and optional variables based on information already expressed in the query.

A variable is considered required if:

- it is a selection variable, or
- it is a shared variable, or
- other than shared variables, it is the only distinguished variable of a subgoal,

> where the subgoal is not associative. An associative subgoal is one whose shared variables join any other two subgoal.

Selection variables are required because they restrict the query to a specific set of answers, those that are more relevant to the user. If a selection variable is left uncovered, that filter predicate can not be applied over the answers, and consequently the results can not reflect exactly what the user was expecting.

Shared variables are required because they are needed to perform the join conditions stated in the query. Queries with joins are discussed later.

The third type of required variable consider distinguished variables - those that are delivered to the user as results. Distinguished variables are required in special circumstances, when it is possible to realize that the subgoal is useless if that particular variable is not covered.

Figure 5.5 shows some examples of queries and their required variables. Observe that all selection and shared variables are required. Also, observe that not all distinguished variables are required. For instance, the first and second queries are very much alike, but the distinguished variable a is required only in the first, where it is the only distinguished variable of the subgoal.

In the fifth query, neither of the distinguished variables c or d are required. When the subgoal contains more than one distinguished variable, we take a less restrictive approach and set them as optional. Based on the same rule, the third query contains only optional variables.

Another example is given in the last query. In this case, variable d is the only distinguished variable of subgoal2, and yet, it is not considered required. The reason is that the variable d belongs to an associative subgoal. When associative goals are present, we assume that their main goal is to act as a bridge between other subgoals, and therefore, distinguished variables that come from these kind of subgoal are not considered required.

The rules that define required variables were created based on an intuition about the importance of each variable in the query processing and how they would affect the quality of the answers. Generally speaking, the approach can be extended to include other rules or even to suppress some. In fact, one possibility would be to consider entire subgoals as required or optional. Discussions about the impact of such changes are left for future work.

To demonstrate the creation of incomplete rewritings, consider Example 2.

Example 2: Find the title and genre of movies released before the year 2009.

Q_2: **h(title, genre) :- movie(title, genre, year), year $<$ 2009**

This example shows a query that contains a selection variable (*year*), which becomes the only required variable of the query. Therefore, in order to answer Q_2, we need to find wrapped relations that cover at least the *year* variable. Algorithm 2 shows how this is accomplished. It is an extension of Algorithm 1 that changes the definition of the method $findMatch$.

Similarly to the previous case, four table matches are created, for sources S_2, S_3, S_4 and S_5. Each of them generates a rewriting, as presented below:

h(a):-subgoal1(a,b), b = 2
required variables: a,b

h(a,b):-subgoal1(a,b), b = 2
required variables: b

h(a,b):-subgoal1(a,b)
required variables: –

h(a,c):-subgoal1(a,b), subgoal2(b,c)
required variables: a,b,c

h(a,c,d):-subgoal1(a,b), subgoal2(b,c,d)
required variables: a,b

h(a,c,d):-subgoal1(a,b), subgoal2(b,c,d), c = 2
required variables: a,b,c

h(a,b,e):-subgoal1(a,b), subgoal2(b,c), subgoal3(c,e)
required variables: a,b,c,e

h(a,d,e):-subgoal1(a,b), subgoal2(b,c,d), subgoal3(c,e)
required variables: a,b,c,e

Figure 5.5: Examples of queries and their required variables

w_1 :	h(title, genre)	:-	S_2(title, year, _),	year < 2009
w_2 :	h(title, genre)	:-	S_3(title, year),	year < 2009
w_3 :	h(title, genre)	:-	S_4(title, _, year),	year < 2009
w_4 :	h(title, genre)	:-	S_5(title, genre, year),	year < 2009

Observe that the table match for S_5 is the only one that covers all variables of the query. The other three fail to return the genre. Nevertheless, this incompleteness does not invalidate the rewritings, since the missing information is not needed for selecting the tuples (no filter or join condition depends on this variable).

5.3.3 Ranking

Ranking strategies are recommended when a query contains many possible rewritings. In our approach, the ranking is given by a similarity function based on the number of covered variables: the larger the number of covered variables, the more relevant is the wrapped relation.

The similarity is measured between a subgoal $g = \{x_1, ..., x_n\}$ and a wrapped relation $t = \{c_1, ..., c_j\}$, using Equation 5.1. Let $|g|$ be the number of variables of g. Additionally, $cover(t, x)$ returns 1 if the relation t covers variable x, and 0 otherwise.

$$sim(g,t) = \frac{\sum_{i=1}^{n} cover(t, x_i) * sel(x_i)}{|g|} \tag{5.1}$$

```
findMatch(Subgoal g, IndexedRelation r, TableMatches TM)
$ptm \leftarrow \{\}$
for every variable $x$ of $g$ do
    if $(x \in attr(r))$ then {//performing a TAt lookup}
        add into $ptm$ the respective attribute that covers $x$
    else if $x \in req(g)$ then
        Return with no match between $g$ and $r$
    end if
end for
add $ptm$ to $TM$
```

Algorithm 2: Adapted Table Scan Algorithm - finds wrapped relations that cover all required variables of a relation subgoal

The equation returns a normalized score between zero and one. Observe that the size of the subgoal is used as the normalization factor. Another possibility is to use the size of the relation instead. In this case, relations with a large number of attributes would get a lower score.

The function $sel(x)$ computes the selectivity of an attribute. This selectivity - given by Equation 5.2 - measures the inverse proportion of how often each attribute appears in the corpus of indexed relations. When relations cover the same amount of variables, this information helps to determine which relations should get a better score.

$$sel(a) = 1 - \frac{|rel(a)|}{|R^{tat}|} \tag{5.2}$$

The selectivity of the indexed attributes is presented in Table 5.1. Observe that the selectivity of *title* is zero, since it appears in all indexed relations. According to Equation 5.1, any attribute that appear in less relations will be considered more relevant than *title*. Another possibility is to do the opposite, and consider that the attributes that appear in larger number of relations are more relevant to the user.

Attribute	Selectivity Factor
Title	0
Year	0.33
Genre	0.66
Award	0.66
Pub	0.84

Table 5.1: Selectivity of the indexed attributes

To illustrate the ranking, take Example 3 presented next:

Example 3: Find the title, genre, year and award of movies.

Q_3**: h(title, genre, year, award) :- movie(title, genre, year, award)**

Since no required variables are provided, six rewritings are produced in this case, since all six indexed relations cover at least one variable of the query. Table 5.2 shows how these rewritings are ranked according to Equation 5.1.

List of rewritings				Score
1	h(title,genre,year,award)	:-	S_5(title, genre, year)	0.25
2	h(title,genre,year,award)	:-	S_2(title, year, award)	0.25
3	h(title,genre,year,award)	:-	S_1(title, genre)	0.16
4	h(title,genre,year,award)	:-	S_6(title, award)	0.16
5	h(title,genre,year,award)	:-	S_3(title, year)	0.08
6	h(title,genre,year,award)	:-	S_4(title, pub, year)	0.08

Table 5.2: Ranking based on the rewritings of query Q_3

It is important to remark that relations that cover the exact same variables (or whose participant attributes have the same selectivity) will still receive the same ranking. We understand that additional ranking criteria is needed. Some natural possibilities include the number of records of the matched relations and the relevance of the Web source (e.g., the pagerank) from where the relation was extracted. Moreover, indexed attributes could match query variables by similarity. In this case, it is possible to extend the function *cover*(t, x) to take the similarity into account.

5.3.4 Table Search Algorithm: Pivot Table Scan

This variation of the Table Scan Strategy narrows the table search by scanning only a specific list of relations. This list is formed by all relations that cover one determined variable of the query, which we name pivot. Algorithm 3 shows how to adapt the method *tableScan* in order to achieve this goal.

```
tableScan(Subgoal g)
 1: //list of table matches
 2: $TM \leftarrow \{\}$ //Initially empty
 3: $pivot \leftarrow NULL$ //Initially empty
 4: for every variable $x$ of $req(g)$ do
 5:    if $(x \in A)$ then {//performing a ATa lookup}
 6:       if $(pivot = NULL$ or $sel(pivot) > sel(x)$ then
 7:          $pivot \leftarrow x$
 8:       end if
 9:    end if
10: end for
11: for every relation $r$ of $rel(pivot)$ do {//iterating through relations using ATa}
12:    $findMatch(g, r, TM)$
13: end for
```

Algorithm 3: Pivot Table Scan - reduces the number of relations that need to be processed

As the algorithm shows, ATa is used to find relations that cover the pivot variable(line 11). Observe that relations that do not cover the pivot are not processed in the next step, and will not form rewritings. This is not a problem because the pivot is always a required variable. Therefore, only relations that cover this variable can be part of rewritings. Moreover, if the subgoal has more than one required variable, the pivot is the one whose selectivity is higher, which reduces the number of relations that needs to be processed.

To illustrate the use of a pivot, consider Example 4.

Example 4: Find the title of movies released before the year 2009.

Q_4: **h(title) :- movie(title, year), year $<$ 2009**

In Example 4, both `title` and `year` are required, the former for being the only distinguished variable of the subgoal and the latter for being a selection variable. In this case, `year` is chosen as the pivot because its source selectivity is higher.

This approach is limited to the existence of required variables in the subgoal, since only them can become pivots. If all variables are optional, taking one of them as the pivot would prevent the algorithm from finding any rewriting where that specific variable is not covered. Likewise, if a required variable exists, taking a optional variable as the pivot would prevent the algorithm from finding rewritings where that required variable is covered but the optional variable is not.

5.3.5 Table Search Algorithm: Table Intersection

Basically, the Table Intersection strategy uses ATa to find the lists of relations that cover each variable of the subgoal. Then, these lists are combined to find the rewritings.

Algorithm 4 describes how this approach works. For each required variable, a list of table matches is computed. These lists are then incrementally combined by intersecting table matches that refer to the same indexed relation. Algorithm 5 explains how the intersection works.

After all intersections are performed, the final list contains table matches that cover all required variables. The table matches are also updated to include the optional variables. If there are no required variables, the list of table matches contains all relations that cover at least one optional variable.

Algorithm 5 presents the auxiliary method that enables the pair-wise intersection of lists in general. The method is also used in Algorithms 6 and 8, as detailed later. We do no go into details about how the intersection operation itself is performed, as many possibilities emerge (Merge-join, Hash join, etc...). However, we do make a remark that some intersection algorithms can explore the attribute selectivity in order to improve the search. For example, in merge joins, performance is better when the smaller list is traversed in the outer loop.

Additionally, in pair-wise intersection, a more general improvement is achieved by intersecting smaller lists first. For instance, consider Figure 5.6. It shows the intersection process for a query where *title*, *genre* and *year* are required variables. Observe that, if the relations that contain *genre* and *year* are intersected first, the intermediary result is smaller, which reduces the time required in the next step.

5.3.6 Rewriting Check for Queries without Joins

The presence of selection subgoals in a query restricts the universe of interest of the user. Even if a wrapped relation covers all variables of a subgoal, it may not satisfy the provided filter predicates. For example, consider the query below.

Example 5: Find romance movies released before 1950.

Q_5 **h(title):- movie(title,genre,year), year $<$ 1950, genre = 'romance'**

```
tableScan(Subgoal g)
$TM \leftarrow \{\}$ //list of table matches
for every variable $x$ of $req(g)$ do
   if $(x \in A)$ then {//performing a ATa lookup}
      $TM_{aux} \leftarrow \{\}$
      for every relation $r$ of $rel(x)$ do {//iterating through relations using ATa}
         create $tm_r$
         add into $tm_r$ the respective attribute that covers $x$
         add $tm_r$ into $TM_{aux}$
      end for
      if (not $intersect(TM, TM_{aux})$ then
         //Error - no relation covers all required variables.
      end if
   else
      //Error - required variable not found.
   end if
end for
for every variable $x$ of $opt(g)$ do
   if $(x \in A)$ then
      for every relation $r$ of $rel(x)$ do
         if $(|req(S)| == 0)$ then
            create $\omega_r$
            add $tm_r$ into $TM$
         else
            get $tm_r$ from $TM$
         end if
         add into $tm_r$ the respective attribute that covers $x$
      end for
   end if
end for
```

Algorithm 4: Table Intersection Algorithm - based on the intersection of the relations lists that cover each required variable

The only relation that covers all required variables of query Q_5 is S_5. However, in order for this relation to fully satisfy the selection predicates, both predicates should be satisfied for the same tuples. In this case, there is no overlap between the tuples that satisfy predicate `year < 1950` (tuple two) and the tuples that satisfy predicate `genre = Romance` (tuple one). Consequently, this rewriting produces no answers, and is irrelevant to the user. In this work, we call these Empty Rewritings.

In order to prevent empty rewritings from being produced, we apply a rewriting check to each computed rewriting, as described in Algorithm 6. Assume that the algorithm uses a list $validTuples_r$ for every relation r that participate of the rewriting. This list contains all tuples that satisfy the predicates that apply to r.

Given a selection predicate over a relation r, an auxiliary list $filterList$ is populated with the tuples of r that satisfy this predicate. This is repeated to every selection predicate of the query. Auxiliary lists that refer to the same relation are

```
intersect(ref List list1, List list2)
if $list1 == \{\}$ then
    $list1 \leftarrow list2$
else
    $list1 \leftarrow list1 \cap list2$
end if
if $list1 == \{\}$ then
    return false//the intersection is empty.
else
    return true
end if
```

Algorithm 5: Auxiliary intersection method used by Algorithms 4, 6, 8

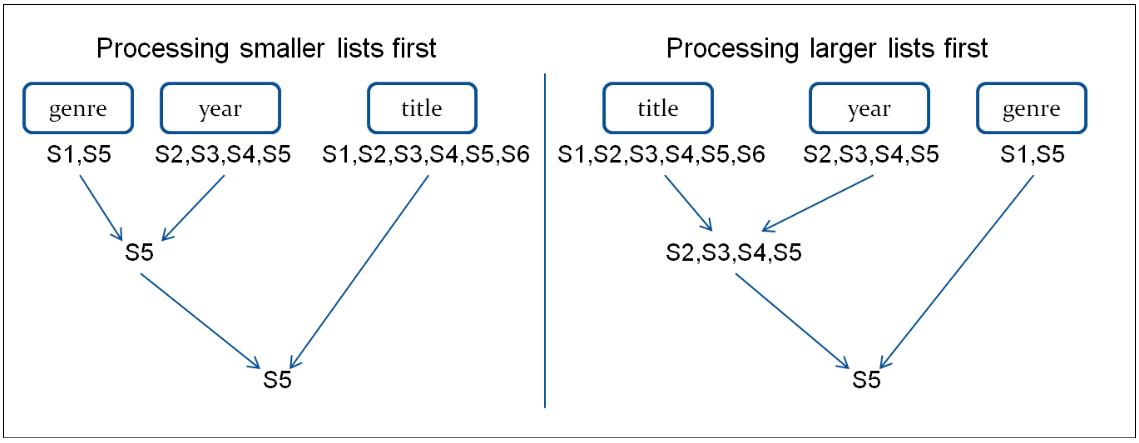


Figure 5.6: Intersection of lists: using smaller lists in one step reduces the amount of work in the next step

intersected. The result of the intersection is incrementally computed and stored in $validTuples_r$. If this list becomes empty, it means that the relation does not contain tuples that satisfy all of the respective selection predicates.

The *checkFilter* function builds a list of tuples that satisfy one specific selection subgoal, where tuples are uniquely identified by the function $ide(a, r, v)$. This pseudo-code can be improved with more efficient implementations. For instance, we can assume that $val(a, r)$ retrieves ordered values, and then perform the comparisons in logarithmic time. Besides, the data type may be used to indicate the semantic of the comparison predicates. For example, since `year` is a numeric value in the index, the values could be compared as numbers, which could return different results as it would if the values were compared as dates or plain strings. The semantic of the data type comparisons is an implementation detail and as such is not discussed in this work.

5.3.7 Early Check

In some situations, it is possible to identify relations that cannot satisfy the selection subgoals of the query before the rewritings are produced. For instance, consider Example 6.

```
checkRewriting(Rewriting w)
//check the filters
for every relation subgoal $g$ of $w$ do
    checkFilters($w$,$g$)
end for
checkFilters(Rewriting w, Subgoal g)
$r \leftarrow$ indexed relation that was matched to $g$
for every matched variable $x$ of $g$ do
    $a \leftarrow$ indexed attribute that was matched to $x$
    for every filter $f$ of $w$ where $f.\alpha = x$ do
        filterList ← checkFilter(f,a,r)
        if (not $intersect(validTuples_r, filterList)$) then
            //Error - the relation does not cover all of its respective selection variables.
        end if
    end for
end for
checkFilter(Filter f, IndexedAttribute a, IndexedRelation r)
$filterList \leftarrow \{\}$ //list of tuples that satisfy the filter $f$
for each value $v$ of $val(a, r)$ do {//iterating through values using ATaVa}
    if ($v$ satisfies filter $f$) then
        add $ide(a, r, v)$ into $filterList$
    end if
end for
return filterList
```

Algorithm 6: Rewriting Check - Identifies rewritings that produce no answers

Example 6: Find the title and genre of movies released in the 90's.

Q_6 **h(title,genre):- movie(title,genre,year), year >= 1990, year < 2000**

Four of the indexed relations cover the required variable year. However, only two of them contains movies released in the nineties(S_4,S_5).

Relations that fail to satisfy the *year* predicates are identified before the rewritings are produced, in a process called selection check. This early check is able to identify relations that cannot provide any answer, and ignore them when producing the rewritings.

Pivot Table Scan and Table Intersection can perform this check when they search for indexed relations that contain a specific attribute. Algorithm 7 shows how to find the relevant relations. In order to enable this check, Pivot Table Scan and Table Intersection should replace the calls for $rel(a)$, that retrieves all relations that contain a, to calls for $relationLookup(a)$.

It is important to remark that two checks are available: the early check and the rewriting check. The former only applies when the rewriting contains selection subgoals. Conversely, rewriting checks are useful in two different situations:

when joins are present In this case, we need to check if at least one tuple from the relation subgoals satisfies the join conditions. This is not an early check

```
relationLookup(IndexedAttribute a)
relations ← {}
for every relation $r$ of $rel(a)$ do {//iterating through relations using ATa}
    $allFiltersList$ ← {} //list of values that satisfy all filters of $a$
    for every filter $f$ of the query where $f.\alpha = a$ do
        filterList = ← checkFilter(f,a,r)
        checkIntersection($allFiltersList, filterList$)
    end for
    if (|allFiltersList| > 0) then
        add $r$ into $relations$
    end if
end for
return $relations$
```

Algorithm 7: Relation lookup that performs the selection check

because the join information exists only after the rewritings are produced. More about this check is presented in Section 5.3.8.1.

when more than one variable of a relation subgoal is a selection variable In other words, more than one filter apply over the same set of tuples. For instance, the query of Example 5 contains two selection variables of the same subgoal (*year* and *genre*).

If none of these conditions hold, the early check suffices. Otherwise, only the rewriting check is able to determine if a rewriting is empty or not. Despite of that, when selection subgoals are provided, the early check can always be used in conjunction with the rewriting check, as it helps pruning irrelevant relations, and consequently, reducing the size of the problem.

5.3.8 Rewriting Queries with Joins

The queries presented so far are composed by a single relation subgoal. In this section we present an approach that supports queries with multiple relation subgoals.

This is accomplished by the M-Bucket algorithm. The name of the algorithm is as a reference to the use of buckets. The concept of bucket was already employed in past work, for the problem of generating rewritings from LAV mappings (The Bucket Algorithm, described in (LEVY; RAJARAMAN; ORDILLE, 1996), was proposed as a rewriting algorithm for the Information Manifold system).

The steps of the M-Bucket algorithm are described below:

1. For each relation subgoal g in the query, create a bucket.

2. Add an entry in the bucket for every wrapped relation from where tuples of g can be possibly retrieved, i.e, relations that are matched with g.

3. Remove relations that only cover shared variables, when the subgoal is not associative.

4. Rank the entries of each bucket.

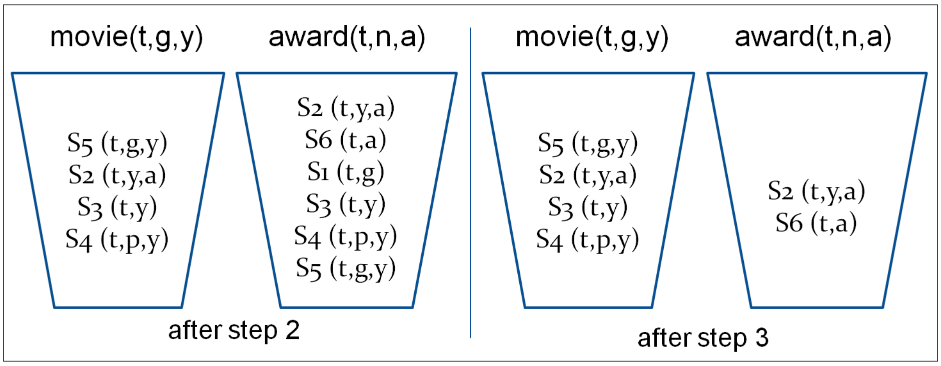


Figure 5.7: Buckets populated with relations that match subgoals *movie* and *award*

5. Generate rewritings as conjunctive queries by combining one entry from each bucket.

6. Simplify the rewritings, eliminating irrelevant subgoals

7. Remove rewritings that produce no answers (rewriting check)

8. Bind the selection predicates variables of the user query to their respective variables of the rewritings.

To illustrate the algorithm, consider Example 7, which contains more than one relation subgoal.

Example 7: Find the title, genre, nominations and awards of movies released in the 20th century.

Q_7 **h(title,genre,nom.,award) :- movie(title,genre,year), award(title,nom.,award), year>=1900, year<=2000**

Given query Q_7, two buckets are created, one for *movie* and one for *award*. Then, the buckets are populated with relevant wrapped relations. This information can be retrieved from any of the algorithms described earlier (Table Scan, Pivot Table Scan, Table Intersection).

Figure 5.7 shows, in the left part, the buckets populated with the matched relations. The subgoal for *movie* contains relations that cover the required variables *title* and *year*, and the subgoal for *award* contains relations that cover the required variable *title*.

In the right, Figure 5.7 shows the buckets after the third step of the algorithm. The relations removed from the *award* bucket are unable to contribute with answers, since they only cover the shared variable *title*.

In the next step, the entries of each bucket are ranked. The rank is obtained based on the number of attributes covered and the attribute selectivity, as explained in Section 5.3.3. To simplify the explanation, the entries of Figure 5.7 are already ordered accordingly.

In the fifth step, each entry of a bucket is combined with each entry of the other bucket. The result is a cartesian product that contains every possible way to rewrite the original query using the relations available. Figure 5.8 shows the rewritings generated based on the final contents of the buckets. The underscore symbol indicate local existential variables that are not important to the query.

Q_7 h(title, genre,		
nom., award):-	movie(title, genre, year),	award(title, nom., award)
Rewriting 1:	S_5(title, genre, year),	S_2(title, _, award)
Rewriting 2:	S_5(title, genre, year),	S_6(title, award)
Rewriting 3:	S_2(title, year, _),	S_2(title, _, award)
~~Rewriting 4~~:	S_2(title, year, _),	S_6(title, award)
~~Rewriting 5~~:	S_3(title, year),	S_2(title, _, award)
~~Rewriting 6~~:	S_3(title, year),	S_6(title, award)
~~Rewriting 7~~:	S_4(title, _, year),	S_2(title, _, award)
Rewriting 8:	S_4(title, _, year),	S_6(title, award)

Figure 5.8: Rewritings of the user query asking for awarded movies (Q_7)

Observe that the ordering of the buckets performed during the forth step leads to ordered rewritings, where rewritings that cover more variables appear first. We remark that the ranking can be obtained in different ways, and even after the cartesian product is performed. For instance, it would be possible to use ranking criteria that refer to the rewriting as a whole. An interesting possibility is to set a higher ranking for rewritings that come from the same location(such as the same Web page or database).

After the rewritings are produced in step 5, the algorithm proceeds by performing a simplification (step 6) and a rewriting check (step 7). The simplification shows how to remove subgoals from a rewriting without loosing meaningful information in the answers. Section 5.3.8.2 shows how to apply this simplification over rewriting 3 of Figure 5.8. The rewriting check identifies rewritings that are not able to satisfy the query predicates (such as the ones marked with a stripe in Figure 5.8). Section 5.3.8.1 shows how this check is performed in the presence of join predicates.

Finally, the last step simply adds the selection subgoals into the rewritings, and replaces the query variables used in the selection with their corresponding matched attributes.

We finish this section with Example 8, which shows a case where two relation subgoals refer to the same real world entity (*worker*).

Example 8: Find the name and the place of birth of the director and the lead actor of the movie Titanic.

Q_8 **h(director,birthPlace1,**
actor,birthPlace2) :- **movie(title,director,actor),**
workers(director,birthPlace1),
workers(actor, birthPlace2),
title = "Titanic"

In this case, the M-Bucket algorithm creates one bucket for each subgoal, regardless of whether it is repeated or not. Consequently, the rewritings may bring tuples of *worker* from different sources, for each instantiation of the same entity. At first, it seems that data from a given entity should always come from the same source. However, rewritings that bring data from different sources may contribute with different (and perhaps more) results.

To illustrate, consider that two wrapped relations match *worker*, and that one of them contains data about directors while the other contains data about actors. If the two *worker* subgoals match the same wrapped source, the rewriting produced would be empty. If the *worker* subgoals are allowed to match different wrapped relations, the m-bucket algorithm is able to find a non-empty rewriting.

5.3.8.1 Rewriting Check for Queries with Joins

A rewriting check determines whether a rewriting is able to produce answers that satisfy the query predicates. We have already presented a limited version of the check in Algorithm 6, where only selection predicates are processed. In this section we show how to extend the algorithm in order to check if a rewriting satisfies both selection and join predicates at the same time.

Algorithm 8 shows how the joins are checked. Since the selection predicate check was already discussed, it is omitted from the specification (line 2). Just as the previous version, a list of valid tuples exist for every relation r that participate of the rewriting ($validTuples_r$). The function *intersect* updates this list during the join check(Line 20), as it is updated during the selection check presented in Algorithm 6. If it becomes empty at any time, it means that no tuple of r satisfies all of the predicates (filters and joins) at the same time. For consistency sake, the lists of the selection predicates are intersected first. This measure prevents dangling tuples, from instances that were joined before the selection predicate had the chance to remove them.

ATaVa lookups ($val(a, r)$) provide the information used to perform the join check. Observe that the values of the attributes that participate of the join are compared by the equality operator. Another approach would be to compare the values by similarity. This similarity operator could also be apply during the selection check, for example, by using fuzzy quantifiers, as described in (KACPRZYK; ZIóLKOWSKI, 1986).

5.3.8.2 Query Simplification

Rewritings produced by the M-Bucket algorithm may contain duplicated subgoals, which refer to the same wrapped relation. In some cases, such rewritings can be simplified by removing the repeated occurrences of the subgoal, or, in other words, by folding the subgoals into one.

Query folding can be applied whenever the duplicated subgoals share variables, and the shared variables appear in the same position in their respective subgoals (i.e., the duplicated relations are self-joined by the same attribute). Take for instance rewriting 3:

h(title,genre,nomination,award) :- S_2(title, year, _), S_2(title, _, award),
year>=1900, year<=2000

In this case, the wrapped relation S_2 is self-joined by *title*. To fold the occurrences of S_2, variables that appear in the same position are replaced by a unified variable in the folded subgoal. Besides, occurrences of the old variables throughout the rewriting are also replaced by their respective unified variable.

```
checkRewriting(Rewriting w)
//check the filters
//...
//check the joins
for every shared variable $x$ of $w$ do
   for every pair of subgoals $g_i$ and $g_j$ of $w$ that contain $x$ do
      $r_i \leftarrow$ indexed relation mapped to $g_i$
      $r_j \leftarrow$ indexed relation mapped to $g_j$
      $a_i \leftarrow$ indexed attribute mapped to $g_i.x$
      $a_j \leftarrow$ indexed attribute mapped to $g_j.x$
      $auxList_i \leftarrow \{\}$ //list of values of $r_i$ that satisfy the join
      $auxList_j \leftarrow \{\}$ //list of values of $r_j$ that satisfy the join
      for each value $v_a$ of $val(a_i, r_i)$ do {//iterating through values using ATaVa}
         for each value $v_b$ of $val(a_j, r_j)$ do
            if ($v_a == v_b$) then
               add $ide(a_i, r_i, v_a)$ into $auxList_i$
               add $ide(a_j, r_j, v_b)$ into $auxList_j$
            end if
         end for
      end for
      if (not intersect($validTuples_{ri}$, $auxList_i$)) or (not intersect($validTuples_{rj}$,
      $auxList_j$)) then
         //Error - the rewriting does not cover all predicates of the query.
      end if
   end for
end for
```

Algorithm 8: Rewriting check for selection and join predicates.

After the folding, the new representation of rewriting 3 is

h($title_u$,genre,nomination,$award_u$) :- S_2($title_u$, $year_u$, $award_u$),
$year_u$>=1900, $year_u$<=2000

We call this process Query Simplification, not minimization. This distinction is important since the folding of the duplicated subgoals does not minimize the query. Query minimization techniques ensure that the folded query is equivalent to the unfolded one. In this scenario, we do not have sufficient information to determine equivalence: there is neither global schema nor detailed information about the sources (e.g., the functional dependencies among them).

Even though we cannot safely affirm that the same tuples are generated after the folding, we demonstrate that, in most cases, the simplification generates the expected tuples. This conclusion is based on a behavior observed in relational queries, which is described in the statement below.

Relational Join Statement: Whenever two relations are joined in a query, at least one of the join sides is a primary key.

This statement is related to the composition of the answer. If none of the join sides represents a primary key, the answers produced could have a many-many relationship between tuples from the two relations (tuples that do not satisfy the forth normal form). However, the users are normally interested in 1-1 compositions(where one tuple from one side relate to only one tuple from the other side) or 1-n compositions(where multiple tuples from one side relate to only one tuple from the other side).

Therefore, assuming that one of the sides of the join is always a primary key, we can intuitively deduce that, when a relation is self-joined by the same attribute, this attribute is the primary key. Consequently, we can ensure that the tuples that result from the join englobe all tuples that exist in the relation, and this means that the join is not necessary.

One good reason to apply the simplification is related to performance. The time required to process the rewriting is reduced, as less joins need to be computed. Another reason is related to the answer itself, and its possible incompatibility with the Relational Join Statement.

To explain, consider rewriting 3 (Figure 5.8). Depending on the extension of S_2, the folded version of the rewriting could bring different results than the unfolded one. The reason is related to the impossibility of affirming that *title* is a primary key in S_2. This is particularly true for unconstrained Web data, where attributes may present repeated values along the wrapped relation, even those that apparently serve as primary keys. If some tuples of S_2 contain the same values for *title*, the result after a self-join would not be in the forth normal form. The simplified version of the query does not present this problem.

Summing up, the reasoning behind our simplification is that relational queries normally bring answers that satisfy the forth normal form. However, even in seldom cases where the user is interested in a many-to-many result, it is possible to affirm that the folded version of the rewriting is at least contained in the unfolded version. That is, using the folded version, the user will get at least a subset of the desired answers.

Next example shows an interesting case of our simplification that deserves attention. Consider the query below.

Example 9: Find the title, director and actors of movies.

Q_9 **h(title,director,actor):- movie1(title,director), movie2(title,actor)**

Complementarily, suppose that the wrapped relation $movies(title, cast)$ exposes the title and cast of movies. A mapping approach based on a thesaurus could match *cast* with both *director* and *actor* variables, and the following rewriting could be produced:

h(title,director,actor) :- *movies*(title, director), *movies*(title, actor)

According to our simplification approach, the duplicated subgoals can be folded as one, as indicated below:

h($title_u$,$director_u$,$director_u$) :- $movies(title_u$, $director_u)$

After the variables are unified, the variable $director_u$ appears twice in the header.

In this case, it indicates that both *director* and *actor* information can be found under the same wrapped attribute (*cast*).

Currently, we count on a simple mapping function that matches query variables with indexed attributes of the same name, so Example 9 could not be solved as demonstrated. However, it is possible to extend this function to obtain better results. Possibilities include name similarity techniques and the canonization of the indexed attributes.

Another idea is to explore the Relational Join statement even further, and possibly discover other kinds of simplifications, or even discover cases where a rewriting can be invalidated. For instance, the joins of a rewriting can be analyzed in conjunction to find out if they all satisfy the statement at the same time.

5.3.9 Query Answering using the Indexes

Rewritings are composed by relation subgoals, which refer to sources available on the Web. During the query answering, the sources of a rewriting are accessed through wrappers. These wrappers extract information from the sources into virtual relations, which are used by the query processor component. In this section, we discuss the possibility of creating virtual relations without accessing the sources, using information stored in the indexes.

Virtual relations of these kind are built based on its respective subgoal and the contents of the index. The attributes of the virtual relation are the variables of the subgoal. The other attributes of the wrapped source that are not demanded by the subgoal are not necessary. Additionally, it is possible to define which tuples of the Web source should be added to the virtual relation. One idea is to consider only the tuples that satisfy the predicates of the query. Algorithm 8, responsible for detecting empty rewritings, shows how to identify these tuples(the list $validTuples_r$ contains the tuples of r that satisfy the respective predicates).

By selecting only the necessary attributes and tuples, these virtual relations assume a reduced size, if compared to the original wrapped source. As a consequence, the Query Processor takes less time the compute the final answer, since it works with less information.

Interestingly, virtual relations created based on the index enables query answering without ever accessing the real sources. As a direct benefit, this approach allows rewritings to be answered even when the original source is unavailable. Additionally, query processing is faster, since it works with reduced relations and because there is no network latency involved to fetch the data.

5.4 Chapter Remarks

This chapter demonstrates one of the proposed solutions to the querying of structured and heterogeneous sources on the Web. To support this idea, we describe indexes that are able to match query subgoals with wrapped relations, along with table search algorithms that access the indexes in order to find the rewritings.

Furthermore, the concept of optional variables is introduced as a way to find additional rewritings to a query. We show how to decide which variables are optional, and how this information can be used. Besides, a ranking strategy is provided to determine which of the rewritings are deemed more relevant.

We also discuss optimization techniques, such as the elimination of relations that are not able to satisfy the query predicates, and the query simplification used when the query contains multiple subgoals.

To finish this chapter, we highlight the importance of the data structures that compose the indexes, as they allow the creation of most of the algorithms and techniques described above. Extremely relevant is the usage of the values level, present in the AVaTa. Based on the information of this index, it is possible to perform powerful computations, such as the rewriting check and the query answering without ever having to access the real Web source.

6 QUERY REWRITING IN THE ABSENCE OF JOIN PREDICATES

The previous chapter shows components of the Eidos architecture that support queries with join predicates. The ability to express this kind of predicate represents a powerful tool, since the user is given the choice to explore Web Data in a flexible way. However, the expressiveness of this approach can also be seen as a burden. Users with no knowledge about the local schemas may experience difficulties when trying to build a query with joins.

In this chapter, we show how to remove the join complexity out of the query and add it into the rewriting components of the Eidos Architecture. Given join-less queries, it is up to the rewriter to discover if it is possible to find answers by joining information from different wrapped relations.

This chapter is organized as follows. Section 6.1 presents the query language that supports our idea. Section 6.2 presents extensions of the index that improve the rewriting process. Finally, Section 6.3 presents the algorithms that are able to find rewritings for the user queries.

6.1 Query Language.

Last chapter proposed the usage of SPJ queries as a mean to let the user define join conditions. Conversely, in this chapter we show a different approach that does not demand the declaration os joins. To support that, we define a query language that extends the traditional keyword-based interfaces with structural semantics. A query is defined as a list of attributes, followed by a (optional) list of selection predicates. An example is provided below:

Example 1: Find the title and year of movies released after 1990.

Q_1 → **title year [year > 1990]**

The variables involved in brackets are the (optional) selection predicates, whereas the remaining variables are projection predicates (variables that need to be returned as part of the answer).

Join predicates are not supported by the language. However, it is possible to unfold these simple queries into rewritings that span several wrapped relations. To exemplify, Table 6.1 shows queries and rewritings based on the relations of Figure 4.3.

The first case shows a rewriting composed by a single relation subgoal. This kind of rewriting is called single-relation rewriting. On the other hand, the second case shows a rewriting composed by more than one relation subgoal. These are called multi-relation rewritings. Given a query such as one of those, the goal of the rewriting algorithms is to find out whether it is possible to answer it using single-relation and/or multi-relation rewritings.

Query		Possible Rewriting
Q_x	title year	R_{x1}(title,year) :- S_3(title, year)
Q_y	title genre award	R_{y1}(title,genre,award) :- S_1(title,genre),S_6(title,award)

Table 6.1: Examples of queries unfolded into rewritings that can span several wrapped relations

6.2 The AVaTa Index

The indexes described in Chapter 5 (ATa, ATaVa, TAt) are useful for discovering which relations contains the variables of queries. This section presents a complementary index (AVaTa) that is important in the context of join-less queries.

The AVaTa index([A]ttribute-[Va]lue-[Ta]ble) extends the A Index with the quadruple (U, V, R, I).

The element $U = \{u_1, ..., u_n\}$ is the list of uniqueness factors. Furthermore, $unq(a) = u$ is the function that returns the uniqueness factor of a, where $u \in U$. This information indicates the probability of each attribute having unique values inside a relation.

The element $V = \{v_1, ..., v_n\}$ is the list of indexed values. Furthermore, $val(a) = V_x$, where $V_x \in V$ is the function that returns the list of values that are used as instances of a.

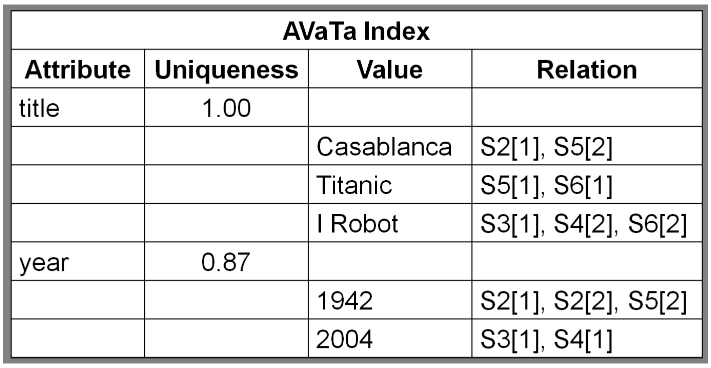
In addition, $R = \{r_1, ..., r_n\}$ is the list of indexed relations of each pair (a, v). The function $rel(a, v) = R_x$, where $R_x \in R$, is the function that returns the list of indexed relations related to the pair (a, v).

Finally, $I = \{i_1, ..., i_n\}$ contains the tuple identification of each triple (a, v, r). This identification is the vertical coordinate of a value v inside a indexed relation r, for attribute a. The function $ide(a, v, r) = i$, where $i \in I$, returns the respective tuple identification.

Tuples from a wrapped source are mapped to the AVaTa according to Definition 5.

Definition 5 *(AVaTa Mapping)*
The AVaTa Mapping extends the A Mapping to support the values/relations level of the AVaTa. Consider the wrapped attribute $c \in t$ and the domain value $d \in DOM(c)$ that belongs to the tuple τ of table t. Given this, let $mapV(c, d) = v$ be a function that maps the domain value d to the indexed value v, and $mapR(c, d, t) = r$ be the function that maps the wrapped relation t to the indexed relation r. Furthermore, considering the maps $mapA(c) = a$, $mapV(c, d) = v$ and $mapR(c, d, t) = r$, then $v \in val(a)$, $r \in rel(a, v)$ and $|rel(a, v)|$ must be greater than one. Finally, $ide(a, v, r)$ is given by the function $ide(\tau)$.

As explained in Section 5.2.3, the computation of $ide(\tau)$ is provided by the Wrapper module, and it depends on how the wrapper extracts information from the



AVaTa Index			
Attribute	Uniqueness	Value	Relation
title	1.00		
		Casablanca	S2[1], S5[2]
		Titanic	S5[1], S6[1]
		I Robot	S3[1], S4[2], S6[2]
year	0.87		
		1942	S2[1], S2[2], S5[2]
		2004	S3[1], S4[1]

Figure 6.1: AVaTa index populated with the contents of sources S_1 to S_6

underlying data model of the wrapped source. Observe that the same function is used in both ATaVa and AVaTa, so the tuple identification across the indexes is consistent.

Moreover, the uniqueness factor of an attribute ($unq(a)$) is given by Equation 6.1. The equation computes the result over all relations that contain a wrapped attribute mapped to the indexed attribute a. In the equation, $repeatedValueCount(t,c)$ returns the number of repeated (non empty) values that appear in attribute c of table t. For example, the repeated value count of attribute *year* in S_2 and S_3 is respectively 1 and 0. On the other hand, $totalValueCount(t,c)$ returns the total number of (non empty) values that appear in attribute c of table t. For example, the total value count of attribute *award* in S_2 and S_6 is respectively 1 and 2. The uniqueness factor is a normalized value between 0 and 1. The higher the value, the greater is the probability of the attribute having unique values inside a relation.

$$unq(a) = \sum_{k=1}^{n} \frac{totalValueCount(t_k,c) - repeatedValueCount(t_k,c)}{totalValueCount(t_k,c)} \tag{6.1}$$

where n is the number of wrapped relations that contain c and $mapA(c) = a$.

To exemplify the mapping, Figure 6.1 shows the AVaTa populated with the contents of sources from S_1 to S_6. Only two attributes appear: *title* and *year*. These are the attributes that appear in more than one relation with the same value, as described in the AVaTa Mapping definition ($|rel(a,v)|$ greater than one).

The value inside the square brackets identifies the tuple from which the specified value can be obtained, similarly to the ATaVa. The tuple identification helps in the process of checking rewritings that contain both join and filter predicates. Another useful information is the uniqueness factor. In the example given, it shows that *title* is more likely to be unique inside the relations than *year*. Our ranking strategy explores this factor in order to determine the relevance of rewritings. More about the check and the ranking is presented in Sections 6.3.5 and 6.3.6, respectively.

Attributes that appear in AVaTa are called join attributes. Join attributes act as associations between otherwise disconnected relations. This information is useful for discovering multi-relation rewritings, since it provides the location of relations that contain complementary data.

Observe that both ATaVa and AVaTa can be put together in the same index, sharing the first level. In this case, the second level would be composed by two different structures: indexed relations(the ATaVa component) and indexed values(the AVaTa component). Experiments in Chapter 7 show how this combination of indexes(and others) affect the memory consumption of the repository.

6.3 Query Rewriting

In what follows, we present a series of examples that illustrate the proposed rewriting algorithms for join-less queries. Special attention is dedicated to the process of finding multi-relation rewritings, where we start with a naïve solution and continue with more scalable approaches. It is also shown how the AVaTa is able to reduce the search space required to find relations that contain complementary data.

To start, consider Example 2. The query is composed by two projection predicates and no selection predicates.

Example 2: Find the title and year of movies.

Q_2: **title year**

Observe that join-less queries naturally map to SPJ queries that contain a single relation subgoal. For instance, a SPJ query that corresponds to Q_2 is shown below:

Q_2: **h(title,year) :- movie(title,year)**

Similarly, selection predicates are also easily mapped to selection subgoals in SPJ queries. For instance, consider Example 3 below, composed by one projection predicate and one selection predicate.

Example 3: Find movies released after 2003.

Q_3: **title [year $>$ 2003]**

One SPJ query that corresponds to Q_3 is shown below:

Q_3: **h(title) :- movie(title,year), year $>$ 2003**

Any query in our keyword language can be expressed as a conjunction of selection subgoals and one relation subgoal. By treating them as such, it is possible to find rewritings using the search algorithms detailed in Chapter 5, such as Table Scan or Pivot Table Scan.

The definition of required variables is similar to the one described in Chapter 5, for queries that may contain joins. However, since now queries are join-less, shared variables are not considered. In this case, a variable is required if:

- it is a selection variable, or
- it is the only distinguished variable of the subgoal.

In query Q_2, both variables are optional, which leads to six rewritings, one for each of the wrapped relations. On the other hand, Q_3 has two required variable (*title* and *year*), which leads to four rewritings, using sources S_2, S_3, S_4 and S_5. Observe that an early check helps pruning relations that do not satisfy the selection predicate of Q_3, which would remove relations S_2 and S_5. In addition, if the query contained selection predicates over distinct variables, such as [year $>$ 2003] [genre $=$ Drama], the rewriting check would also need to be performed.

6.3.1 Multi-Relation Rewritings

In this section we show how to find rewritings that join information from multiple wrapped relations.

To illustrate the approach, consider the wrapped relations of Figure 6.2. Observe that only the schema information is informed. For now, tuples are irrelevant to our discussion.

S_1(title,genre,year)
S_2(title,director,year)
S_3(title,genre)
S_4(title,genre)

Figure 6.2: Wrapped relations used to find multi-relation rewritings

Given these relations, suppose a user submits the query described below:

Example 4: Find the title, director, genre and year of movies.

Q_4: **title director genre year**

Observe that all wrapped relations are incomplete with respect to the query. The table search algorithms would find four partial table matches, since none of them is able to cover all query variables. However, it is possible to find combinations of these relations that provide a full coverage. Used in multi-relation rewritings, these combinations may produce more complete answers.

Join attributes are important in the generation of multi-relation rewritings. Recall that a join attribute is an attribute that belongs to the AVaTa index. They are the only attributes that appear in different wrapped relations with the same content. Multi-relation rewritings that do not contain any join attribute are irrelevant, since the join would produce empty result sets. Therefore, our purpose is to find multi-relation rewritings where the shared variable is a join attribute.

To create multi-relation rewritings, we first identify the partial table matches (relations that do not cover all variables of the query). Then, the incomplete relations are separated into join lists (JLs).

A JL is a list that clusters together relations that cover the same join attribute. Besides the join attribute, relations of a JL also need to cover at least another variable of the query.

For Example 4, suppose that just `title` is a join attribute. In this case, a single JL is produced, with all relations of Figure 6.2, since they all cover *title* and at least one of the remaining variables *genre*, *year* or *director*.

The general idea is to combine entries of the JL in order to find rewritings. By combining relations it is possible to increase the number of covered variables. For example, the combination of S_1 and S_2 can be transformed into a rewriting that provides a full coverage, as presented below:

$$h(title, director, genre, year) : -S_1(title, genre, year), S_2(title, director, \_)$$

The transformation of a combination into a rewriting occurs as follows. Each relation that compose the combination is turned into a relation subgoal (S_1 and S_2). Additionally, attributes that match variables of the query are transformed into variables of the rewriting (*title*,*director*,*genre*,*year*). The join attribute of the JL becomes the only join variable (*title*).

If two or more attributes (from different relations) cover the same variable of the query(other than the one used as the join variable), only one of them will cover the variable in the rewriting. That is the case for variable *year*, covered by attribute of S_1 and S_2. When this situation happens, we assume that the content of the attributes in the different relations is the same - for tuples that represent the same object - and arbitrarily select one of them. In the case above, the *year* attribute of S_1 was selected. The respective attribute in S_2 is treated as a local existential variable, represented by an underscore.

Other solutions are possible. For instance, if two or more attributes (from different relations) covers the same variable of the query, and the variable is distinguished, an idea would be to return all attributes instead of only one. This solution is interesting in cases where the content of the attributes differ, so the user receives all different contents as part of the answer.

Given a JL, many combinations form rewritings, using the transformation process described above. To our goal, not all of them are valid. In order to be valid, a rewriting needs to be both complete and minimal.

A minimal rewriting is one that produces different results if any of its relation subgoals are removed (non minimal rewritings are more time consuming since the number of relations that need to be joined is larger). As a general rule, a combination cannot form a valid rewriting if one or more relations could be removed and the rewriting would still cover all variables of the query.

A complete rewriting is one that covers all query variables. It is semantically equivalent to the AND operator of traditional keyword-based query languages. As a general rule, a combination cannot form a valid rewriting if its relations together do not cover all variables of the query. This restriction simplifies the process of finding the combinations. The support for incomplete multi-relation rewriting is left for future work.

Next sections show strategies that find valid rewritings. These strategies traverse the entries of a JL in different ways in order to find combination of relations that are both complete and minimal.

6.3.2 Combination Algorithm: Naïve

The Naïve algorithm finds valid rewritings using a simple combination strategy. First, we define the complement of each relation of the JL. The complement is a signature that is composed by the variables of the query that are not covered by the relation.

For instance, the JL created from query Q_4 is composed by S_1[director], S_2[genre], S_3[director,year], S_4[director,year], where the attributes inside the square brackets represent the complement of each relation. In the running example, relation S_1 does not cover *director*, S_2 does not cover *genre* and S_3 and S_4 do not cover *director* and *year*.

Algorithm 9 shows how the Naïve strategy works. Basically, the entries(relations) of the JL are processed sequentially, and every possible combination of entries is created.

```
$findValidPaths_1$(JoinList JL)
1: $C \leftarrow \{\}$//list of combinations
2: for every relation $r_i$ of JL do
3:     create combination $c$
4:     add relation $r_i$ to combination $c$
5:     add combination $c$ to the list $C$
6:     $findValidPaths_2$(JL,i,c,C)
7: end for
8: $checkCombinations(C)$
$findValidPaths_2$(JoinList JL, Number x, Combination c, List C)
1: for every relation $r_i$ of JL, where $i > x$ do
2:     create combination $c_{new}$
3:     $c_{new} \leftarrow c$
4:     add relation $r_i$ to combination $c_{new}$
5:     add combination $c_{new}$ to the list $C$
6:     $findValidPaths_2(JL, i, c_{new}, C)$
7: end for
```

Algorithm 9: The Naïve Algorithm - The valid rewriting check is performed after all possible combinations of relations is formed.

Table 6.2 presents the list with the 15 possible combinations created for Q_4. For lack of space, attributes names are abbreviated. The number of combinations for a list with n elements can be computed according to Equation 6.2. Observe that the number of combinations grows exponentially as the number of elements increase.

$$Total\ \#\ of\ combinations = \sum_{k=1}^{n} \frac{n!}{k!(n-k)!} \tag{6.2}$$

As Table 6.2 indicates, some of the combinations are not valid (not complete and minimal). It is possible to check if a combination $c = \{r_1..r_j\}$ is invalid if at least one of two conditions holds:

- the intersection of the complements of c is not empty (e.g., the combination S_1,S_3). This means that not all variables are covered, so the rewriting is incomplete.

- the intersection of the complements of c_x is the same as the intersection of the complements of c_y, where $c_x <> c_y$, for any $c_x \in c$ and $c_y \in c$ (e.g., the combination S_1,S_2,S_3). This means that the rewriting has redundant subgoals.

After all combinations are created, Algorithm 9 finds out which of the rewritings do not satisfy these conditions (Line 8). Algorithm 10 shows how to implement this

Combination	Missing Variables	redundant subgoals	Valid
$S_1(t,g,y)$	d		×
$S_1(t,g,y),S_2(t,d,y)$			√
$S_1(t,g,y),S_2(t,d,y),S_3(t,g)$		S_3	×
$S_1(t,g,y),S_2(t,d,y),S_3(t,g),S_4(t,g)$		S_3,S_4	×
$S_1(t,g,y),S_2(t,d,y),S_4(t,g)$		S_4	×
$S_1(t,g,y),S_3(t,g)$	d		×
$S_1(t,g,y),S_3(t,g),S_4(t,g)$	d		×
$S_1(t,g,y),S_4(t,g)$	d		×
$S_2(t,g,y)$	g		×
$S_2(t,d,y),S_3(t,g)$			√
$S_2(t,d,y),S_3(t,g),S_4(t,g)$		S_4	×
$S_2(t,d,y),S_4(t,g)$			√
$S_3(t,g)$	d,y		×
$S_3(t,g),S_4(t,g)$	d,y		×
$S_4(t,g)$	d,y		×

Table 6.2: Combinations found for a JL composed by the four relations of Figure 6.2. For lack of space, attributes names are abbreviated.

verification in a simplified manner. As indicated, the complements of a combination are intersected incrementally. The function *getComp*(r) returns the complement of a relation r. If the intersection becomes empty before the last relation is processed, it means that the second condition fails. If the intersection becomes empty after the last relation is processed, both conditions are satisfied.

After the verification, three combination are considered valid. Each of them is turned into a multi-relation rewriting, as presented below.

Rewriting 1:	h(title,director,genre,year)	:-	$S_1(title, genre, year)$, $S_2(title, director, \_)$
Rewriting 2:	h(title,director,genre,year)	:-	$S_2(title, director, year)$, $S_3(title, genre)$
Rewriting 3:	h(title,director,genre,year)	:-	$S_2(title, director, year)$, $S_4(title, genre)$

Since *title* is the join attribute, the relations are joined based on the content of this attribute. Observe that the Naïve strategy does not assure that any tuples will result from the join, so the rewriting check is necessary.

6.3.3 Combination Algorithm: Stream-Driven

The way in which the Naïve Algorithm accesses the entries of a JL can be visualized as a walk in a tree, where each of the combinations form a path from the root node to the leaf node. Figure 6.3 shows the "trace"of the processing, as the Algorithm steps from one relation to the other. For lack of space, the name of the attributes is represented in the abbreviated form.

This brute-force backtracking approach ends up accessing branches of this virtual tree that cannot form valid combinations (e.g., the path $/S_1/S_3/S_4$ is neither complete nor minimal). The Stream-Driven Algorithm reduces the number of node

```
checkCombinations(List C)
for every combination $c$ of C do
    $checkCombination(c)$
end for
checkCombination(Combination c)
$MV \leftarrow \{\}$ //missing variables
for every relation $r$ of $c$ do
    $MV \leftarrow MV \cap getComp(r)$
    if $MV == \{\}$ then
        if $r$ is the last relation of $c$ then
            return true //this combination is valid
        else
            return false //this combination is invalid
        end if
    end if
end for
```

Algorithm 10: The algorithm that performs the valid rewriting check

visits by removing some of the invalid paths, which prevents the subtree underneath the removed path from being processed.

To describe this idea, consider the path $p = \{r_1..r_j\}$, composed by relations of a JL. Given this, the relation r_{j+1} (where r_{j+1} is the relation that succeeds r_j in the JL) is not traversed if:

- the complements intersection of p is already empty. This means that the path traversed so far is already a complete rewriting, and it makes no sense to continue along this path. For example, the path $/S_1/S_2$.
- the complements intersection of r_1 to r_{j+1} is the same as the complements intersection of r_1 to r_j. In other words, the last relation of the path does not reduce the intersection. This means that one of the nodes along the path leads to a non-minimal rewriting. For example, the path $/S_1/S_3$.

Algorithm 11 shows how to detect invalid paths while the combination of relations is performed. Paths that are already valid or that contain unneeded relations are interrupted, and a new path is processed.

The benefit of this approach is depicted in Figure 6.3. The branches of the tree indicated with the dotted rectangle are not processed (they are not event formed), which reduces the search space without loosing any valid rewriting.

It is important to state that not all invalid paths are eliminated using this algorithm. Therefore, it is necessary to run the check described in Algorithm 10 after all rewritings are computed, as in the Naïve strategy. Moreover, a rewriting check is required to verify if the formed rewritings are empty. However, the number of rewritings that need to go through these checks tends to be much smaller if compared to the Naïve strategy.

6.3.4 Combination Algorithm: Template-Driven

In real cases, several wrapped relations will miss the same attributes of the query. In other words, their complements will be the same. Relations that contain the same

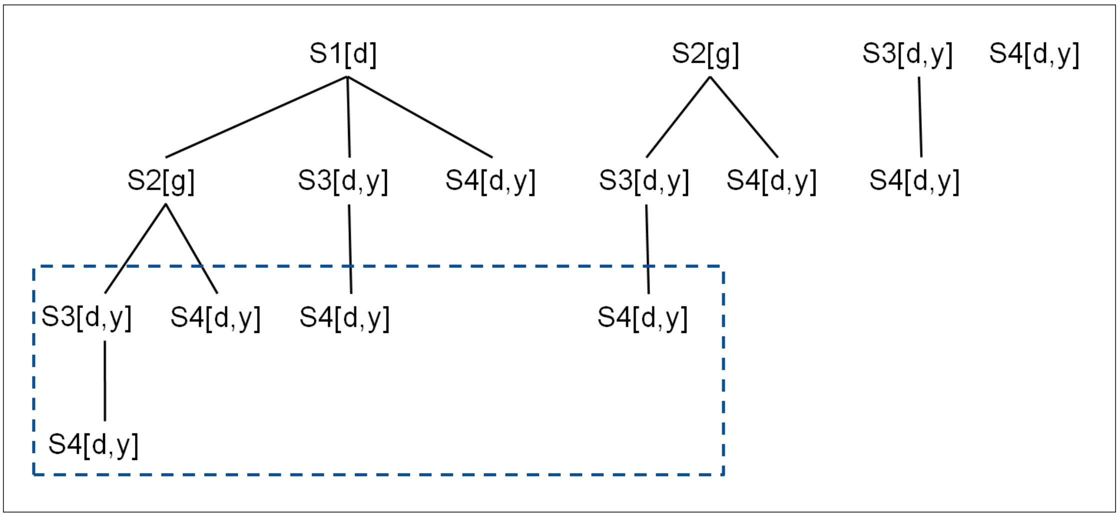


Figure 6.3: Trace of combinations left by the Naïve strategy. For lack of space, the name of the attributes is represented in the abbreviated form.

complement cannot be part of the same combination, since they would lead to non-minimal rewritings. The Template-Driven approach explores this fact by applying a summarization technique that groups these relations. The overall goal is to reduce the search space in order to find the rewritings.

Given a query, a pre-processing step groups relations with the same complement within a single entry in a JL. Consequently, there will be no more than one entry with the exact same complement. In this case, the size of the JL tends to be small, since it is no longer related to the number of relations, but to the number of possible different complements. The combination equation presented earlier(Equation 6.2) gives the number of possible different complements for a query with $n+1$ attributes. For instance, having a query with five attributes (one attribute that is fixed (the join attribute) and four other attributes), in the worst case there would be 24 entries (combination of one,two,three and four attributes).

To describe this approach, consider Example 5, and the data sources presented in Table 6.3. Also, assume that *title* is the only join attribute of the index.

Example 5: Find the title, director and genre of movies.

Q_5**: title director genre**

As Table 6.3 shows, despite the number of relations, there are only two different complements (with respect to query Q_5), which is in fact the maximum number of possible complements, provided that only two non-join attributes are part of the query. By grouping relations with the same complement, the number of entries in the JL reduces 80%.

After the JL with the grouped relations is generated, a template tree is build over it. This tree contains paths that lead to valid combinations - all other paths are ignored. The creation of this tree can be performed using Algorithm 11 as a basis. Line 4 (where the combination is validated) would have to be replace by a routine that reads the selected path and adds it to the template tree.

Figure 6.4 shows a comparison between full trees - that consider all possible combinations - and the optimized template trees, for Examples 4 and 5. For lack

```
$findValidPaths_1$(JoinList JL)
$C \leftarrow \{\}$//list of combinations
for every relation $r_i$ of JL do
    create combination $c$
    add $r_i$ to $c$
    $findValidPaths_2$(JL,i,c,C)
end for
$findValidPaths_2$(JoinList JL, Number x, Combination c, List C)
for every relation $r_i$ of JL, where $i > x$ do
    $c_{new} \leftarrow c \cap getComp(r_i)$
    if ($c_{new} == \{\}$) then
        add $c_{new}$ to $C$ //combination is valid
    else if ($c_{new} - c == \{\}$) then
        //combination does not reduce intersection(not minimal)
    else
        //only in this case the sub-paths are processed
        $findValidPaths_2(JL, i, c_{new}, C)$
    end if
end for
```

Algorithm 11: The Stream-Driven Algorithm - The valid rewriting check is performed while the combinations of relations is performed.

of space, the name of the attributes is represented in the abbreviated form. As illustrated, the trees in the left side contain paths that lead to invalid combinations, whereas all paths in the right side are valid.

Algorithm 12 shows how to form combinations using the template tree. The tree is traversed in a left-to-right, breath-first walk. The path from a root to each of the leafs leads to a group of valid combinations, where each combination comes from the cartesian product of the relations that are part of each node.

One positive aspect of template trees is that, although built in memory, they are considerably smaller then their respective Naïve/Streamed versions, as the examples show. Another benefit of the Template-Driven approach is that the summarization decreases the number of validity checks (check if complete and minimal) that need to be performed, and this improvement can be meaningful when dealing with a large number of relations. However, rewriting checks are still necessary.

6.3.5 Smart Join Lists

A join list clusters relations that share values for a given join attribute. However, it does not indicate which relations share the same value. When one of the combination algorithms(Naive, Stream or Template) process a JL, it is very likely that most of the combinations formed lead to empty rewritings that do not satisfy the join predicates. In this section, we show how to optimize the combination process by using smart join lists.

Instead of a single large JL for each join attribute, smart JLs are created for the same join attribute. As opposed to the large JL, smart JLs tend to be small. Besides, relations from a smart JL provide one guarantee: they all share the same value for the join attribute.

Relation Schema	Complement with respect to Q_5
S_1(title,genre)	director
S_2(title,director,year)	genre
S_3(title,genre)	director
S_4(title,genre,award)	director
S_5(title,genre,award)	director
S_6(title,director,award)	genre
S_7(title,director,award)	genre
S_8(title,director)	genre
S_9(title,genre,year)	director
S_{10}(title,director)	genre

Table 6.3: Data sources used to describe the Template-Driven strategy

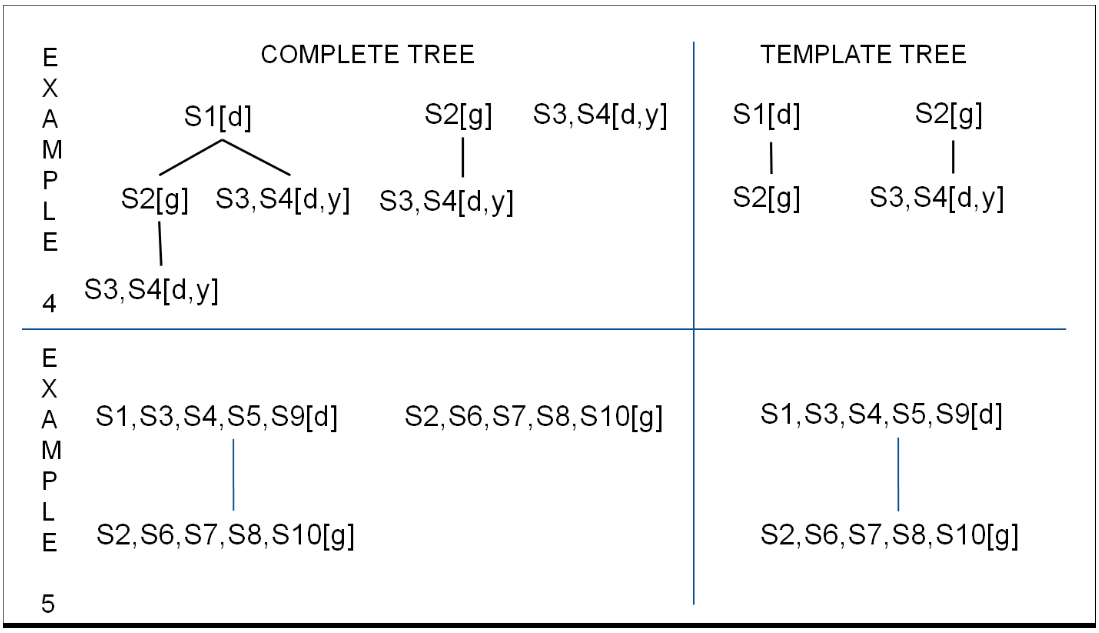


Figure 6.4: Comparison of full trees and template trees. For lack of space, the name of the attributes is represented in the abbreviated form.

Algorithm 13 shows how to break a join list into smart JLs. Given a join attribute, AVaTa lookups provide the values that appear under this attribute in different relations. A smart JL can be associated with each of these values. Given a value, a list is created with the relations that are indexed under the value($val(a, v)$) and that appear in the original JL. If the list contains more than one entry, it is transformed into a smart JL.

To illustrate the creation of smart JLs, consider Example 7. Additionally, consider the wrapped relations presented in Figure 4.3.

Example 7: Find the title, genre and year of movies.

Q_7: **title genre year**.

Given Example 7, a table search is able to find one wrapped relation that covers all variables of the query(S_5). For additional (complete)rewritings, information from multiple sources need to be joined.

```
findCombinations()
$C \leftarrow \{\}$//list of combinations
for every root node $n_i$ of the template tree do
    for every relation $r \in n_i$ do
        create combination $c$
        add relation $r$ to combination $c$
        $findCombinations(n_i, c)$
    end for
end for
findCombinations(Node n, Combination c, List C)
if ($n$ is a leaf node) then
    add combination $c$ to list $C$ //combination is valid
end if
for every child node $n_i$ of $n$ do
    for every relation $r \in n_i$ do
        create combination $c_{new}$
        add relation $r$ to $c_{new}$
        $findCombinations(n_i, c_{new}, C)$
    end for
end for
```

Algorithm 12: The algorithm that finds valid combinations from a template tree

Two join attributes are available for Q_7, as presented in the AVaTa index of Figure 6.1 (*title* and *year*). Choosing *title*, a join list would be composed by relations (S_1, S_2, S_3,S_4), which are the incomplete relations that cover the join attribute and another variable of the query.

According to the AVaTa, movies that are overlapped by *title* can be found at (S_2,S_5), (S_5,S_6) and (S_3, S_4, S_6). From these possibilities, relations that do not appear in the original JL are removed. Consequently, the resulting lists contain one entry in the first case(S_2), no entries in the second and two entries in the third(S_3, S_4).

Recall that rewritings are produced by combining relations from a JL. Therefore, join lists with less than two relations are irrelevant, since they do not have enough entries to generate combinations. In the example given, only the third case contains the minimum number of relations. The resulting smart JL (S_3,S_4) is half the size of the original one (S_1, S_2, S_3,S_4).

This approach presents drawbacks and benefits. A drawback is that is does not prevent duplicated rewritings from being produced. It can happen when relations share more than one value with each other. In this case, they would be part of more than one smart JLs. If these relations form a rewriting in one smart JL, they will form the same rewriting in all other smart JLs. One idea to prevent duplicate rewritings is to keep track of all produced rewritings so that duplicated ones are not allowed.

On the benefits side, the advantage of breaking the JL into smaller ones is deeply related to the data distribution of the indexed relations. If shared values are scarcely distributed in the available sources, the reduction of the join list tends to be considerable. The experiments in Chapter 7 show how this approach reacts in real cases,

```
joinListBreaker(JoinList JL)
a ← join attribute of the JL
R ← list of relations of the JL
JoinLists ← {} //list of the smaller JLs
for every value $v \in val(a)$ do {//iterating through values using AVaTa}
    $JL_{new} \leftarrow rel(a, v)$ //a new smart list is created
    $JL_{new} \leftarrow JL_{new} \cap R$
    if ($|JL_{new}| >= 2$) then
        add $JL_{new}$ to JoinLists
    end if
end for
```

Algorithm 13: The algorithm that breaks a JL into smart JLs

when a large number of sources are available.

An indirect benefit of using the AVaTa index is that it prunes combinations that would produce empty rewritings. For instance, without smart JLs, the combination algorithms would produce the following empty rewriting:

h(title,genre,year):-S_1(title,genre),S_2(title,year,_)

Empty rewritings such as the one above would only be identified during the rewriting check. The cost of identifying rewritings that fail to satisfy join predicates is eliminated if the smart join lists are used.

Moreover, if no selection predicates are provided in the query, the rewritings found do not need to go through the rewriting check. The smart JLs assures that all involved relations in a rewriting share at least one value for the join attribute, which makes the check unnecessary.

The check is still required when the query contains selection variables, since it would not be possible to assure that some of the tuples that satisfy the join predicate also satisfy the selection predicates. However, it is possible to optimized the rewriting check when the rewritings are produced based on smart join lists.

Algorithm 14 shows how to adapt Algorithm 8 to achieve this optimization. In the original algorithm, joins are checked by comparing tuples to see which ones share values. In the new version, the AVaTa index indicates directly which tuples share values.

6.3.6 Ranking

This section presents the strategy we have adopted to rank rewritings generated from join-less queries. With this type of query, the classification of rewritings as single-relation and multi-relation serves as a natural way to group rewritings that should be considered more relevant. The list below indicates the chosen order of relevance:

1. Complete single-relation rewritings
2. Complete multi-relation rewritings
3. Incomplete single-relation rewritings

```
checkRewriting(Rewriting w)
//check the filters
//...
//check the joins
$a$ = join attribute of w
$v$ = value used in the smart join list that originated w
for every pair of relation subgoals $g_i$ and $g_j$ do
    $r_i$ = indexed relation mapped to $g_i$
    $r_j$ = indexed relation mapped to $g_j$
    $auxList_i = \{\}$ //list of values of $r_i$ that satisfy the join
    $auxList_j = \{\}$ //list of values of $r_j$ that satisfy the join
    for each relation $r$ of $rel(a, v)$ do {//iterating through relations using AVaTa}
        if ($r == r_i$) then
            add $ide(r_i)$ into $auxList_i$
        end if
        if ($r == r_j$) then
            add $ide(r_j)$ into $auxList_j$
        end if
    end for
    if (not intersect($validTuples_{ri}$, $auxList_i$)) or (not intersect($validTuples_{rj}$, $auxList_j$)) then
        //Error - the rewriting does not cover all predicates of the query.
    end if
end for
```

Algorithm 14: Rewriting Check using information from the AVaTa and the smart JLs.

Complete rewritings are intuitively more relevant than incomplete ones, since they cover more variables. Besides, complete answers that come from a single source are also deemed more relevant, mainly because there is no need to perform joins.

The benefit of separating rewritings into these three groups is that we obtain a partial ordering without doing any ranking-related processing. Of course, further ranking is necessary for rewritings that belong to the same group. In the case of single-relation, the order is based on the table match similarity presented in last chapter. On the other hand, the order among the multi-relation rewriting is computed differently.

Two criteria are provided. The first is to consider the uniqueness factor of the join attribute. Generally speaking, rewritings whose join attribute has a higher uniqueness factor are deemed more relevant. The intuition behind this criteria is that some join attributes are not appropriate for joining, and this fact is related to the uniqueness of the attribute.

For instance, query Q_7 has two possibilities of join attributes: *title* and *year*. However, rewritings created with *year* as the join attribute are likely to bring incorrect answers, since *year* does not uniquely identify tuples of movies. Considering the uniqueness factor of the AVaTa index (Figure 6.1), rewritings that use *title* as the join attribute will get a better ranking position.

Another criteria is necessary to rank rewritings that use the same join attribute. In such cases, rewritings that use less relations are considered more relevant, since they decrease the cost for finding the answers by performing a smaller number of joins.

As discussed in the previous chapter, the relevance of the relations can be computed by additional ways, such as the page rank of where the relation was extracted. The number of tuples of the participating relations can also influence the ranking. Moreover, the number of tuples of the final answer would be a good indicator of relevance. This information is directly available for single-relation rewritings that contain at most one selection variable, through ATaVa lookups. In other cases, the number of tuples is available only after the query processor builds the answer. All of the above are possibilities not explored in this work, but that could be easily supported by the proposed architecture.

6.4 Chapter Remarks

This chapter demonstrates our second solution to the querying of structured and heterogeneous sources on the Web. Oppositely to the first, the query language is simpler, similarly to the keyword queries found in general search engines. Besides, joins are not allowed. Rewritings are classified as single-relation and multi-relation, where the latter indicates the fact that multiple relations are joined to find the results.

It is shown that single-relation rewritings can be produced using the table search algorithms presented in Chapter 5. On the other hand, multi-relation rewritings are found using a new approach. To support this idea, we extend the proposed indexes with the AVaTa, and propose combination algorithms that arrange incomplete relations in order to form multi-relation rewritings that cover all variables of the query.

The AVaTa is particularly useful to improve the efficiency of the combination algorithms, by reducing the number of relations that need to be combined. Furthermore, in some cases it enables the generation of non-empty rewritings without the need to perform the rewriting check. The benefits of this complementary index are discussed in the experimental results chapter. Among other things, this chapter also compares the two rewritings approached proposed in the thesis.

7 EXPERIMENTS

This chapter presents the results obtained when evaluating the proposed approaches for indexing and query rewriting. We measure the table search algorithms with respect to efficiency and ability to find all possible answers(recall). The combination algorithms related to join-less queries are also compared. Additionally, we assess the importance of the rewriting check and the memory cost of the proposed indexes. Finally, we present a search engine application build based on the ideas of this thesis.

7.1 Table Search Algorithms

The data set used to evaluate the table search algorithms contains synthetic tables. The tables were generated based on the information described in Table 7.1, which defines the probability of each attribute being part of a table. The underlined attributes (`title`, `film`, `movie`) have a disjoint probability of belonging to the same table (their individual probabilities sum up to 100%).

The distribution of attributes presented in Table 7.1 was derived from a collection of 671 tables that we have collected from the Web (see Section 7.3 for details). Based on the distribution of the Figure, a data set of 10.000 tables was created.

Attribute	Frequency
other_notes	85%
<u>title</u>	75%
notes	65%
director	58%
genre	56%
cast	55%
year	30%
role	17%
<u>film</u>	17%
category	8%
<u>movie</u>	8%
rank	1%

Table 7.1: Attribute frequencies used to define the computer generated tables

The experiments were conducted on a Pentium Core 2 Duo with a 2.67 GHZ processor and 4GB of RAM. Each experiment was executed 100 times, and we report the one that took the lowest time.

7.1.1 Measuring Efficiency

We measured the cost in time for the three table search strategies presented in this work (Table Scan, Pivot Table Scan, Table Intersection).

The query workload used in the experiments in described in Table 7.2. The number of query variables of the workloads changes iteratively according to the iteration. Furthermore, assume that the variables belong to the same relation subgoal, and they are either all required or all optional.

For example, in the first iteration, workload 1 contains variable *rank* only, whereas in the third iteration it contains variables *rank*,*category* and *role*.

In the first and second workloads, queries in the succeeding iterations are added with variables that are more frequent in the indexed relations. In the third and fourth iterations, queries in the succeeding iterations are added with less frequent variables. Moreover, queries in the first and third workloads contain only required variables.

Iteration	Workload 1	Workload 2	Workload 3	Workload 4
	all required	all optional	all required	all optional
1	rank	rank	notes	notes
2	category	category	title	title
3	role	role	other_notes	other_notes
4	year	year	director	director
5	cast	cast	genre	genre
6	genre	genre	cast	cast
7	director	director	year	year
8	other_notes	other_notes	role	role
9	title	title	category	category
10	notes	notes	rank	rank

Table 7.2: Query workloads - each workload is divided in 10 iterations.

Figure 7.1 shows the results obtained after running the first and second workloads, where the queries start with less frequent variables and are added with more frequent ones.

We remark that the table search strategies are optimized to explore the less frequent attributes of the query. In the table scan, these attributes are processed first. Likewise, in the table intersection, smaller lists of tables are intersected first. In the pivot table scan, the less frequent attribute becomes the pivot. The results of this tuning are described next.

When all variables are required (workload 1), table scan presents a behavior that oscillates to a constant cost, as the number of variables increase. The reason is that few relations cover the less frequent variables. As they are added to the query, the chances are that the algorithm discovers that a relation does not provide a full coverage before all variables are processed.

Pivot table scan behaves similarly. The difference is that it does not process all indexed relations, only those that contain the attribute *rank*, the less frequent variable that is present in all iterations of the workload. It is notable the benefits of using a pivot to reduce the search space.

Table intersection initially has a increasing cost, and then remains constant when five or more variables are used. This mark indicates a relation subgoal that is not

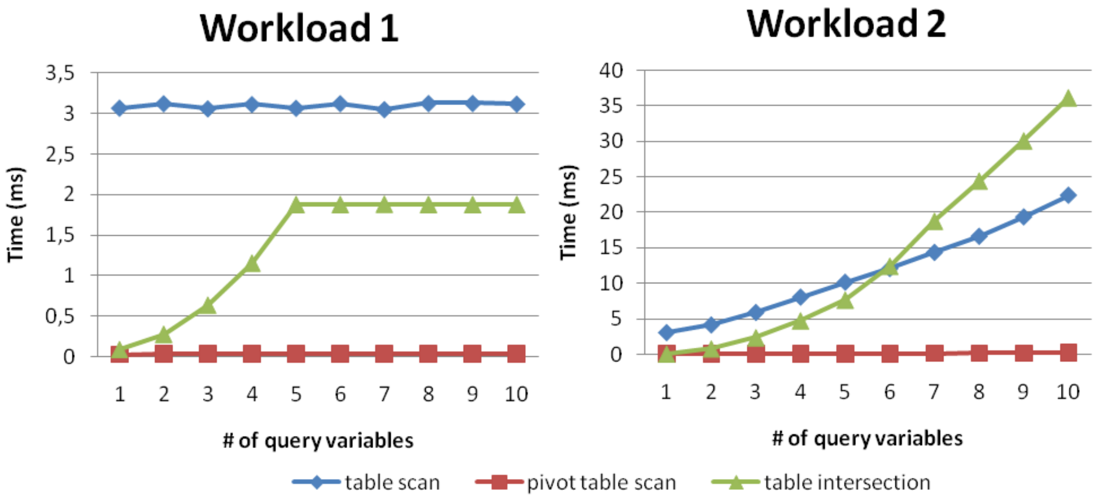


Figure 7.1: Efficiency results for workloads 1 and 2

fully covered by any indexed relation. Therefore, adding variables has no effect since the algorithm stops processing when the intersection of relations becomes empty, which happens when the five less frequent variables are processed.

The cost of the pivot table scan does not change when the query contains only optional variables (workload 2). The reason is that, in both cases, the attribute *rank* is used to select the relations to process. The drawback is that not all possible table matches can be found, only those that contain the *rank* attribute. Experiments in Section 7.1.3 presents the recall when the pivot table scan answers queries where all variables are optional.

As the Figure 7.1 shows, table scan has a linear cost in workload 2. Differently than in workload 1, all variables of the query are processed, which explains why the performance is proportional to the number of variables.

When all variables are optional, the table intersection simply increases the list of table matches (no intersection is performed). In the first iterations, when attributes are less frequent, the number of table matches are small. In the final iterations, when more frequent attributes appear, the number of table matches is much higher. Therefore, the amount of work is higher during the last iterations, which explains the curve presented in the Figure.

Figure 7.2 shows the results obtained after running the third and forth workloads. As opposed to the first case, queries in these workloads start with more frequent variables and are added with less frequent ones.

When all variables are required (workload 3), the three search strategies have one point in common: The cost reaches a peak in iterations 3-5, and then gradually reduces. The descent in the Figure happens because the iterations that are closer to the end contain less frequent attributes. When a new less frequent attribute is added to the query, the algorithms use it to perform the search. Since the less frequent attribute is processed first, the number of relations processed by the pivot table scan and table intersection is smaller for the last iterations. Similarly, the table scan strategy is able to verify that a relation does not match the query sooner, since it checks the coverage for the less frequent attributes first.

The ascend before the peak happens because the first iterations contains attributes that are relatively frequent, and the cost to process one more attribute in

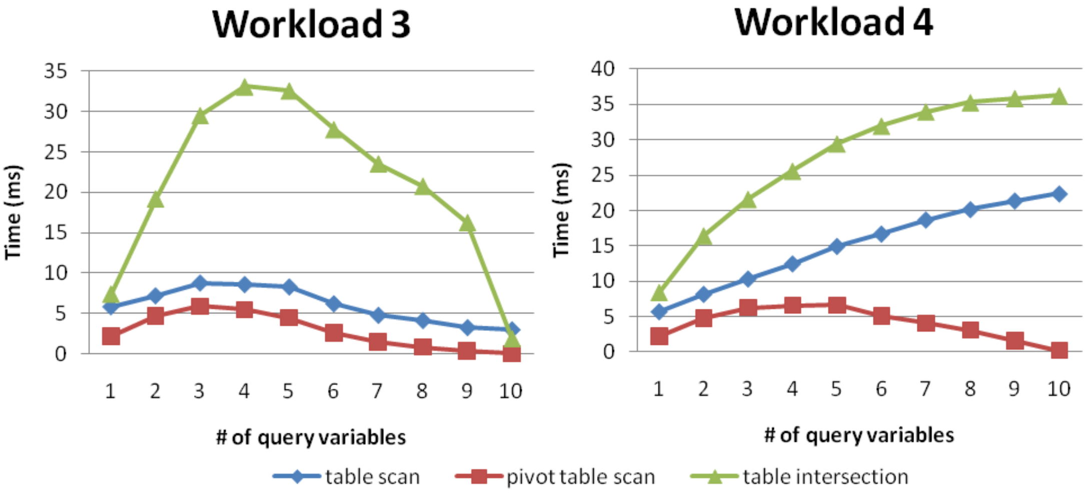


Figure 7.2: Table search efficiency results for workloads 3 and 4

a succeeding iteration overcomes the gain that the added less frequent attribute brings.

When all variables are optional (workload 4), the cost does not decrease for the table scan and table intersection. The reason is simple: all attributes need to be processed, since the search cannot stop when the algorithm verifies that an attribute is not covered. However, the cost for the table intersection is not linear as it is for the table scan. This difference occurs because the amount of work is higher during the first iterations of the table intersection, which is a consequence of having queries with more frequent attributes in the first iterations. Observe that the curves for workload 2 and 4 resemble an inverted image of each other (for the table intersection), where the only difference between the workloads it that they have an inverted order of the iterations.

Additionally, observe that the pivot table scan curve behaves similarly to the one presented in workload 3. The reason is that this algorithm assumes that the less frequent attribute in the query is required, which ends up reducing the cost in the final iterations.

Comparing the four workloads, we can see that pivot table scan is much faster than the other two. However, it may miss rewritings, for queries that only contain optional variables, as will be discussed later. The differences between the table scan and table intersection is deeply related to the query. As the experiments show, table intersection is better for queries that contain low frequency attributes. Considering horizontal search engines that needs to index information from disparate domains, it is very likely that a major number of attribute have low frequencies. In such cases, table intersection would be more efficient than having to process all indexed relations. Observe that, if the query contains at least one required variable, pivot table scan would be a better choice, since no rewritings would be missed.

7.1.2 Measuring Scalability

The scalability of the table search algorithms is depicted in Figure 7.3. The figure shows the results obtained when running queries from the workloads 1 and 2. The collection of query variables are taken from the last iteration of the workloads, where 10 variables are requested.

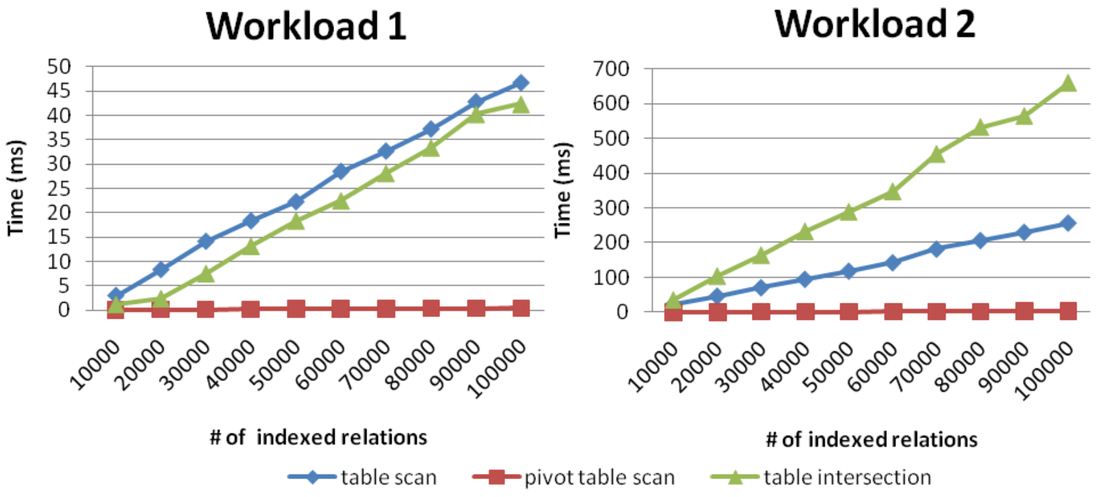


Figure 7.3: Table search scalability results for workloads 1 and 2

The cost in time grows linearly as the number of indexed relations increases. As expected, pivot table scan is the most efficient. The cost is also linear, but does not increase as much as the other two search approaches.

Observe that the table intersection is better than table scan when all variables are required. On the other hand, table scan is better when all variables are optional. The results are related to the queries used in the experiment. As discussed in the previous Section, depending on the query, table intersection can be better than table scan even with only optional variables.

It is also important to remark that dealing with optional variables is much more time consuming than dealing with required variables, as the Figure shows. The table search algorithms work better with required variables because they are able to stop processing a relation when they discover that the relation does not cover one of the variables.

7.1.3 Measuring Recall

As explained earlier in the Chapter, pivot table scan may not retrieve all possible table matches when all variables are optional. In this section we present experimental results that show the recall of the algorithm in such cases. The expected answers used to measure recall were obtained by running either the table scan or table intersection, as they both bring all possible answers.

Workloads 2 and 4, composed by optional variables, are used for the experiment. Besides, we have tested two different flavors of pivot table scan: Most Selective and Less Selective. The first follows the traditional idea of using the less frequent attribute as the pivot, in order to increase efficiency. The second uses the most frequent attribute as the pivot. This setting is less efficient, but naturally is able to find more answers, as the results in Figure 7.4 show.

In the first iteration, the settings do not miss any rewriting, since only one variable is requested. In the last iteration, when 10 variables are requested, the difference is notable. For the Less Selective, the recall in workload 4 reaches a point where it is close to constant. This happens because the iterations succeed each other by adding less frequent variables. Given a new variable in the query, there is hardly any rewriting that contains this variable and was not yet found by the previous iteration, where that variable was absent.

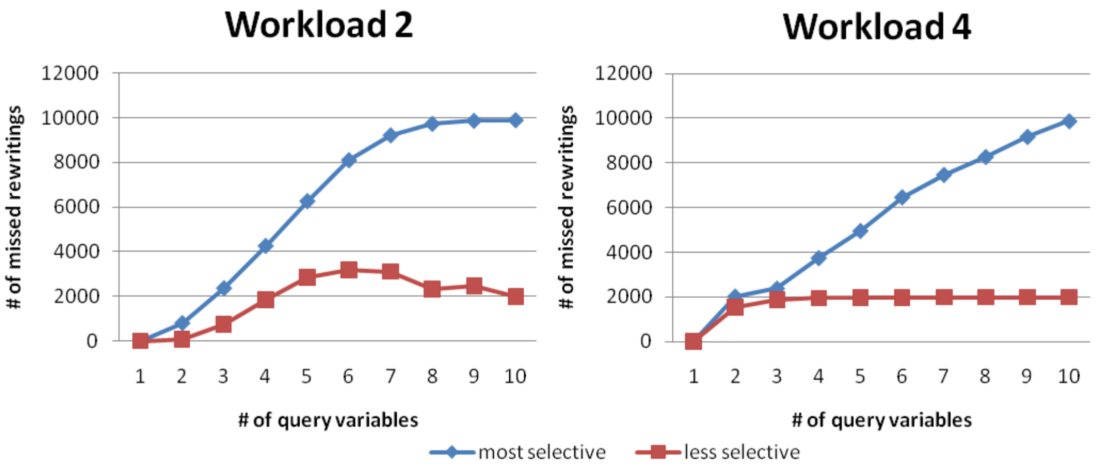


Figure 7.4: Number of missed rewritings when pivot table scan process queries with optional variables

Interestingly, workload 2 has an decreasing recall until variable *genre* is used as the pivot (iteration 6). The total number of rewritings in the first iterations is lower than the final ones, which helps explaining why the number of missed rewritings increase in the beginning. During the final iterations the recall starts to increase again, since the variables that are added to these iterations appear in a high number of relations.

Figure 7.5 shows the same results measured by the relative recall, with respect to the total amount of possible rewritings. Recall drops severely in the Most Selective setting when the query contains non-frequent variables, since it is biases by the frequency of its less frequent variable, as indicated in Table 7.1. On the other hand, recall is much higher when the Less Selective setting is used.

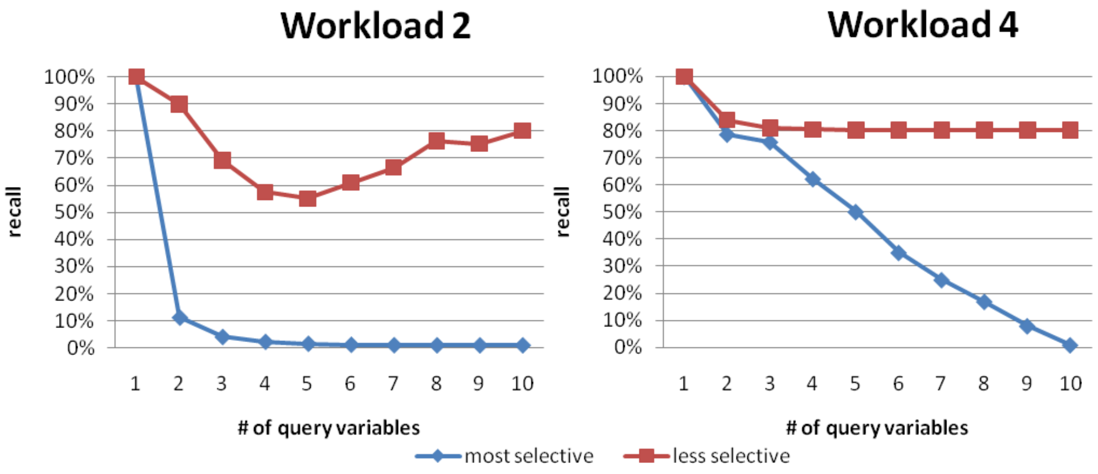


Figure 7.5: Recall obtained when pivot table scan process queries with optional variables

Even though the Less Selective approach outperforms the other, and recall is meaningful, the number of missed rewritings deserves attention. For instance, take iteration 4 in workload 2, where the query requests four non-frequent variables. This would be a normal request in a structured search engine that indexes a large number of attributes. In this case, the number of missed rewritings is close to 2000. It represents 20% of the whole set of indexed relations. In larger repositories, the number would be even greater. Besides, it is important to remark that our data set contains a reduced number of attributes, with a individual frequency that could be considered high. In real world domains, this frequency would be much lower. Consequently, the number of indexed relations that share the same non-frequent attributes would be minimal, and this fact would have a direct impact on the pivot table scan recall.

7.2 Combination Strategies

In this section we discuss how variations of the combination strategies affect the cost to find multi-relation rewritings.

Initially we compare the Naïve, the Stream-Driven and the Template-Driven strategies. In essence, these algorithms receive join lists and compute valid combinations. Therefore, we measure their efficiency by artificially creating join lists.

The join list used in the first experiment is composed by two different complements: [year] and [rank]. The number of entries that correspond to each complement varies according to the iteration of the execution. In the first iteration, each complement can be found in one entry of the join list. In the fifth (last) iteration, each complement can be found in five entries of the join list. Table 7.3 shows information related to each iteration. Observe the relation between the number of possible combinations and the actual number of valid combinations. Recall that, in order to be valid, a combination must be complete and minimal.

Iteration	rank entries	year entries	sum	possible comb.	valid comb.
1	1	1	2	3	1
2	2	2	4	15	4
3	3	3	6	63	9
4	4	4	8	255	16
5	5	5	10	1023	25

Table 7.3: First experiment: the number of entries changes according to the iteration

For the second experiment, each iteration works with join lists that contain different sets of complements, as depicted in Table 7.4. There are two entries to each different complement. This number remains the same throughout the iterations. For lack of space, the complements are indicated in the abbreviated form.

Figure 7.6 presents the results found when the combination algorithms process the join lists mentioned above. Other setting of join lists were tested as well, and they present curves similar to the ones illustrated in the Figure.

As expected, the performance becomes more expressive as the total number of possible combinations increase. We can see that the Naïve approach takes exponential time to compute valid answers, since it needs to compute all possible combinations. The Stream-Driven approach presents a great improvement over the Naïve, since it is able to reduce the number of computed combinations. The best results are achieved with the Template-Driven approach, mainly because it is capable of

Iteration	distinct complements in the JL	sum	possible comb.	valid comb.
1	[r],[g]	4	15	4
2	[r],[g],[y]	6	63	12
3	[r],[g],[y],[d]	8	255	24
4	[r],[g],[y],[d],[g,y]	10	1023	32
5	[r],[g],[y],[d],[g,y],[d,r]	12	4095	44
6	[r],[g],[y],[d],[g,y],[d,r],[g,r,y]	14	16383	48

Table 7.4: Second experiment: the number of different complements changes according to the iteration

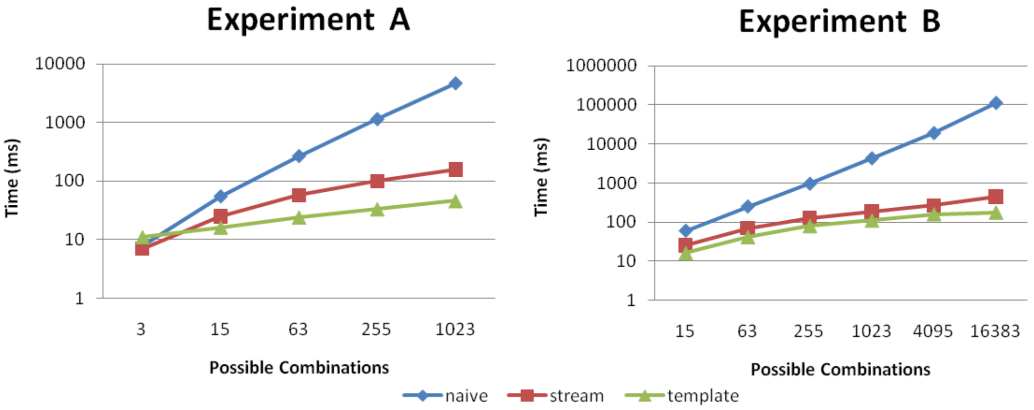


Figure 7.6: Efficiency results for the combination strategies

grouping entries with the same signature.

Interestingly, Experiment A shows that the Template approach is worst than the stream approach in the first iteration. The reason is that the join list is two small in this case, and the cost of computing template trees becomes the most expensive operation when the join lists are too small. Future work could investigate how to automatically choose the best strategy for each case. One possible tuning would be to use the template tree only when a join list is larger than a pre-defined threshold.

The third experiment includes the AVaTa index optimization in the measurement. As explained in Section 6.3.5, this index is able to create smart join lists that together tend to be smaller than the original join list. From the smart JLs it is possible to directly compute all valid rewritings that are not empty, without performing the rewriting check.

To evaluate the AVaTa optimization, we created a JL composed by two different complements: [rank] and [genre]. There are 50 entries that correspond to each complement, which gives us 2500 valid combinations. However, it is very likely that only a few of those are not empty.

No AVaTa index is actually used in the experiment. Instead, we simulated the number of smart lists the index would compute, based on the maximum number of valid combinations, as indicated in Table 7.5. For instance, in the first iteration, 125 lists are created, which represents 5% of the total number of combinations(2500). Additionally, we stipulated than each list is composed by two relations only. Therefore, at most one rewriting may exist for each smart list.

After splitting the original join list into the smart lists, the Template-driven approach was used to compute the rewritings. Recall that the ATaVa optimization may generate duplicated rewritings, for instance, when two smart join lists contain the same relations. Therefore, when all rewritings are put together, it is necessary

Iteration	percentage	amount
1	5%	125
2	10%	250
3	15%	375
4	20%	500
5	25%	625
6	30%	750
7	35%	875
8	40%	1000
9	45%	1125
10	50%	1250

Table 7.5: The number of simulated smart lists changes according to the iteration

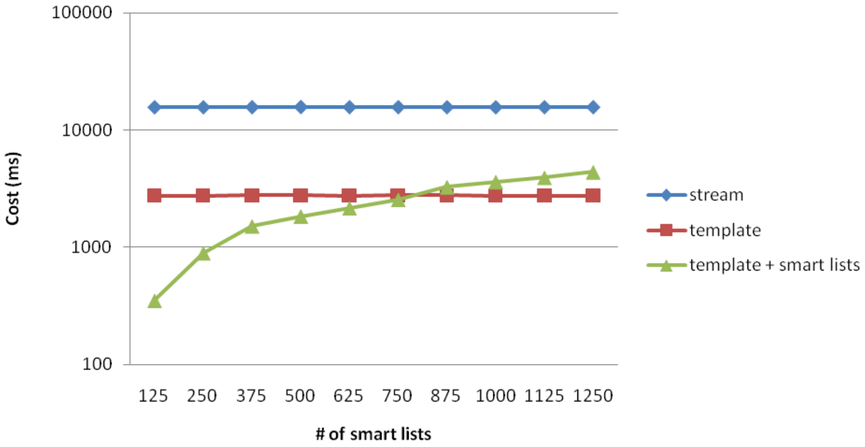


Figure 7.7: Efficiency results when smart lists are used

to ignore the duplicates. This step is also included in the evaluation.

Figure 7.7 shows the time cost to find the rewritings for the Stream-Driven, the Template-Driven and the Template-Driven with smart lists. The first two remain constant, since the cost is not affected by the number of smart lists (their processing is based on the original list). We can see that the ATaVa brings an improvement when the number of smart lists is relatively small. The turning point occurs when more than 750 lists needs to be processed. The results show that, for this particular case, it is faster to use one single list with 100 entries than to use 750 lists with 2 entries each - in the former, only one template tree is computed, against 750 template trees in the second.

It is important to remark that different settings could yield different curves, for instance, when more query complements exist(in the example, there are only two: [rank] and [genre]). One possibility is to automatically analyze the scenario in order to decide whether smart lists should be used. We leave these considerations as ideas for future work. Another thing that deserves attention is the cost to perform the rewriting check, which is not needed when the smart lists are used, and given queries with no selection predicates. This cost is not taken into account in the experiment, which suggests that the benefits of the smart lists exceed the border depicted in the Figure.

7.3 Rewriting Check

The rewriting check is able to identify empty rewritings and prevent them from being presented to the user. In this section we measure the effort the user would spend to find non-empty rewriting if this check is not performed.

To evaluate this type of check, the repository was populated with HTML tables crawled from the Web. We have implemented a simple parser that automatically extracts information from well-behaved HTML tables (i.e., tables whose header row is defined with TH tags). A Wikipedia page that brings a list of the 100 best American movies [1] was used as the seed for the crawl.

The crawler focus on pages that contains at least one HTML table where one of the following values can be found in the table header: `title`, `film` or `movie`. The goal of this crawling template is twofold:

- detect Web tables that are used for data tabulation purposes instead of formatting purposes.
- discover relevant sources of data for the movie domain.

This data set is intended to show how our approach scales for relations that are part of the same domain.

The crawling stopped after 200 movies related wikipedia pages were retrieved. From these sources, we extracted into the repository all HTML tables that satisfy the template stated above. A total of 671 tables were indexed. By manually analyzing 10% of the indexed tables, we estimate that 95% of them are indeed part of the movie domain.

Table 7.6 shows the query workload used. The first two queries show variations in the selection predicate for the variable year. The last three queries exercise the creation of multi-relation rewritings. No indexed relation covers all variables of queries Q3 to Q5, so joins need to be performed in order to find answers. In all cases empty rewriting can be generated, either because the selection predicates or the join predicates are not satisfied.

Query	Type of Predicate
Q1: title year [year = 2000]	Selection Predicate
Q2: title year [year < 1995]	Selection Predicate
Q3: title year director cast genre notes	Join Predicate
Q4: title genre director year cast	Join Predicate
Q5: title cast role	Join Predicate

Table 7.6: Query workload used to find statistics about rewritings emptiness

Table 7.7 shows the results achieved for queries Q1 and Q2. The evaluation was performed over data sets of different sizes. Column c1 refers to the total number of rewritings found and column c2 refers to the number of rewritings that are not empty. Column c3 refers to the number of empty rewritings the user would have to open until 10 non empty rewritings are found (if no rewriting check is applied).

[1] The list was created by the American Film Institute `http://en.wikipedia.org/wiki/100_Years...100_Movies`

#tables	Q1			Q2		
	c1	c2	c3	c1	c2	c3
101	7	1	6	7	5	2
208	28	8	20	28	19	5
313	28	8	20	28	19	5
412	28	8	20	28	19	5
501	29	9	20	29	20	5
605	34	10	24	34	22	5
679	36	12	24	36	24	5

Table 7.7: Statistics about rewriting emptiness for Q1 to Q2, where c1 = total number, c2 = not empty, c3 = empty rewritings until 10 not empty

In the best case scenario, only 2 empty rewritings would have to be opened (for query Q2 and having a data set of 101 source relations). The worst case occurs for Q1 and having a data set of 679, where 24 rewritings would have to be opened. The filter predicate of Q1 is more restrictive than Q2, which explains the difference indicate in the Figure. Also, as expected, the user effort increases as the number of indexed relation increases. If thousands of relations were available, the task of finding non-empty rewritings would be unfeasible.

Table 7.8 shows the results achieved for queries Q3 to Q5. The total number of rewritings is much larger if compared to the other queries. The reason is related to the number of possible combinations of incomplete indexed relations that are able to provide a full coverage. The comparison between c1 and c2 shows that only a small part of the rewritings are non-empty. This case clearly shows that the process of finding non-empty rewritings needs to be performed by the search engine, and not delegated to the user.

#tables	Q3			Q4			Q5		
	c1	c2	c3	c1	c2	c3	c1	c2	c3
101	529	23	346	529	23	346	231	0	231
208	3780	40	1246	3780	40	1239	1755	14	570
313	6496	78	624	6496	78	628	3016	51	288
412	9212	102	1513	9212	102	1520	4277	74	719
501	12494	128	289	12006	130	269	5796	100	269
605	18469	161	829	17272	163	907	7620	116	832
679	23560	205	403	20340	207	391	9605	157	391

Table 7.8: Statistics about rewriting emptiness for Q3 to Q5, where c1 = total number, c2 = not empty, c3 = empty rewritings until 10 not empty

7.4 Comparison with Related Work

From the related work, the one of Dong and Halevy (DONG; HALEVY, 2007) is the most similar to ours, in that the ultimate goal is to seamlessly query a large corpus of structured information available on the Web. In this section we compare our approach to the one proposed by Dong.

To start, we briefly describe the two main indexes of the related work - ATIL and AAIL - that we use as a basis of the comparison. Further details can be found in Section 3.2.4.

The ATIL is a list of keywords, where each keyword corresponds to the concatenation of a value and an attribute. Each keyword contains a inverted list of tuple instances. For example, 2004//year:[T1-2, T4-3] indicates that the attribute `year` contains the value 2004 in two different tuples: T1-2, represents the second row of table T1 and T4-3 represents the third row of table T3.

The AAIL is a variation of ATIL that supports associations. An association is a role between two related tuples. Furthermore, associations are bi-directional, and each direction is given a role name. Each association is stored in the AAIL several times, one for each attribute of the tables involved in the association.

7.4.1 Memory Cost

In order to evaluate memory cost, we collect data from the WT10G data set. This collection contains over 1.5M web pages crawled from the Web (details can be found at `http://ir.dcs.gla.ac.uk/test_collections/wt10g.html`).

From these sources, we apply a more general template extraction than the one used in the movie data set. An HTML table matches a template if it contains one of the following attributes: `artist, city, company, country, name, product and title`. In the end, a total of 3471 tables were indexed (out of 2845 pages).

To compare the indexes, we implemented indexers that takes the parsed HTML tables and store it according to the ATIL and AAIL. However, our approach has no support to identifying associations as proposed in (DONG; HALEVY, 2007). Instead, we updated AAIL with the associations we were able to find automatically, in which tuples are related if they share the same attribute name and same value. The name of the association is computed as the name of the attribute prefixed with the word `same`. For instance, if the attribute name is title, the name of the association becomes `same title`, in both directions.

All indexes are represented as Java primitive data types. Plus, the relationships between the indexed elements are given by inverted lists, structured as primitive arrays. The size of an index is computed as the total amount of bytes the index occupies. It is important to remark that we care about optimizing memory consumption, for all evaluated indexes. To accomplish this, we prevent object duplicates - unique objects are represented only once. The data set used in the experiments as well as the source code that computes the memory cost are available at `http://www.inf.ufrgs.br/~heuser/atava.zip`.

Figure 7.8 shows the results achieved when measuring memory cost. Observe that the size of all indexes grows linearly with respect to the number of tables.

At the very low part of the graphic is the ATa index. Despite its low memory consumption, it does not provide enough information to perform the rewriting check. Therefore, queries with selection predicates may lead to empty rewritings.

On the other hand, the ATaVa is able to perform the rewriting check, but it takes a considerably higher amount of memory. ATaVa is slightly better than ATIL. These indexes are analogous, in the sense they do not store associations but allow queries with selection predicates.

We also compare the associations-aware indexes ATaVa+AVaTa and AAIL. The memory cost in ATaVa + AVaTa is lower that its analogous AAIL. The reason is the overhead involved when adding an association in AAIL. Adding an association to an instance with n attributes implies in adding n new keywords.

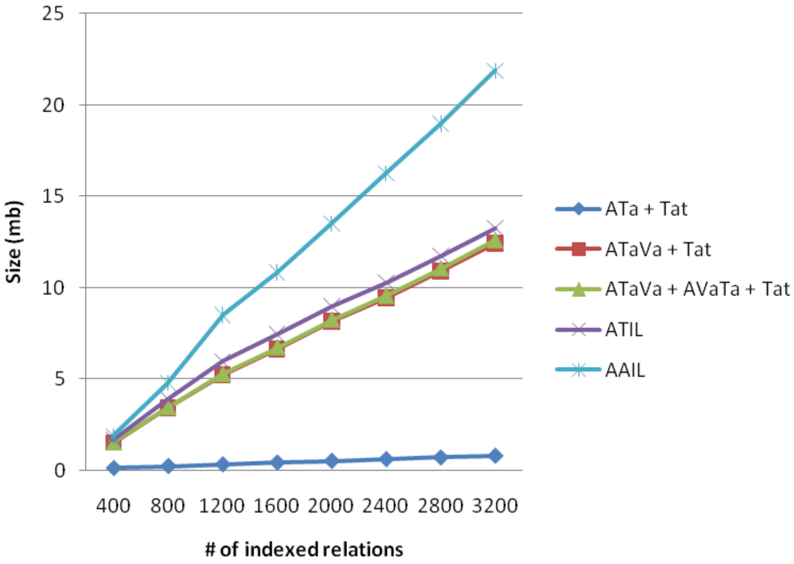


Figure 7.8: Memory consumption for the WT10G data set

Interestingly, adding the AVaTa to the ATaVa hardly increases the memory consumption. Recall that the AVaTa only keeps information about the join attributes. Naturally, the size of this index varies according to the total number of join attributes. However, we have found that only a small part of the whole data set needs to be stored in this index, even considering all existing attributes. Our statistics (considering both movie and wt10g data sets) indicate that 73% of the table values are unique, 22% of them appear in different relations, but for attributes with different names, and only 5% appear in different relations for attributes with the same name.

Finally, the TAt component is included in the three variations of our indexes. Information from the TAt is used by the table scan and pivot table scan algorithms to traverse the list of indexed tables. The TAt is created by adding links among elements that are already part of the ATa. Therefore, there is no significant overhead to keep it in the repository.

7.4.2 Answering Queries without Selection Predicates

Queries without selection predicates are those where the user only wants to discover sources than contain the specified attributes, regardless of the contents of the tuples.

In our approach, this type of query can be answered using information from the ATa. On the other hand, ATIL was designed with the purpose of answering selection predicate queries, and not to merely define the attributes of interest. In order to find out which sources contain information about a specific attribute, a full scan in the index is required, since the attribute part of the indexed keywords is hidden behind the value part.

One possible overcome would be to index additional keywords, using only attribute names. We did implemented a variation of the ATIL that adds these new keywords. The results are positive, and indicate that this variation does not have a severe impact with respect to the memory consumption.

7.4.3 Answering Queries with Selection Predicates

Queries with selection predicates are those where the user wants to discover sources than contain the specified attributes, and the values of some attributes need to be within a determined range.

The ATaVa is able to identify the indexed relations that satisfy the selection variables of a query. Furthermore, the data type information stored in the index allows the selection operations to be semantically performed.

ATIL easily supports queries whose selection predicates contain the equality operator. Tuples that satisfy this kind of predicate are directly identified in the index. However, other kinds of operators require additional processing. In such cases, the index would have to be scanned to find out which values satisfy a selection predicate. After the values are found, another step is required to check if the indexed attribute is the one of the predicate. Additionally, the index does not smoothly accommodate data types, so values would always be compared as strings.

7.4.4 Index Distribution

The ATaVa/AVaTa indexes can be distributed in several nodes, creating partitions at the attribute level. Given a query with n attributes, at most n nodes would have to be accessed, considering the all requested attributes are kept in distinct nodes. This organization helps to amortize the cost for storing the tuples, which is the information that represents the higher amount of memory, as indicated in Figure 7.8.

Considering a distributed index, the TAt component would have to be kept in a separate node. However, the cost of this index is low, since it does not care about tuples. The memory consumption is similar to the ATa, where the relationship among attributes and relations occur in the opposite direction. Also, since the TAt is no longer kept in the same node, the indexed objects would have to be uniquely identified across the indexes, so that the rewriting algorithms are able to correlate them.

In ATIL, the entries are prefixed by values. Therefore, there is no meaningful way to distribute the information in several nodes. One idea would be to create partitions at the values level. In this case, if a selection predicate uses the equality operator, only the node that keeps the specified value needs to be accessed(eg. `"find information where year = 2000"`). However, if the selection predicate uses any other operator, defining the necessary nodes would be a complex and time consuming task(eg. `"find information where year > 2000"`). A similar problem occurs if a requested attribute is not part of any selection predicate. This situation would required the index to be adapted, as already mentioned in Section 7.4.3.

7.4.5 Associations Discovery

This thesis describes two different ways of handling associations. In the first, no associations are indexed, and it is up to the user to specify the join among the sources. In the second, associations are indexed between tuples that share the same column name and value. On the other hand, AAIL supports a more general kind of association. An association is a role between two tuples, and its not conditioned to the equality of column names. However, automatically finding associations between tuples and defining their role names is a complex process, and probably would have to be manually consolidated. In (DONG; HALEVY, 2007), it is not clear how this information is discovered.

The number of associations found by our indexer is lower than associations found by the AAIL, but it is a process that can be performed automatically. Of course, this comes with a price of lower precision / recall rates, specially when mixing information from multiple domains into a single index.

Nevertheless, in the context of exploratory search, this behavior is acceptable, and sometimes, can even produce interesting answers. For instance, given the movie source `(title, director)` and the book source `(title, year)`, the query `title director year` would return the title and director of a movie, along with the year in which a homonymous book was released.

One limitation of the way in which we use associations in rewritings is that the join attribute needs to be specified in the query. If the provided attributes of the query are not good join attributes, relevant associations may not be found.

We intend to leverage this limitation by artificially adding join attributes to the query. Given a query, schema-completion techniques (CAFARELLA et al., 2008) can be used to discover which lacking attributes best match the schema of the query, and try to use these as join attributes.

7.4.6 Answering Keyword Queries

Keyword queries (where no metadata information is provided) are easily answered with the ATIL, since all indexed keywords are prefixed by the value. For example, the query "The Aviator"is able to find relations that contain the terms "the"and "Aviator"as contents of an attribute.

In our approach, keyword queries cannot be answered, since the value information is hidden behind the attributes level.

Observe that the ability to answer keyword queries is orthogonal to the ability to answer general predicate queries, and this is a direct consequence of the index organization. ATIL/AAIL are based on values, which makes them more powerful for keyword search. On the other hand, ATaVa/AVaTa are based on attributes, which makes them more powerful for predicate queries, including those that use operators other than equality.

8 EIDOS PROTOTYPES

This chapter presents prototypes of a structured search engine, build based on the concepts of the Eidos architecture. Two versions are presented: one that supports queries with joins and another one that supports join-less queries.

8.1 SPJ Queries Support

Figure 8.1 shows the interface that allows users to express SPJ queries in SQL. To demonstrate, we have populated the repository with HTML tables from the movie data set, using the extraction rules presented in Section 7.3.

The Figure shows a list of rewritings that contain possible answers to an SQL query asking for the title, year and notes of movies. Close to the rewritings is a summary of the involved indexed relations as well as links to the URL of the Web pages where the HTML tables are located.

When the user clicks on a rewriting, the search engine accesses the actual Web page and extracts the desired information from the HTML table located in that page. The output of the selected rewriting (Rewriting 2) is displayed to the user in a tabular format, composed by a list of records and segmented as one or more attributes.

The query of Figure 8.1 contains one relation subgoal, so each rewriting retrieves answers from a single indexed relation. When queries contain multiple relation subgoals, the search engine delivers rewritings that access more than one indexed relations to find the answers.

The next example compares a result computed by the search engine to the actual source from where the result was obtained. In this case, the user wants to explore information about actors and awards. The first rewriting returned is shown in Figure 8.2, together with an excerpt of the Web page from where the information was extracted. Since the data is retrieved from the source, the result is always fresh.[1]

Observe that the content of the output table is a subset of the content that can be found in the source HTML table, which includes links and images (relative links are automatically converted to their respective absolute links). By keeping the hypertext-based information we give the user the power to navigate through the results, exploring its contents and discovering additional information that are related to the query.

[1] For efficiency, results could also be cached.

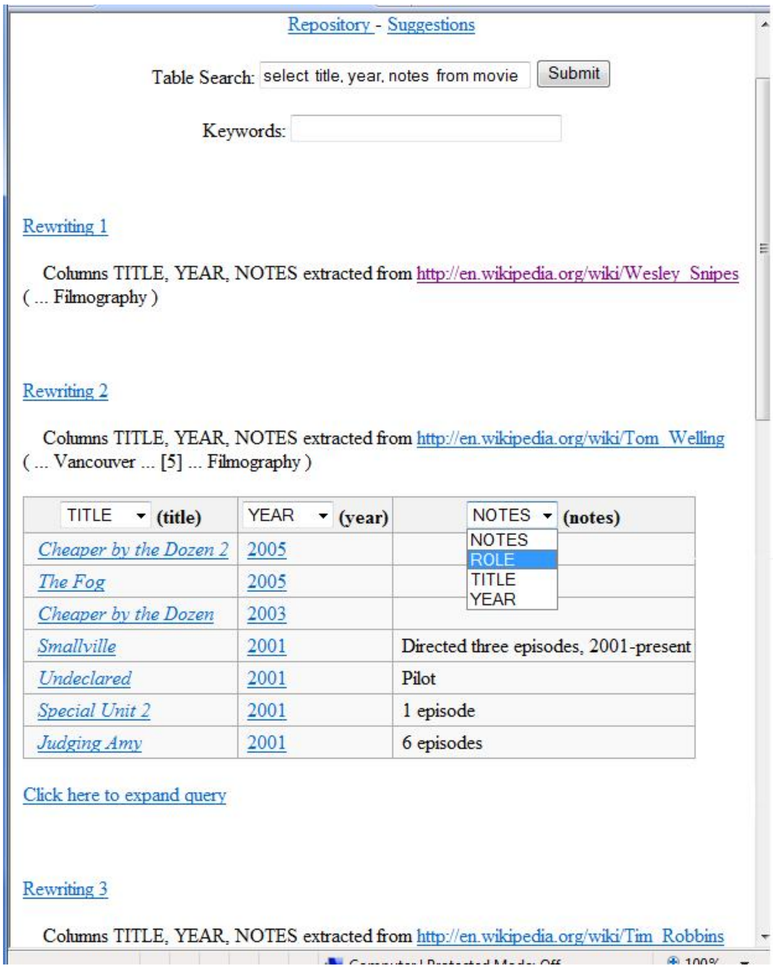


Table Search: select title, year, notes from movie

Keywords:

Rewriting 1

Columns TITLE, YEAR, NOTES extracted from http://en.wikipedia.org/wiki/Wesley_Snipes (... Filmography)

Rewriting 2

Columns TITLE, YEAR, NOTES extracted from http://en.wikipedia.org/wiki/Tom_Welling (... Vancouver ... [5] ... Filmography)

TITLE (title)	YEAR (year)	NOTES (notes)
Cheaper by the Dozen 2	2005	
The Fog	2005	
Cheaper by the Dozen	2003	
Smallville	2001	Directed three episodes, 2001-present
Undeclared	2001	Pilot
Special Unit 2	2001	1 episode
Judging Amy	2001	6 episodes

Click here to expand query

Rewriting 3

Columns TITLE, YEAR, NOTES extracted from http://en.wikipedia.org/wiki/Tim_Robbins

Figure 8.1: Query interface of the structured search engine. Users pose SQL queries and the system retrieves HTML tables that best match the queries.

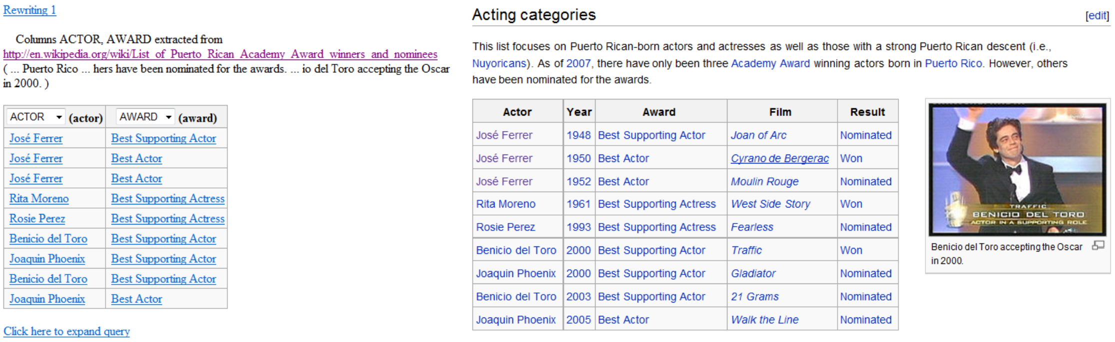
8.2 Join-less Queries Support

In this section we present a prototype that accepts join-less queries, expressed in the proposed keyword language. The application is similar in spirit to the one that accepts SPJ queries. Therefore, only the different feature is shown here, which is related to the ability to find multi-relation rewritings.

To demonstrate how the search engine behaves on real data retrieved from the Web, the tables extracted from the WT10G collection compose our data set. This collection is much larger than the movie data set and the universe of discourse is much more diverse, which composes a scenario that better represents the heterogeneity found over the Web.

Interestingly, from the seven join attributes used in the extraction template, several different domains could be retrieved, such as restaurants, industry and politics. Therefore, this data set is particularly interesting for wide purpose queries, such as the ones asked in a horizontal search engine.

Figure 8.1 shows a list of arbitrary queries posed to our search engine. Queries from 1 to 5 are answered using single-relation rewritings (a single Web table is accessed). For instance, one of the rewritings for the first query refers to a relation that contains a list of 16 cities of Utah. The rewriting check assures that only



Rewriting 1

Columns ACTOR, AWARD extracted from
http://en.wikipedia.org/wiki/List_of_Puerto_Rican_Academy_Award_winners_and_nominees
(... Puerto Rico ... hers have been nominated for the awards. ... io del Toro accepting the Oscar in 2000.)

ACTOR (actor)	AWARD (award)
José Ferrer	Best Supporting Actor
José Ferrer	Best Actor
José Ferrer	Best Actor
Rita Moreno	Best Supporting Actress
Rosie Perez	Best Supporting Actress
Benicio del Toro	Best Supporting Actor
Joaquin Phoenix	Best Supporting Actor
Benicio del Toro	Best Supporting Actor
Joaquin Phoenix	Best Actor

Click here to expand query

Acting categories [edit]

This list focuses on Puerto Rican-born actors and actresses as well as those with a strong Puerto Rican descent (i.e., Nuyoricans). As of 2007, there have only been three Academy Award winning actors born in Puerto Rico. However, others have been nominated for the awards.

Actor	Year	Award	Film	Result
José Ferrer	1948	Best Supporting Actor	*Joan of Arc*	Nominated
José Ferrer	1950	Best Actor	*Cyrano de Bergerac*	Won
José Ferrer	1952	Best Actor	*Moulin Rouge*	Nominated
Rita Moreno	1961	Best Supporting Actress	*West Side Story*	Won
Rosie Perez	1993	Best Supporting Actress	*Fearless*	Nominated
Benicio del Toro	2000	Best Supporting Actor	*Traffic*	Won
Joaquin Phoenix	2000	Best Supporting Actor	*Gladiator*	Nominated
Benicio del Toro	2003	Best Supporting Actor	*21 Grams*	Nominated
Joaquin Phoenix	2005	Best Actor	*Walk the Line*	Nominated

Benicio del Toro accepting the Oscar in 2000.

Figure 8.2: A rewriting returned for the query `select actor, award from movie` and its corresponding result is shown on the left. An excerpt of the Wikipedia page from which the result was extracted is displayed on the right.

#	Query
1.	city [state = UT]
2.	city [zipcode > 53701]
3.	zipcode [city = madison]
4.	product [company = ibm]
5.	restaurant telephone_ [city = malibu]
6.	company location ticker
7.	country country_code monetary_unit

Table 8.1: Example of Arbitrary Queries

relations that satisfy the predicates are selected.

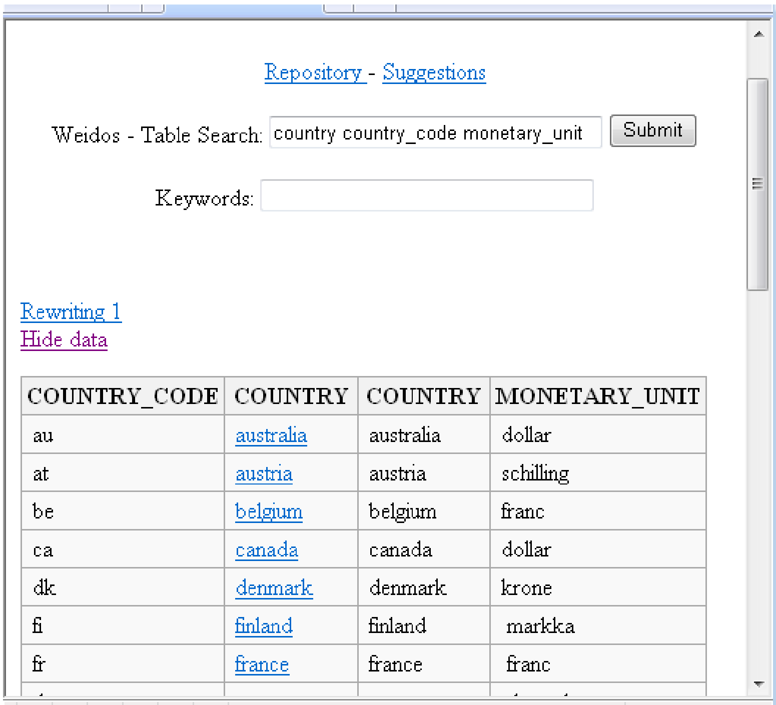
Queries six and seven are answered using multi-relation rewritings, having 'company' and 'country' as join attributes, respectively. Some of the results obtained for query 7 are shown in Figure 8.3.

It is important to notice that the number of relations from the WT10G data set that belong to the same domain is very small. Despite the size of the collection, there are only few relations that contain the same or similar schema. We did try to perform scalability experiments with this collection, but the limited amount of information that overlaps did not give us enough data to draw conclusions. However, we do expect that scalability results using larger overlapping schemas would be similar to the ones found for the movie data set.

8.3 Additional Features

The Eidos architecture - along with the information kept in the proposed indexes - enables interesting features to be implemented as part of the search engine. This section explores some of these possibilities. The presented features were added as part of the SQL search engine, but it could be implemented for join-less queries as well.

Figure 8.4 shows the query expansion feature, which allows the user to discover additional information that is available in the retrieved relations. If the user clicks on the link 'Click here to expand query' below the rewriting, a new answer is computed, composed by all attributes from the indexed relations. This feature allows the user



Repository - Suggestions

Weidos - Table Search: country country_code monetary_unit

Keywords:

Rewriting 1
Hide data

COUNTRY_CODE	COUNTRY	COUNTRY	MONETARY_UNIT
au	australia	australia	dollar
at	austria	austria	schilling
be	belgium	belgium	franc
ca	canada	canada	dollar
dk	denmark	denmark	krone
fi	finland	finland	markka
fr	france	france	franc

Figure 8.3: Some results for the query 'country country_code monetary_unit'

to quickly discover/visualize complementary information related to the query.

Additionally, next to each attribute‘s header, a drop-down list is provided that enables users to refine a query by switching a returned attribute to any other attribute that is available in the indexed relation. The Figure 8.4 illustrates what happens if the user switches the attribute *gross* for the attribute *studio*.

Besides structured queries, the search engine also supports keyword-based queries. Keyword-based queries can be used alone or in conjunction with structured queries. If the user issues a keyword-based query, the computed rewritings access pages that contain the specified keywords. Each rewriting refers to a particular HTML table and returns the contents of all columns of the table. To support keywords, we keep a full-text index built using Lucene[2].

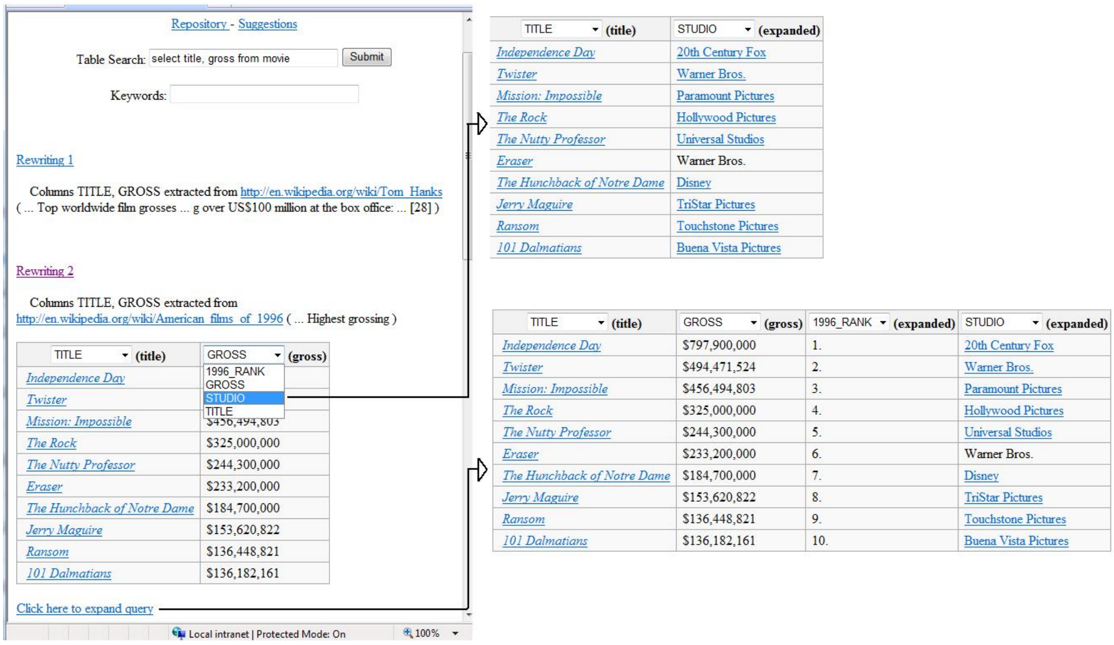
For instance, Figure 8.5 shows the first rewriting produced when the query 'groucho' is issued. This rewriting retrieves a source table in a page that contains information about the Marx Brothers. When a structured query and keyword-based query are used in the same request, rewritings are derived that:

- cover the attributes specified in the structured part of the query.
- come from a page from where the keywords can be found.

For example, suppose the user enters "select year from movie"as the structured query and 'Groucho' as the keyword query. Also suppose that all indexed relations cover the attribute *year*, but only one page contains the requested keyword. In this case, only the indexed relations that come from this specific page will generate rewritings.

The search engine also allows users to navigate through the schemas of the indexed relations. The 'Suggestions' link (located above the Table Search box) can

[2] http://lucene.apache.org



Repository - Suggestions

Table Search: select title, gross from movie Submit

Keywords:

Rewriting 1

Columns TITLE, GROSS extracted from http://en.wikipedia.org/wiki/Tom_Hanks (... Top worldwide film grosses ... g over US$100 million at the box office: ... [28])

Rewriting 2

Columns TITLE, GROSS extracted from http://en.wikipedia.org/wiki/American_films_of_1996 (... Highest grossing)

TITLE (title)	GROSS (gross)
Independence Day	
Twister	
Mission: Impossible	$456,494,803
The Rock	$325,000,000
The Nutty Professor	$244,300,000
Eraser	$233,200,000
The Hunchback of Notre Dame	$184,700,000
Jerry Maguire	$153,620,822
Ransom	$136,448,821
101 Dalmatians	$136,182,161

Click here to expand query

TITLE (title)	STUDIO (expanded)
Independence Day	20th Century Fox
Twister	Warner Bros.
Mission: Impossible	Paramount Pictures
The Rock	Hollywood Pictures
The Nutty Professor	Universal Studios
Eraser	Warner Bros.
The Hunchback of Notre Dame	Disney
Jerry Maguire	TriStar Pictures
Ransom	Touchstone Pictures
101 Dalmatians	Buena Vista Pictures

TITLE (title)	GROSS (gross)	1996_RANK (expanded)	STUDIO (expanded)
Independence Day	$797,900,000	1.	20th Century Fox
Twister	$494,471,524	2.	Warner Bros.
Mission: Impossible	$456,494,803	3.	Paramount Pictures
The Rock	$325,000,000	4.	Hollywood Pictures
The Nutty Professor	$244,300,000	5.	Universal Studios
Eraser	$233,200,000	6.	Warner Bros.
The Hunchback of Notre Dame	$184,700,000	7.	Disney
Jerry Maguire	$153,620,822	8.	TriStar Pictures
Ransom	$136,448,821	9.	Touchstone Pictures
101 Dalmatians	$136,182,161	10.	Buena Vista Pictures

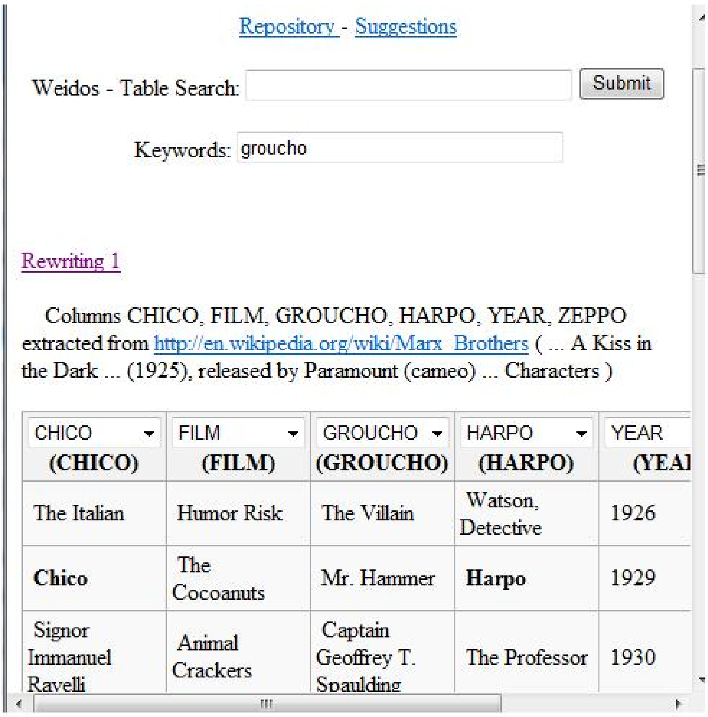
Figure 8.4: Users can expand/refine a rewriting in order to see other information related to the query.

be used to discover relation that are similar to the relation of the user query, for queries that contain a single relation. For instance, Figure 8.6 shows schemas that are related to the query "select title, actors from movies". Each entry in the result consists of the schema description and the number of relations that use this schema.

Schemas are ranked according to their relevance to the user query. Those that cover more attributes of the query are deemed more relevant. If two schemas cover the same number of attributes, the one that can be found in the higher number of relations is deemed more relevant. When the user clicks on a schema, the rewritings that use the selected schema become available.

Users can also browse the different schemas for all relations in the repository using the 'Repository' link (located above the Table Search box, Figure 8.4)). The schemas are ranked according to their frequency - the proportional number of source tables in the repository that use the schema. Similarly to the Suggestions link, when the user clicks on a schema, the rewritings that use the selected schema become available.

Precision could also be improved if additional information was collected from the Web pages other than the tables. Our Table Parser module only extracts information that appear inside HTML tables. For instance: in our collection, there are no tables that contain both the name of an artist and its role in different movies. However, we have noticed that the column `role` usually appear in Web Pages of a particular artist. Thus, even though the name of the artist is not directly represented in the table, it could be found in the surroundings, or sometimes even in the URL itself. Therefore, even if some information is not directly returned to the user, it is possible to take the initial answers as a starting point, and find out additional information by exploratory means, such as refining the query, checking the URL or going straight to the data source.



Weidos - Table Search:

Keywords: groucho

Columns CHICO, FILM, GROUCHO, HARPO, YEAR, ZEPPO extracted from http://en.wikipedia.org/wiki/Marx_Brothers (... A Kiss in the Dark ... (1925), released by Paramount (cameo) ... Characters)

CHICO	FILM	GROUCHO	HARPO	YEAR
(CHICO)	**(FILM)**	**(GROUCHO)**	**(HARPO)**	**(YEA**
The Italian	Humor Risk	The Villain	Watson, Detective	1926
Chico	The Cocoanuts	Mr. Hammer	**Harpo**	1929
Signor Immanuel Ravelli	Animal Crackers	Captain Geoffrey T. Spaulding	The Professor	1930

Figure 8.5: Keyword queries can be combined with structured queries to restrict the search

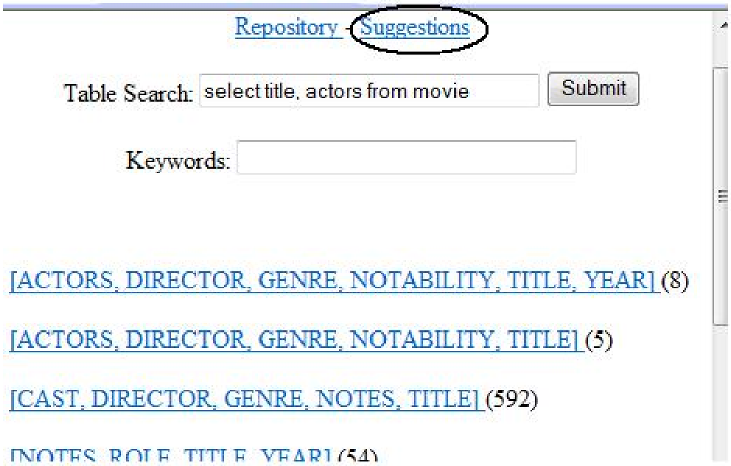


Table Search: select title, actors from movie

Keywords:

[ACTORS, DIRECTOR, GENRE, NOTABILITY, TITLE, YEAR] (8)

[ACTORS, DIRECTOR, GENRE, NOTABILITY, TITLE] (5)

[CAST, DIRECTOR, GENRE, NOTES, TITLE] (592)

[NOTES, ROLE, TITLE, YEAR] (54)

Figure 8.6: The suggestion feature shows schemas of indexed relations that are related to the relation of the query.

9 CONCLUSIONS

The main contribution of the thesis is an architecture for querying and correlating information from dataspaces in which pre-defined schemas or mappings are not required. Information from the sources are wrapped as relations and captured by specialized indexes that decouples the content as relational tuples composed by attributes and values. In querying time, the indexes are accessed in order to find rewritings - sources that best match a user‘s request.

So that user queries can be effectively and efficiently translated into queries over the underlying relations, different indexes and rewriting algorithms have been proposed. Simpler indexes, such as ATa, consume less memory but present a limited usability, restricting the user to search sources that contain the desired attributes. When the values of the relations are considered, more powerful indexes are created(ATaVa and AVaTa). These indexes allow rewriting checks to be performed over queries that contain selection and join predicates. Experiments show that the overhead of the most complete indexes is acceptable. Comparisons with related work demonstrate that the memory cost is similar, but the benefits are meaningful, considering the querying capabilities.

An important benefit from not having pre-defined mappings is that the cost of maintenance is reduced: new sources can be added to the system and queried without the overhead of creating new mappings (or updating a global schema). Of course this best-effort approach cannot give guarantees of coverage (i.e., that all answers are retrieved) or even that the returned answers are ‘correct’. Therefore, it is not a substitute for the traditional integration approaches, which are needed for application that require precise answers. However, it provides the means whereby users can more easily explore dataspaces, learn about different domains, identify relevant data sources and their associations, as a prelude to a more structured (e.g., mediator-based) integration systems.

Looking at the search engine perspective (instead of the integration system), our approach allows the user to easily access structured information sources. As already occurs in traditional search engines, false-positives are expected, so the answers cannot be taken from granted. When the results seem to be incorrect, it is up to the user to fine-tune the queries in order to improve the quality of the search. We understand that the trade-off is acceptable when considering a system that is able to reach a large volume of information.

To demonstrate the architecture’s flexibility, two query languages were tested. The first requires users to specify the joins, and the second is able to automatically identify joins between associated sources. The latter is easier to use, which makes of it the best choice considering the search engine scenario. On the other hand, the former allows the user to discover associations between related sources that would be hard to find automatically (in a scenario where foreign key relationships are not known), which makes of it the best choice for data integration/exploratory purposes.

The list below describes the contributions achieved by the thesis:

- No human interaction is needed to populate the indexes.
- Queries with selection predicated are allowed.
- Query answering can be done using indexed information.
- Associations between relations are discovered automatically.
- Detection of empty rewritings is possible.
- Algorithms to find rewritings scale well.
- Indexed information can be distributed in several nodes.
- The architecture can be easily extended.

Querying dataspaces is a challenging problem. As demonstrated, traditional integration systems are not able to handle the diversity found on the Web. Attempts have already been made to substitute the schema-first approaches, mainly using pay-as-you-go strategies or by indexing the structure sources. Following the indexing direction, we have taken another step towards a mechanism that automatically provides meaningful access to structured information sources on the Web, making it possible to seamlessly query and understand dataspaces and their connections. There are still several topics that need to be addressed in future works. In what follows we present a comprehensive list with the most relevant issues that remain open.

Ranking of the Rewritings An user query may return a reasonable number of rewritings. Therefore, there is a need to rank the rewriting so that the most relevant results appear first. The number of attributes covered and the estimated number of tuples returned are possible features that may be used, and that are easily extracted from the indexes.

Another possible heuristic involves analyzing the amount of joins among relations from different locations. The higher this number, the less relevant is the rewriting. This heuristic is based on the fact that different locations (Web Servers, databases, ...) may be inconsistent with respect to each other, and joining them may bring fewer tuples or even incorrect ones. The location itself is likely to be an useful ranking information. For example, if the user is searching for restaurants to go at night, the most relevant results are those that indicates near-by options.

Match by Similarity Currently, the attributes of a user query are matched to indexed attributes with the same name. Schema matching techniques (HE; CHANG, 2003; RAHM; BERNSTEIN, 2001) can be used to allow attributes to match even if their names are not the same (for example using name and data type similarity). Besides, similarity techniques are also useful to match tuples with selection predicates that contain similar content. The ranking mechanism suggested above can be adapted to take the match score into consideration.

Associations Discovery Similarly to the previous topic, associations are automatically found between relations that cover the same attribute, and only when the attributes contain the exact same content in at least one tuple of the associated relations. Schema matching techniques also serve to allow the join attribute to have similar names in different relations (not equal). Moreover, tuples with similar content on determined attributes can be used to determine which relations have overlapping information, and therefore should be associated. The similarity can be used along with the attribute‘s selectivity in order to determine the ranking of the rewritings found.

Join Attribute Discovery If one or more attributes in a user query span multiple relations, the implemented algorithms derive rewritings that join these relations. Observe that this only works if the original query contains a "good"join attribute. One possibility to overcome this limitation is by adding a preprocessing step into the query answering that tries to find the join attributes even if they are not declared in the user query. Schema auto-completion techniques can be explored in order to achieve this goal.

Definition of Evaluation Measures The querying mechanism proposed is based on inferences on how the query relates to the sources. This may lead to false positives rewritings (in the case of homonyms) or false negatives rewritings (in the case of synonyms). Thus, there is a need to evaluate the quality of the answers in order to determine the mechanism efficiency. This becomes more crucial if matches are to be found by similarity, as pointed earlier.

The precision and recall metrics are traditional approaches used to evaluate the quality of information retrieval systems (BAEZA-YATES; RIBEIRO-NETO, 1999). Generally speaking, these metrics are computed based on the number of correct answers and the number of incorrect answers of a query: the precision measures how many of the returned answers are correct, while the recall measures how many of the correct answers were returned.

In the context of our work, results can be incomplete (when not all query attributes are found. Therefore, a rewriting might not be correct neither incorrect, but something in between. It is important to adapt/complement the precision and recall metrics so that more consistent evaluations are performed.

Query Guidance and User Feedback Although users tend to refine queries in order to achieve better results, we believe that new mechanisms are needed to guide the query formulation, such as query wizards or navigation through trees of connected data sources. "Did you mean"features, where the query is fixed by adapting terms that might be incorrect also need to be investigated. As a proof of concept, we have implemented a mechanism that suggest new queries based on the schema of the original query.

Another interesting possibility is to use pay-as-you-go strategies by leveraging user feedback. To test the benefits of user feedback, our prototype already allows the user to interact with the query results by replacing the incorrect attributes with other attributes available in the data source. This feature not only brings immediate value to the user, but it naturally shows that it is possible to map the user interactions to a knowledge base, and therefore improve the search capabilities in the long-term. Other forms of interactions can be easily mapped, such as which rewritings were actually opened and the time the user spent in each opened rewriting.

Index Optimizations The experiments showed that the memory cost of the index is acceptable, considering the alternative solution found in the literature. Nevertheless, optimizations may still be able to reduce this cost, exploring existing techniques such as index summarization and canonization.

An additional idea to optimize the index is to learn from queries history. Instead of indexing all columns of a relation, the index can store only columns that appear more often in the queries history. Similarly, the values level can be indexed only for attributes that appear in selection predicates of past queries.

Building of Wrappers and Extractors To evaluate the Eidos architecture, a simple HTML table wrapper was implemented. The wrapper is able to recognize tables built with the proper HTML constructs, and only when the cells are well formed. Adaptations of the wrapper are necessary in order to reach more diverse tables. It is also important to develop new wrappers for other kinds of data sources, such as XML. Another good and vast source of information is the deep Web. Existing techniques for finding and surfacing hidden databases on the Web can be explored to amplify the range of the search engine (BARBOSA; FREIRE, 2007a,b; BARBOSA; FREIRE; SILVA, 2007).

Source Selection Control Our rewriting mechanism selects data sources that best match the attributes of a query. It is important to notice that other information related to the sources (source descriptions) can improve the selection, such as locality and capabilities. The Eidos architecture can be extended in different ways so this control can be implemented, mainly by adapting the proposed algorithms or by adding a new component that rearrange/remove the computed rewritings. Another useful control is to distributed the access to the data sources, to prevent communications bottlenecks cause when the same location is largely visited.

Functional Dependencies The usage of functional dependencies for query rewriting (BAI; HONG; MCTEAR, 2003) were already addressed in the literature. However, our approach makes no assumptions about how attributes of a wrapped relation (or even of the query) relate to each other. There is no knowledge available that informs which are the primary keys and which attributes functionally depend on others.

The availability of this information might be useful to perform query containment test. In this case, considering the matches are correct, it would be possible to determine if the answers provided by a rewriting are indeed sound. To support the test, it becomes necessary to infer the functional dependencies of the wrapped relations and the queries (MARCHI; LOPES; PETIT, 2002; BAUCKMANN; LESER; NAUMANN, 2006). Additionally, the query language can be extended so the user can declare useful information, such as which of the requested attributes correspond to a primary key.

Another research area involves exploring the functional dependencies to apply query minimization techniques, which reduces the size of the rewritings and consequently the time required to process them.

Peer-to-Peer Adaptations The proposed architecture considers a centralized solution that contains the index and the components required to compute the rewritings. The only distributed component is the publisher, responsible for providing access to data that is not reached by the crawler.

One idea worth investigation is the usage of a peer-to-peer solution that distributes the rewriting components among user nodes. This approach would alleviate the processing on the Eidos server, which would be responsible solely to keep the indexed information. Several P2P extensions can be built over this idea, including private networks, where nodes publish information that are accessible only by authorized users.

The following items are published material, which we list as part of the contributions of the thesis, and as additional information regarding the ideas described along the text. Part of the work was done at the University of Utah, with the collaboration of Professor Juliana Freire.

EDBT 2010 Conference paper that details the architectural components required to automatically discover associations among data sources (MERGEN; FREIRE; HEUSER, 2010a).

IJMSO 2010 Journal paper that further details the components introduced in the RED 2008 paper (MERGEN; FREIRE; HEUSER, 2010b).

SBBD 2008 Demo paper that describes the prototype of a search engine that provides access to structured information sources on the Web (MERGEN; FREIRE; HEUSER, 2008a).

RED 2008 Conference paper that details the architectural components required to perform queries that declare the join conditions (MERGEN; FREIRE; HEUSER, 2008b).

REFERENCES

ABITEBOUL, S.; DUSCHKA, O. M. Complexity of answering queries using materialized views. In: ACM SIGACT-SIGMOD-SIGART SYMPOSIUM ON PRINCIPLES OF DATABASE SYSTEMS, 1998, New York, NY, USA. **Proceedings...** ACM, 1998. p.254–263. (PODS '98).

AGRAWAL, R. et al. The Claremont report on database research. **SIGMOD Record**, New York, NY, USA, v.37, p.9–19, September 2008.

BAEZA-YATES, R. A.; RIBEIRO-NETO, B. **Modern Information Retrieval**. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1999.

BAI, Q.; HONG, J.; MCTEAR, M. F. Query rewriting using views in the presence of inclusion dependencies. In: ACM INTERNATIONAL WORKSHOP ON WEB INFORMATION AND DATA MANAGEMENT, 5., 2003, New York, NY, USA. **Proceedings...** ACM, 2003. p.134–138. (WIDM '03).

BARBOSA, L.; FREIRE, J. An adaptive crawler for locating hidden-Web entry points. In: WORLD WIDE WEB, 16., 2007, New York, NY, USA. **Proceedings...** ACM, 2007. p.441–450. (WWW '07).

BARBOSA, L.; FREIRE, J. Combining classifiers to identify online databases. In: WORLD WIDE WEB, 16., 2007, New York, NY, USA. **Proceedings...** ACM, 2007. p.431–440. (WWW '07).

BARBOSA, L.; FREIRE, J.; SILVA, A. Organizing Hidden-Web Databases by Clustering Visible Web Documents. **International Conference on Data Engineering**, Los Alamitos, CA, USA, v.0, p.326–335, 2007.

BAUCKMANN, J.; LESER, U.; NAUMANN, F. Efficiently Computing Inclusion Dependencies for Schema Discovery. In: INTERNATIONAL CONFERENCE ON DATA ENGINEERING WORKSHOPS, 22., 2006, Washington, DC, USA. **Proceedings...** IEEE Computer Society, 2006. p.2–. (ICDEW '06).

BILKE, A.; NAUMANN, F. Schema Matching Using Duplicates. In: INTERNATIONAL CONFERENCE ON DATA ENGINEERING, 21., 2005, Washington, DC, USA. **Proceedings...** IEEE Computer Society, 2005. p.69–80. (ICDE '05).

BOYD, M. et al. AutoMed: a bav data integration system for heterogeneous data sources. In: INTERNATIONAL CONFERENCE ON ADVANCED INFORMATION SYSTEMS ENGINEERING, 16., 2004. **Proceedings...** Springer-Verlag, 2004. p.82–97. (CAISE '04).

BRIN, S.; PAGE, L. The anatomy of a large-scale hypertextual Web search engine. In: WORLD WIDE WEB 7, 1998, Amsterdam, The Netherlands, The Netherlands. **Proceedings...** Elsevier Science Publishers B. V., 1998. p.107–117. (WWW7).

CAFARELLA, M. J. et al. WebTables: exploring the power of tables on the web. **VLDB Endowment**, [S.l.], v.1, p.538–549, August 2008.

CALÌ, A.; CALVANESE, D.; GIACOMO, G. D.; LENZERINI, M. Data Integration under Integrity Constraints. In: INTERNATIONAL CONFERENCE ON ADVANCED INFORMATION SYSTEMS ENGINEERING, 14., 2002, London, UK, UK. **Proceedings...** Springer-Verlag, 2002. p.262–279. (CAiSE '02).

CALVANESE, D. et al. Information Integration: conceptual modeling and reasoning support. In: IFCIS INTERNATIONAL CONFERENCE ON COOPERATIVE INFORMATION SYSTEMS, 3., 1998, Washington, DC, USA. **Proceedings...** IEEE Computer Society, 1998. p.280–291. (COOPIS '98).

CERI, S.; GOTTLOB, G.; TANCA, L. What You Always Wanted to Know About Datalog (And Never Dared to Ask). **IEEE Transactions on Knowledge and Data Engeneering**, Piscataway, NJ, USA, v.1, p.146–166, March 1989.

CHANDRA, A. K.; MERLIN, P. M. Optimal implementation of conjunctive queries in relational data bases. In: ACM SYMPOSIUM ON THEORY OF COMPUTING, 1977, New York, NY, USA. **Proceedings...** ACM, 1977. p.77–90. (STOC '77).

CHAWATHE, S. et al. The TSIMMIS Project: integration of heterogeneous information sources. In: MEETING OF THE INFORMATION PROCESSING SOCIETY OF JAPAN, 16., 1994, Tokyo, Japan. **Proceedings...** [S.l.: s.n.], 1994. p.7–18.

CHEKURI, C.; RAJARAMAN, A. Conjunctive Query Containment Revisited. In: INTERNATIONAL CONFERENCE ON DATABASE THEORY, 6., 1997, London, UK. **Proceedings...** Springer-Verlag, 1997. p.56–70.

CHURCH, K. W.; HANKS, P. Word association norms, mutual information, and lexicography. **Computational Linguistics**, Cambridge, MA, USA, v.16, p.22–29, March 1990.

COHEN, W. W.; HURST, M.; JENSEN, L. S. A flexible learning system for wrapping tables and lists in HTML documents. In: WORLD WIDE WEB, 11., 2002, New York, NY, USA. **Proceedings...** ACM, 2002. p.232–241. (WWW '02).

DALVI, N.; KUMAR, R.; SOLIMAN, M. Automatic wrappers for large scale web extraction. **VLDB Endowment**, [S.l.], v.4, p.219–230, January 2011.

DAS SARMA, A.; DONG, X.; HALEVY, A. Bootstrapping pay-as-you-go data integration systems. In: ACM SIGMOD INTERNATIONAL CONFERENCE ON MANAGEMENT OF DATA, 2008., 2008, New York, NY, USA. **Proceedings...** ACM, 2008. p.861–874. (SIGMOD '08).

DAVULCU, H.; FREIRE, J.; KIFER, M.; RAMAKRISHNAN, I. V. A layered architecture for querying dynamic Web content. In: ACM SIGMOD INTERNATIONAL CONFERENCE ON MANAGEMENT OF DATA, 1999., 1999, New York, NY, USA. **Proceedings...** ACM, 1999. p.491–502. (SIGMOD '99).

DO, H.-H.; RAHM, E. COMA: a system for flexible combination of schema matching approaches. In: VERY LARGE DATA BASES, 28., 2002. **Proceedings...** VLDB Endowment, 2002. p.610–621. (VLDB '02).

DOAN, A.; DOMINGOS, P.; HALEVY, A. Learning to Match the Schemas of Data Sources: a multistrategy approach. **Machine Learning**, Hingham, MA, USA, v.50, p.279–301, March 2003.

DONG, X.; HALEVY, A. Indexing dataspaces. In: ACM SIGMOD INTERNATIONAL CONFERENCE ON MANAGEMENT OF DATA, 2007., 2007, New York, NY, USA. **Proceedings...** ACM, 2007. p.43–54. (SIGMOD '07).

DUSCHKA, O. M.; GENESERETH, M. R. Query planning in infomaster. In: ACM SYMPOSIUM ON APPLIED COMPUTING, 1997., 1997, New York, NY, USA. **Proceedings...** ACM, 1997. p.109–111. (SAC '97).

EMBLEY, D. W. et al. Conceptual-model-based data extraction from multiple-record Web pages. **Data Knowlege Engeneering**, Amsterdam, The Netherlands, The Netherlands, v.31, p.227–251, November 1999.

ENSLOW, P. H. What is a Distributed Data Processing System? **Computer**, Los Alamitos, CA, USA, v.11, p.13–21, January 1978.

ETZIONI, O.; BANKO, M.; SODERLAND, S.; WELD, D. S. Open information extraction from the web. **Communications of the ACM**, New York, NY, USA, v.51, p.68–74, December 2008.

ETZIONI, O. et al. Web-scale information extraction in knowitall: (preliminary results). In: WORLD WIDE WEB, 13., 2004, New York, NY, USA. **Proceedings...** ACM, 2004. p.100–110. (WWW '04).

FAGIN, R.; MENDELZON, A. O.; ULLMAN, J. D. A simplied universal relation assumption and its properties. **ACM Transactions on Database Systems**, New York, NY, USA, v.7, p.343–360, September 1982.

FRANKLIN, M.; HALEVY, A.; MAIER, D. From databases to dataspaces: a new abstraction for information management. **SIGMOD Record**, New York, NY, USA, v.34, p.27–33, December 2005.

FRIEDMAN, M.; LEVY, A.; MILLSTEIN, T. Navigational plans for data integration. In: ARTIFICIAL INTELLIGENCE AND THE ELEVENTH INNOVATIVE APPLICATIONS OF ARTIFICIAL INTELLIGENCE CONFERENCE INNOVATIVE APPLICATIONS OF ARTIFICIAL INTELLIGENCE, 1999, Menlo Park, CA, USA. **Proceedings...** American Association for Artificial Intelligence, 1999. p.67–73. (AAAI '99/IAAI '99).

GOLDSTEIN, J.; LARSON, P.-A. Optimizing queries using materialized views: a practical, scalable solution. In: ACM SIGMOD INTERNATIONAL CONFERENCE ON MANAGEMENT OF DATA, 2001., 2001, New York, NY, USA. **Proceedings...** ACM, 2001. p.331–342. (SIGMOD '01).

GONZALEZ, H. et al. Google fusion tables: web-centered data management and collaboration. In: MANAGEMENT OF DATA, 2010., 2010, New York, NY, USA. **Proceedings...** ACM, 2010. p.1061–1066. (SIGMOD '10).

GRUSER, J. R.; RASCHID, L.; VIDAL, M. E.; BRIGHT, L. Wrapper Generation for Web Accessible Data Sources. In: IFCIS INTERNATIONAL CONFERENCE ON COOPERATIVE INFORMATION SYSTEMS, 1998. **Proceedings...** [S.l.: s.n.], 1998. p.14–23.

GUBANOV, M.; BERNSTEIN, P. Structural text search and comparison using automatically extracted schema. In: INTERNATIONAL WORKSHOP ON THE WEB AND DATABASESCOMMUN, 9., 2006. **Proceedings...** ACM, 2006.

HALEVY, A.; FRANKLIN, M.; MAIER, D. Principles of dataspace systems. In: ACM SIGMOD-SIGACT-SIGART SYMPOSIUM ON PRINCIPLES OF DATABASE SYSTEMS, 2006, New York, NY, USA. **Proceedings...** ACM, 2006. p.1–9. (PODS '06).

HALEVY, A. Y. Theory of answering queries using views. **SIGMOD Record**, New York, NY, USA, v.29, p.40–47, December 2000.

HALEVY, A. Y. Answering queries using views: a survey. **The VLDB Journal**, Secaucus, NJ, USA, v.10, p.270–294, December 2001.

HE, B.; CHANG, K. C.-C. Statistical schema matching across web query interfaces. In: ACM SIGMOD INTERNATIONAL CONFERENCE ON MANAGEMENT OF DATA, 2003., 2003, New York, NY, USA. **Proceedings...** ACM, 2003. p.217–228. (SIGMOD '03).

HERSH, W. Managing Gigabytes: compressing and indexing documents and images (second edition). **Information Retrieval**, Hingham, MA, USA, v.4, p.79–80, April 2001.

HRISTIDIS, V.; PAPAKONSTANTINOU, Y. Discover: keyword search in relational databases. In: VERY LARGE DATA BASES, 28., 2002. **Proceedings...** VLDB Endowment, 2002. p.670–681. (VLDB '02).

HULGERI, A.; NAKHE, C. Keyword Searching and Browsing in Databases using BANKS. In: INTERNATIONAL CONFERENCE ON DATA ENGINEERING, 18., 2002, Washington, DC, USA. **Proceedings...** IEEE Computer Society, 2002. p.431–. (ICDE '02).

IMIELINSKI, T.; LIPSKI JR., W. Incomplete Information in Relational Databases. **ACM Transactions on Database Systems**, New York, NY, USA, v.31, p.761–791, September 1984.

KACPRZYK, J.; ZIóLKOWSKI, A. Database queries with fuzzy linguistic quantifiers. **IEEE Transactions on Systems, Man and Cybernetics**, Piscataway, NJ, USA, v.16, p.474–479, May 1986.

KAPPEL, G.; KAPSAMMER, E.; RETSCHITZEGGER, W. Integrating XML and Relational Database Systems. **World Wide Web**, Hingham, MA, USA, v.7, p.343–384, December 2004.

KENT, W. The breakdown of the information model in multi-database systems. **SIGMOD Record**, New York, NY, USA, v.20, p.10–15, December 1991.

KHALID, B.; PATON, N. W.; EMBURY, S. M.; FERNANDES, A. A. A.; HEDELER, C. Feedback-based annotation, selection and refinement of schema mappings for dataspaces. In: INTERNATIONAL CONFERENCE ON EXTENDING DATABASE TECHNOLOGY, 13., 2010, New York, NY, USA. **Proceedings...** ACM, 2010. p.573–584. (EDBT '10).

KOSSMANN, D. The state of the art in distributed query processing. **ACM Computing Surveys**, New York, NY, USA, v.32, p.422–469, December 2000.

LAENDER, A. H. F.; RIBEIRO-NETO, B. A.; SILVA, A. S. da; TEIXEIRA, J. S. A brief survey of web data extraction tools. **SIGMOD Record**, New York, NY, USA, v.31, p.84–93, June 2002.

LENZERINI, M. Data integration: a theoretical perspective. In: ACM SIGMOD-SIGACT-SIGART SYMPOSIUM ON PRINCIPLES OF DATABASE SYSTEMS, 2002, New York, NY, USA. **Proceedings...** ACM, 2002. p.233–246. (PODS '02).

LEVY, A. Y.; RAJARAMAN, A.; ORDILLE, J. J. Querying Heterogeneous Information Sources Using Source Descriptions. In: INTERNATIONAL CONFERENCE ON VERY LARGE DATA BASES, 22., 1996, San Francisco, CA, USA. **Proceedings...** Morgan Kaufmann Publishers Inc., 1996. p.251–262. (VLDB '96).

MADHAVAN, J.; COHEN, S.; DONG, X. L.; HALEVY, A. Y.; JEFFERY, S. R.; KO, D.; YU, C. Web-Scale Data Integration: you can afford to pay as you go. In: CONFERENCE ON INNOVATIVE DATA SYSTEMS RESEARCH, 2007. **Proceedings...** www.crdrdb.org, 2007. p.342–350. (CIDR '07).

MADHAVAN, J.; HALEVY, A. Y. Composing mappings among data sources. In: VERY LARGE DATA BASES - VOLUME 29, 29., 2003. **Proceedings...** VLDB Endowment, 2003. p.572–583. (VLDB '2003).

MAIER, D.; ULLMAN, J. D.; VARDI, M. Y. On the foundations of the universal relation model. **ACM Transactions on Database Systems**, New York, NY, USA, v.9, p.283–308, June 1984.

MANOLESCU, I.; FLORESCU, D.; KOSSMANN, D. Answering XML Queries on Heterogeneous Data Sources. In: INTERNATIONAL CONFERENCE ON VERY LARGE DATA BASES, 27., 2001, San Francisco, CA, USA. **Proceedings...** Morgan Kaufmann Publishers Inc., 2001. p.241–250. (VLDB '01).

MARCHI, F. D.; LOPES, S.; PETIT, J.-M. Efficient Algorithms for Mining Inclusion Dependencies. In: INTERNATIONAL CONFERENCE ON EXTENDING DATABASE TECHNOLOGY: ADVANCES IN DATABASE TECHNOLOGY, 8., 2002, London, UK. **Proceedings...** Springer-Verlag, 2002. p.464–476. (EDBT '02).

MARIE, A.; GAL, A. Boosting Schema Matchers. In: OTM 2008 CONFEDERATED INTERNATIONAL CONFERENCES, 2008, Berlin, Heidelberg. **Proceedings...** Springer-Verlag, 2008. p.283–300. (OTM '08).

MCCANN, R. et al. Integrating Data from Disparate Sources: a mass collaboration approach. In: INTERNATIONAL CONFERENCE ON DATA ENGINEERING, 21., 2005, Washington, DC, USA. **Proceedings...** IEEE Computer Society, 2005. p.487–488. (ICDE '05).

MERGEN, S.; FREIRE, J.; HEUSER, C. A. Mesa: a search engine for querying web tables. In: SESSãO DE DEMOS DO SIMPOSIO BRASILEIRO DE BANCO DE DADOS, 2008. **Proceedings...** [S.l.: s.n.], 2008. (SBBD '08).

MERGEN, S.; FREIRE, J.; HEUSER, C. A. Querying structured information sources on the web. In: INTERNATIONAL CONFERENCE ON INFORMATION INTEGRATION AND WEB-BASED APPLICATIONS & SERVICES, 10., 2008, New York, NY, USA. **Proceedings...** ACM, 2008. p.470–476. (iiWAS '08).

MERGEN, S.; FREIRE, J.; HEUSER, C. A. Indexing relations on the web. In: INTERNATIONAL CONFERENCE ON EXTENDING DATABASE TECHNOLOGY, 13., 2010, New York, NY, USA. **Proceedings...** ACM, 2010. p.430–440. (EDBT '10).

MERGEN, S.; FREIRE, J.; HEUSER, C. A. Querying structured information sources on the web. In: INTERNATIONAL JOURNAL OF METADATA, SEMANTICS AND ONTOLOGIES, 2010. **Proceedings...** [S.l.: s.n.], 2010. p.208–221. (IJMSO '10, v.5).

MILLER, R. J.; HAAS, L. M.; HERNáNDEZ, M. A. Schema Mapping as Query Discovery. In: INTERNATIONAL CONFERENCE ON VERY LARGE DATA BASES, 26., 2000, San Francisco, CA, USA. **Proceedings...** Morgan Kaufmann Publishers Inc., 2000. p.77–88. (VLDB '00).

NOY, N. F. Semantic integration: a survey of ontology-based approaches. **SIGMOD Record**, New York, NY, USA, v.33, p.65–70, December 2004.

OOI, B. C. et al. PeerDB: peering into personal databases. In: ACM SIGMOD INTERNATIONAL CONFERENCE ON MANAGEMENT OF DATA, 2003., 2003, New York, NY, USA. **Proceedings...** ACM, 2003. p.659–659. (SIGMOD '03).

PACITTI, E.; SIMON, E. Update propagation strategies to improve freshness in lazy master replicated databases. **The VLDB Journal**, Secaucus, NJ, USA, v.8, p.305–318, February 2000.

POTTINGER, R.; HALEVY, A. MiniCon: a scalable algorithm for answering queries using views. **The VLDB Journal**, Secaucus, NJ, USA, v.10, p.182–198, September 2001.

RAHM, E.; BERNSTEIN, P. A. A survey of approaches to automatic schema matching. **The VLDB Journal**, Secaucus, NJ, USA, v.10, p.334–350, December 2001.

SHETH, A. P.; LARSON, J. A. Federated database systems for managing distributed, heterogeneous, and autonomous databases. **ACM Computing Surveys**, New York, NY, USA, v.22, p.183–236, September 1990.

SICILIA, M. A. On the use of existing upper ontologies as a metadata integration mechanism. In: DUBLIN CORE AND METADATA APPLICATIONS: VOCABULARIES IN PRACTICE, 2005., 2005. **Proceedings. . .** Dublin Core Metadata Initiative, 2005. p.18:1–18:5.

TALUKDAR, P. P.; IVES, Z. G.; PEREIRA, F. Automatically incorporating new sources in keyword search-based data integration. In: MANAGEMENT OF DATA, 2010., 2010, New York, NY, USA. **Proceedings. . .** ACM, 2010. p.387–398. (SIGMOD '10).

TATARINOV, I. et al. The Piazza peer data management project. **SIGMOD Record**, New York, NY, USA, v.32, p.47–52, September 2003.

TSATALOS, O. G.; SOLOMON, M. H.; IOANNIDIS, Y. E. The GMAP: a versatile tool for physical data independence. **The VLDB Journal**, Secaucus, NJ, USA, v.5, p.101–118, April 1996.

TURNEY, P. D. Mining the Web for Synonyms: pmi-ir versus lsa on toefl. In: EUROPEAN CONFERENCE ON MACHINE LEARNING, 12., 2001, London, UK. **Proceedings. . .** Springer-Verlag, 2001. p.491–502. (EMCL '01).

ULLMAN, J. D. Information Integration Using Logical Views. In: INTERNATIONAL CONFERENCE ON DATABASE THEORY, 6., 1997, London, UK. **Proceedings. . .** Springer-Verlag, 1997. p.19–40.

VAZ SALLES, M. A. et al. iTrails: pay-as-you-go information integration in dataspaces. In: VERY LARGE DATA BASES, 33., 2007. **Proceedings. . .** VLDB Endowment, 2007. p.663–674. (VLDB '07).

WIEDERHOLD, G. Mediators in the Architecture of Future Information Systems. In: HUHNS, M. N.; SINGH, M. P. (Ed.). **Readings in Agents**. San Francisco, CA, USA: Morgan Kaufmann, 1997. p.185–196.

XU, L.; EMBLEY, D. W. Combining the Best of Global-as-View and Local-as-View for Data Integration. In: INFORMATION SYSTEMS TECHNOLOGIES AND ITS APPLICATIONS, 2004. **Proceedings. . .** [S.l.: s.n.], 2004. p.123–136. (ISTA '04).

YAN, L. L.; MILLER, R. J.; HAAS, L. M.; FAGIN, R. Data-driven understanding and refinement of schema mappings. In: ACM SIGMOD INTERNATIONAL CONFERENCE ON MANAGEMENT OF DATA, 2001., 2001, New York, NY, USA. **Proceedings. . .** ACM, 2001. p.485–496. (SIGMOD '01).

APPENDIX A RESUMO DA TESE

O volume de dados estruturados disponibilizados na *Web* cresceu consideravelmente no passado recente, como os dados produzidos por *Web Services* e bases de dados online que publicam conteúdo na forma de tabelas HTML. Além disso, existem muitas fontes de dados distribuídas relativas ao mesmo domínio, e que frequentemente possuem informações complementares sobre objetos comuns.

A literatura se refere a essas fontes *Web* estruturadas como dataspaces. Um modo simples de definir um *dataspace* é um conjunto de fontes de informação estruturadas e distribuídas que servem a um propósito específico e que contém conexões (implícitas ou explícitas) entre elas.

Contrastando com documentos desestruturados, a presença de estrutura permite que consultas ricas sejam efetuadas em *dataspaces.* Isso cria novas oportunidades para mineração, correlação e integração de um número sem precedentes de fontes de dados distintas.

No entanto, as abordagens existentes para integração de dados apresentam limitações no tocante aos *dataspaces.* Considere, por exemplo, os mediadores. Estes sistemas definem um esquema global e mapeamentos pré-definidos que conectam este esquema aos esquemas locais. Com base nestes mapeamentos, consultas sobre o esquema global são traduzidas em consultas sobre as fontes de informação.

Naturalmente, não é viável criar e manter um sistema desse tipo para milhares de fontes. Na *Web*, novas fontes surgem a todo instante, e fontes existentes são modificadas. Assim, estar atento a essas mudanças e atualizar mapeamentos é um processo que tem um custo proibitivo. Além do mais, é improvável que um único esquema global seja suficiente para satisfazer as necessidades de todo perfil de usuários existente.

Em uma direção ortogonal, motores de busca também estão sendo considerados para o ambiente de *dataspaces.* Motores de busca são bastante usados para realizar pesquisas por palavra-chave, onde informações são modeladas como um conjunto de termos e pesquisadas através de interfaces de consulta simples. Dadas as devidas adaptações, motores de busca podem ser usados para lidar com fontes estruturadas. Até o momento, as pesquisas neste campo evoluíram a ponto de reduzir os custos relacionados à consulta. No entanto, a estrutura dos dados ainda não é explorada de forma adequada. Por exemplo, a expressividade das consultas é limitada, e tende a seguir o modelo baseado em palavras-chave, onde as consultas são simples coleções de termos.

Quando o interesse recai sobre *dataspaces*, motores de busca especializados são necessários para lidar com estrutura. Até o momento, os trabalhos relacionados conseguiram utilizar motores de busca como uma forma de reduzir o custo de manutenção, o que é um dos maiores problemas quando se lida com um grande número de fontes. Entretanto, nenhum dos trabalhos usa a estrutura de forma significativa. A expressividade das consultas é limitada e tende a seguir o modelo baseado em palavras-chave, onde as consultas são simples coleções de termos.

Esta tese propoe uma nova abordagem de integração projetada para atender a natureza escalável e dinâmica das fontes de dados estruturadas na *Web*. Nosso objetivo é prover aos usuários a habilidade de consultar e integrar *on the fly* um grande número de fontes de dados estruturadas disponíveis na *Web*, mesclando a expressividade das consultas dos sistemas de integração de dados com a flexibilidade dos sistemas de integração de *dataspaces*.

Dessa união, três características importantes se destacam:

- Não existe esquema global: Ao invés de submeter consultas em um esquema global, o usuário formula consultas com base no seu conhecimento do domínio e naquilo que ele espera encontrar. As consultas podem ser refinadas a medida que o usuário explora e aprende mais a respeito das fontes de informação.

- Mapeamentos são derivados automaticamente: Ao invés de exigir mapeamentos pré-definidos, as consultas de um usuário são reescritas em consultas sobre as fontes com base nas correspondências identificadas entre os atributos da consulta e os atributos das fontes.

- Consultas são estruturadas: As consultas são mais do que simples listas de palavras-chave. Explorando a estrutura das fontes, nossa abordagem permite que os usuários submetam consultas expressivas e consequentemente obtenham respostas de maior qualidade.

Para lidar com essas questões da forma adequada, construímos a arquitetura Eidos. Uma das premissas da arquitetura é tratar cada fonte de dados individual como uma relação. Para apoiar essa idéia, nosso modelo é baseado em três entidades interconectadas: relações, atributos e valores. Os relacionamentos entre essas entidades são capturados em índices especializados.

A arquitetura é dividida em três camadas. A camada de dados intermedia a comunicação entre as fontes e os demais componentes da arquitetura. A camada de mediador contém componentes responsáveis pela integração propriamente dita. Essa camada contém uma subcamada de reescrita, cujos componentes são dedicados ao processo de reescrita.

A tese está focada na camada de reescrita. Essa camada corresponde ao núcleo da arquitetura, uma vez que ela contém os componentes que formam o mecanismo de reescrita. Esses componentes são isolados dos demais componentes, o que permite com que a estratégia de reescrita mude sem que o restante da arquitetura precise ser alterado.

A camada de reescrita é composta por três componentes. O Indexer é ativado durante a indexação. Ela recebe informações sobre as fontes e as armazena no índice. Os outros dois componentes são ativados durante a consulta. O Query parser recebe uma consulta estruturada e a transforma no formato adequado. Dada uma consulta transformada, o Rewriter deve acessar os índices para descobrir diferentes formas de responder a consulta do usuário utilizando as fontes de dados.

Cada forma de resposta diferente (que utilize um conjunto de fontes específicas) é chamada de reescrita. Nosso objetivo é atingir o maior número de reescritas possíveis. Uma possibilidade é entregar ao usuário a união de todas as reescritas existentes. No entanto, decidimos apresentar ao usuário cada reescrita de forma isolada. Essa decisão foi motivada pelos seguintes fatores:

1. Dependendo do número de fontes, o custo para computar todas reescritas pode ser proibitivo. Além disso, o usuário normalmente foca nos primeiros resultados apenas.

2. É mais fácil relacionar tuplas de resposta com suas respectivas fontes de origem. Saber a proveniência das tuplas ajuda a identificar as melhores fontes de informação.

3. É possível associar uma reescrita a tuplas entendidas como erradas pelo usuário e ignorar todas tuplas que vêm dessa reescrita.

4. É possível conceber um mecanismo de *feedback*, onde o usuário pode informar que algumas (ou todas) informações das tuplas estão erradas. Com base nessa informação, o sistema pode adaptar a reescrita para aprimorar a eficácia.

Com base nessa arquitetura, foram propostas duas abordagens de consulta diferente. Cada uma delas deu origem a diferentes linguagens de consulta e componentes da camada de Reescrita.

Na primeira abordagem, o usuário tem a liberdade de criar consultas com junções. Para que isso seja possível, propomos os índices ATa, ATaVa e TAt. O primeiro é uma lista invertida de atributos, onde cada atributo aponta para as relações que o possuem. O segundo é uma extensão do primeiro que possui um nível adicional para armazenar informações sobre os valores encontrados dentro de cada par atributo/relação. O terceiro é uma lista invertida de relações, onde cada relação aponta para os atributos que ela possui.

De modo geral, o índice *ATa* permite encontrar relações que possuam os atributos presentes em uma consulta. O índice *ATaVa* também fornece essa informação, e ainda indica se a relação satisfaz os predicados de seleção da consulta, caso eles existam. O índice *TaT* é complementar aos outros dois, e serve para facilitar a navegação entre os elementos indexados.

Além dos índices foram criados algoritmos de busca de tabela, cujo objetivo é acessar os índices para encontrar as relações que forneçam respostas para a consulta do usuário. Os algoritmos propostos são Table Scan, Pivot Table Scan e Table Intersection.

Os experimentos mostram que *Pivot Table Scan* tem um desempenho bem superior aos outros dois. No entanto, ele pode deixar de encontrar todas as reescritas possíveis quando a consulta possui apenas atributos opcionais. Já os outros dois algoritmos sempre encontram todas as reescritas, mesmo quando a consulta possui apenas atributos opcionais. Desses dois, os experimentos mostram que o *Table Intersection* obtém um desempenho superior quando a consulta é composta por atributos pouco frequentes. Considerando que diversos domínios sejam indexados, é provável que nenhum atributo indexado seja muito frequente. Nessas circunstâncias, onde a consulta não possui atributos obrigatórios e todos os atributos são pouco frequentes, o *Table Intersection* é o mais recomendado.

Também foram propostos algoritmos que exploram o índice *AVaTa* de diversas formas, como para otimizar o processo de busca de tabela (para quando a consulta possuir predicados de seleção), para encontrar reescritas vazias e para realizar o processamento da consulta propriamente dito, sem precisar que as fontes de dados sejam acessadas.

Na segunda abordagem, o usuário não precisa informar critérios de junção na consulta. Cabe ao próprio componente de reescrita descobrir se existem reescritas que tragam o resultado fazendo a junção de dados de múltiplas fontes. Para que isso seja possível, propomos o índice AVaTa, complementar aos demais índices propostos. Esse índice é uma lista invertida de atributos, onde cada atributo aponta para os valores que ele possui. O índice ainda conta com um nível adicional para armazenar informações sobre as relações associadas com cada par atributo/valor.

O índice *AVaTa* só possui informações para atributos que apareçam em mais de uma relação, sendo que nessas relações deve existir pelo menos um valor em comum dentro desses atributos. De certo modo, essas informações indicam algumas possibilidades de junção entre fontes de dados diferentes. Os algoritmos de reescrita acessam esse índice para descobrir como fontes diferentes podem colaborar para responder a uma consulta do usuário.

Quanto aos algoritmos de reescrita para a segunda abordagem, foram realizados experimentos testando diferentes estratégias de combinação, onde o objetivo é encontrar combinações de relações que formem reescritas válidas. Para ser válida, estipulamos que a reescrita deve ser mínima e completa. Três estratégias foram medidas: Naïve, Stream-Driven e Template-Driven. Das três, a *Template-Driven* possui o melhor desempenho, só perdendo para a *Stream-Driven* quando o número de relações a ser combinadas for muito baixo.

Os experimentos também comparam os índices propostos na tese com os índices propostos em um trabalho relacionado. As comparações indicam que nossa abordagem é mais indicada quando o usuário tem interesse em pesquisar os atributos das fontes, enquanto o trabalho relacionado é mais indicado quando o usuário tem interesse em pesquisar as fontes usando palavras-chave. Como consequência disso, nossa abordagem é mais adequada para consultas com predicados de seleção. Ainda, experimentos medindo o consumo de memória mostram que os índices propostos na tese consomem menos memória que os índices do trabalho relacionado.

As duas abordagens de consulta foram implementadas em protótipos. Os índices foram alimentados com dados reais obtidos de páginas da *Wikipedia* e da coleção WT10G, que contem cerca de 1.5M. de páginas extraídas da *Web*. Acessando esses índices, os protótipos desenvolvidos permitiram que diversos tipos de consulta fossem respondidos.

Um importante benefício de não trabalhar com mapeamentos pré-definidos é que o custo de manutenção é reduzido: novas fontes de dados são adicionadas ao sistema e consultadas sem a sobrecarga da criação de novos mapeamentos (ou a atualização do esquema global). É importante salientar que essa abordagem *best-effort* não garante que todas respostas sejam recuperadas, e que as respostas recuperadas sejam corretas. Dessa forma, ela não substitui as abordagens de integração tradicionais, que são necessárias para aplicações que necessitem respostas precisas. No entanto, essa abordagem fornece os meios para que usuários facilmente explorem *dataspaces*, aprendam sobre domínios diferentes, e identifiquem fontes de dados relevantes e suas associações, como um prelúdio para sistemas de integração de dados mais estruturados.

Já do ponto de vista de um motor de busca, nossa abordagem permite que o usuário facilmente acesse fontes de dados estruturadas. Como já ocorre em motores de busca tradicionais, falsos-positivos são esperados, ou seja, não se pode assumir que as respostas sejam corretas. Quando as respostas parecem estar erradas, cabe ao usuário refinar a consulta de forma a aprimorar a qualidade da busca. Nós entendemos que esse custo-benefício é aceitável para um sistema apto a atingir um grande volume de informações.

A lista abaixo descreve as principais contribuições alcançadas na tese:

- Nenhuma interação humana é necessária para popular os índices.
- Consultas com predicados de seleção são suportadas.
- O processamento de consultas pode ser feito usando informações dos índices.
- Associações entre relações são descobertas automaticamente.
- A detecção de reescritas vazias é possível.
- Os algoritmos propostos para encontrar reescritas são escaláveis.
- As informações indexadas podem ser distribuídas.
- A arquitetura pode ser facilmente estendida.

A integração e posterior realização de consultas sobre *dataspaces* são problemas desafiadores. Como demonstrado, sistemas de integração tradicionais não lidam bem com a diversidade encontrada na *Web*. Tentativas já foram feitas para eliminar o custo de manutenção, como através de abordagens pay-as-you-go ou através de indexação. Seguindo o caminho da indexação, demos um importante passo rumo a um mecanismo que automaticamente disponibilize acesso à fontes de dados estruturadas na *Web*, tornando possível a realização de consultas transparentes e a compreensão dos *dataspaces* e suas conexões. Ainda existem diversos tópicos que precisam ser atacados em trabalhos futuros. A seguir destaco três que consideramos importantes:

Ranking das Reescritas Uma consulta pode retornar um número razoável de reescritas. Assim sendo, existe a necessidade de ordenar as reescritas para que os resultados mais relevantes apareçam primeiro. O número de atributos cobertos e o número estimado de tuplas retornadas são critérios que podem ser usados, e que são facilmente extraídos a partir dos índices.

Outro critério possível envolve a quantidade de junções entre relações que residem em diferentes fontes. Quanto maior o número, menos relevante é a reescrita. Essa heurística é baseada no fato de que diferentes fontes (servidores *Web*, bancos de dados) podem ser inconsistentes entre si, e junções entre eles podem trazer poucas tuplas ou tuplas incorretas. A localização propriamente dita é outra informação útil para *ranking*. Por exemplo, se o usuário está buscando por restaurantes, os resultados mais relevantes são aqueles que indicam opções geograficamente próximas.

Casamento por Similaridade No estado atual, os atributos de uma consulta são casados com atributos indexados que possuem o mesmo nome. Técnicas de casamento de esquema podem ser usadas para permitir que atributos casem mesmo que seus nomes sejam diferentes (por exemplo, usando similaridade de nome ou de tipo de dados). Técnicas de similaridade também são úteis para casar tuplas com predicados de seleção que contenham conteúdo similar. O mecanismo de *ranking* comentado acima pode ser adaptado para levar o escore de casamento em consideração.

Descoberta de Associações Associações são automaticamente identificadas entre relações que cobrem o mesmo atributo, e apenas quando os atributos contêm o mesmo conteúdo em pelo menos uma tupla das relações associadas. Técnicas de casamento de esquema podem ajudar a encontrar atributos de junção que tenham nomes diferentes. Ainda, a associação de relações pode ser computada com base em tuplas com conteúdo similar.