ABSTRACT

CHEN, DONGFENG. Information Integration: The Semantic-Model Approach. (Under the direction of Professor Rada Chirkova and Professor Fereidoon Sadri).

This dissertation describes a multiple-coordinator system for large-scale information integration and interoperability, presents algorithms for query processing and optimization based on *semantic-model* approach, and experimentally evaluates the algorithms on real-life and synthetic data. In addition to supporting gradual large-scale information integration and efficient inter-source processing, the semantic-model approach eliminates the need for mediation in deriving the global schema, thus addressing the main limitation of information-integration systems.

This dissertation focuses on performance-related characteristics of several alternative approaches proposed for efficient query processing in the semantic-model environment. The theoretical results and practical algorithms are of independent interest and can be used in any information-integration system that avoids loading all the data into a single repository.

Query-processing approaches proposed in this dissertation are applicable both to the original stored data and to materialized views, including restructured views, which are a framework for representing, e.g., the pivot operation available in many database-management systems. These approaches account for the practical issues of information overlap across data sources and of inter-source processing. While most of these algorithms are platform- and implementation-independent, XML-specific optimization techniques that allow for system-level tuning of query-processing performance are proposed as well. Finally, using real-life datasets and the implementation of an information-integration system shell, this dissertation provides experimental results that demonstrate that these algorithms are efficient and competitive in the information-integration setting.

Information Integration: The Semantic-Model Approach

by
Dongfeng Chen

A dissertation submitted to the Graduate Faculty of
North Carolina State University
in partial fullfillment of the
requirements for the Degree of
Doctor of Philosophy

Computer Science

Raleigh, North Carolina

2008

APPROVED BY:

Dr. Ting Yu	Dr. Rudra Dutta
Dr. Rada Chirkova Chair of Advisory Committee	Dr. Fereidoon Sadri

DEDICATION

To my parents
Rihong Chen and *Caiju Chen*
for their endless love and support.

BIOGRAPHY

Dongfeng Chen was born in Yongding, Fujian, China in 1979. He received his B.S. in Computer Science from Beijing Information & Technology Institute, Beijing, China in 2001. Then, he continued his study and graduated with a M.S. degree from the University of Science & Technology of China, Hefei, China in 2004. Since fall 2004, he has been a full-time Ph.D. Student at the Computer Science Department of North Carolina State University, Raleigh, North Carolina, United States.

ACKNOWLEDGMENTS

I am very lucky to have two great advisors, Dr. Rada Chirkova and Dr. Fereidoon Sadri, who have been giving their expert guidance throughout my research. Besides their instructions that lead me into the path of the academic world, I am also very grateful for their kindness, patience, financial support and encouragement.

The other two members in my advisory committee, Dr. Ting Yu and Dr. Rudra Dutta, have been the instructors of the most valuable courses I have taken in the graduate school. I also learned the most valuable theories for my research in these courses. I would like to thank them for their continuous assistance on applying the theories into this research.

I would thank all my friends in the Database Research Laboratory, for their help to my research, and the happy time spent together in the office: Gang Gou, Manik Chandrachud, Zohreh Asgharzadeh Talebi, and Maxim Kormilitsin. I would also thank my best friends: Huiying Zhang, Liming Pan, Wei Mu, and others.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

Introduction

In this chapter, we discuss the problem of information integration and previous works related to information integration (Section 1.1). We also outline our contributions to information integration and interoperability in Section 1.2 and present the structure of this dissertation in Section 1.3.

1.1 Information Integration

The need for decentralized data sharing arises naturally in a wide range of applications, including enterprise data management, scientific projects undertaken across universities or research labs, data sharing among governmental databases, and the World-Wide Web. The recent advent of XML as a standard for online data interchange holds much promise toward promoting interoperability and data integration. In addition the focus of information integration has shifted from small-scale integration to providing integration and interoperability among a large number of independent and autonomous information sources. Historically, research and practice in data sharing have focused on information-integration systems, which query distributed shared data through a single central point with a fixed *mediated* schema [69].

Information-integration projects have been a research and commercial success for applications requiring integration of relatively few data sources. At the same time, the need for the mediated schema is a major bottleneck in developing data-sharing products for many real-life applications [65]. Peer data-management systems (PDMS) (*e.g.*, see [34] and references therein) address this limitation by eliminating the need for a mediated schema

altogether. In PDMS, each (physical) peer uses its own schema of its stored data and typically interacts with one or more other peers using agreed-on *local* mappings between the respective schemas. This framework makes it easy for a peer to join or leave the peer system at any point in time (more details in Chapter 2).

PDMS mechanisms for querying the shared data must take into account compositions of the peer-to-peer mappings, which may lead to unsatisfactory query-processing costs in large-scale systems. In addition, in many practical applications — such as banking, large-scale collaboration among scientific projects undertaken across universities or research labs, data sharing among governmental databases and agencies, and medical information systems — several information sources may store fragments of the same kind of data (conceptually, of the same logical relation), such as information about employees or user accounts in individual bank branches. Coupled with the need for evaluating queries that involve joins of data stored in more than one data source *(inter-source processing),* such data configurations present a further complication in query processing in peer-to-peer systems.

In this dissertation we address these and other query-processing challenges in large-scale data-sharing systems, by developing algorithms for query evaluation and optimization in our *semantic-model* approach to information integration and interoperability. In the semantic-model (SM) approach, introduced in [46], information at each source is viewed as a collection of (logical) binary relations, which we call the *semantic-model view.* These relations are basically a decomposition of the information into its "atomic components". (For practical simplicity, we allow views that combine binary relations with the same key into a single relation.) The SM approach is consistent with approaches based on ontological modeling [55] in the Semantic-Web initiative, where applications are modeled by the relevant concepts and their properties (basically, by binary relationships), and the semantic-model view for a given source can be designed using an available ontology. To include a source in the integration effort, the source's owner provides mappings from the source's data to the SM view. A source can participate by providing mapping for as few as a single relation, and add more mappings if desired at will. Hence the system supports gradual (pay-as-you-go) integration, where new sources can join incrementally and with small overhead.

1.2 Contributions

In general, there could be a number of ways to process user queries, with widely different performances. No single query-processing strategy would be optimum for all queries and cases. Rather, an intelligent query-optimization approach would be to be able to choose from a number of alternatives. In this dissertation we focus on the problem of efficiency of query evaluation in information-integration systems, with the objective of developing query-processing strategies that are widely applicable and easy to implement in real-life applications. In our algorithms we take into account the following important features of today's data-sharing applications:

- *XML as representation for data sources:* The recent advent of XML as a standard for online data interchange (and perhaps even for data storage, see, e.g.,the datasets of [19, 63]) holds much promise toward promoting interoperability and data integration.
- *Overlapping information in data sources:* In many practical applications (e.g., banking or medical information systems) data sources may overlap on the data they store, such as information about user accounts in individual bank branches.
- *Inter-source processing:* Some applications (e.g., banking or data sharing among governmental agencies) require evaluation of queries that involve joins of data stored across data sources.

While our theoretical results and most of our proposed techniques are implementation-neutral and thus applicable in a variety of settings for information-integration systems, we are also introducing platform-specific approaches (such as our semantic optimization,[1] see Section 4.5) that allow for system-level tuning of query-processing performance.

To compare and rate the proposed approaches, we have obtained experimental results using our information-integration system shell [70] that incorporates an implementation of all the algorithms and optimizations proposed in this paper. This software system enables interaction between (i) data sources that store data using the XML data model, and (ii) relational mediators whose schemas conform to the Semantic Model. To include a data source in the integration effort, the source owner provides mappings from the source data to the semantic-model view. The semantic-model approach is consistent with recent work on large-scale information integration [27, 50], as well as with approaches based on

[1] Our proposed semantic-optimization techniques are also more widely applicable to general XQuery optimization.

ontological modeling [55] in the Semantic-Web initiative. For methods for building source-to-mediator mappings in this setting, we refer the reader to [22, 23, 60] and to references therein. Our system can also serve as a building block in a three-tiered architecture [70] for information integration. The purpose of the architecture is to decouple query processing and optimization into "intra-coordinator processing" and "inter-coordinator processing" (conceptually analogous to query processing in peer-to-peer systems), thus keeping the optimization choices relatively local and improving the overall query-processing performance of the system.

Our specific contributions are as follows:

- We propose query-processing algorithms for information integration; the algorithms do *not* involve building source-to-mediator mappings but they do need the mappings, and are platform- and implementation-independent. To the best of our knowledge, our methods are the first that account for information overlap and for inter-source processing.

- We present theoretical results that allow for further reduction of inter-source processing by using information about integrity constraints in the data sources.

- We develop a suite of algorithms for efficient query processing in presence of *materialized restructured views* [10] (as exemplified by, e.g., operation *unfold*, or *pivot* [48]); the algorithms are applicable to views materialized in both mediators and data sources.

- We propose XML-specific optimization techniques that allow for system-level tuning of query-processing performance.

- Using real-life datasets and our implementation of an information-integration system shell [70], we report experimental results that demonstrate that our algorithms are efficient and competitive.

1.3 Structure of the Dissertation

The thesis is organized as follows: In Chapter 2, we give a survey on the related work including semantic web, mediated schema, and peer data management systems (PDMS). Chapter 3 - Chapter 5 provide the Semantic-Model approach and discuss efficiency

issues and optimization opportunities. Chapter 6 introduces the preliminaries of restructured views and our experiments on restructured views in a single database management system. We also describe query optimization using restructured views in the distributed systems in Chapter 7, and conclude our work and discuss future directions in Chapter 8.

Chapter 2

Related Work

There has been an increased interest in information integration and interoperability and its applications in recent years. The degree of research activities and publications in information integration and related areas, such as schema matching, model management, and information exchange, have also increased substantially. In this chapter, we first review ontological modeling and semantic web tools (Section 2.1) which are used to implement data integration systems, and then a classic system with a fixed mediated schema (Section 2.2). We discuss Peer Data Management Systems (PDMS) and its revelent projects in Section 2.3. In the rest of this section we examine interoperability on XML data in Section 2.4.

2.1 Semantic Web

There has been a revitalization of ontological modeling as a result of the W3C Semantic Web initiative [62]. Some information integration systems have been proposed and implemented using ontological modeling concepts. For example, the ICS-FORTH Semantic Web Integration Middleware (SWIM) [11, 12] uses Semantic Web tools for integration. Main difference with our approach is in our use of database tools and concepts, and emphasis on query optimization, versus their reliance on Semantic Web tools, such as RDF and query language RQL [42]. Further, we pay special attention to queries that require data from multiple sources (inter-source processing), while this important issue has not been addressed in SWIM.

In a recent work on indexing large volumes of data [27], authors advocate a *triple* model as a global model for all data. Triples are closely related to RDF and ontologies,

and to our Semantic-Model view model, demonstrating the validity and naturalness of our approach for modeling the information contents of information sources.

2.2 Mediated Schema

Historically, significant research effort has been directed toward information-integration systems that query data sources through a single central point with a fixed *mediated* schema [69]. A common integration architecture is shown in Figure 2.1. Several sources are wrapped by software that translates between the source's local language, model, and concepts and the global concepts shared by some or all of the sources. System components, here called mediators, obtain information from one or more components below them, which may be wrapped sources or other mediators. Mediators also provide information to components above them and to external users of the system.

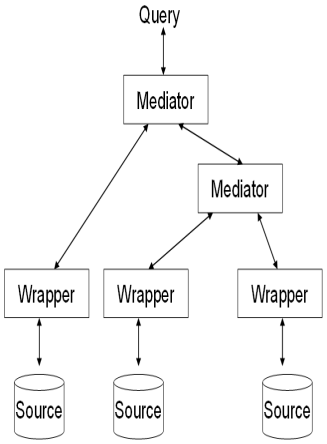

Figure 2.1: Mediator-based architecture.

In the mediator-based framework, the need to establish the mediated schema and translation rules, or *mappings,* between the data sources and the central mediator is a major bottleneck in integration efforts for real-life applications [65].

Some works from mediated schema focused on integrating schemas by defining a global schema in an expressive data model and defining mappings from local schemas to the global one, for instance, local-as-view (LAV) systems, where "local" refers to the local sources/databases, or global-as-view (GAV) systems, where "global" refers to the global (mediated) schema [31, 67, 45]. However, these works neglect the following situations: (i)

data sources may model data in heterogeneous ways, and (ii) sources may employ various terminology. Besides, designing the global schema typically leads to significant overhead in creating and maintaining data systems.

2.3 PDMS

The Piazza and related projects [32, 34, 35, 57, 66] cover various aspects of large-scale data integration, including: (1) Peer-based data management [35, 51, 65, 66], (2) Schema mapping [20, 23, 24, 26, 36, 49], and (3) Theoretical foundations, indexing, and access control [25, 28, 41]. The main idea behind data integration/interoperability in Piazza Peer Data Management System (PDMS) is that users provide mappings between pairs of information sources [35]. There is no need to provide mappings for all pairs. In fact, all that is needed is that the sources graph that represents available mappings be connected. As some peers in a PDMS may act as mediators *(coordinators)* with respect to other peers, a PDMS can be used as a basis for sharing data on the World-Wide Web, as in the Semantic-Web initiative [5]. See Figure 2.2 for an example of a PDMS. The key challenge in query answering in a PDMS is how to make use of the mappings to answer a query. A query should be rewritten using sources reachable through the transitive closure of all mappings. However, mappings are defined "directionally" with query expressions, and a given user query may have to be evaluated against the mapping in either the "forward" or "backward" direction. This means that PDMS's query answering algorithm performs query unfolding and query reformulation using views.

Mappings between any two sources can then be obtained by composing the pairwise mappings along a path connecting the two sources [51, 65]. However, this works well for sources belonging to the same application domain, with similar data. Otherwise the composition process will result in information loss.

As we mentioned in Chapter 1, the main disadvantage of a PDMS is that it doesn't deal with inter-source queries very well. Another advantage of our multiple-coordinator system is that the system is more resilient to erroneous mappings (only one source is affected by an erroneous mapping). In contrast, an erroneous mapping in Piazza affects all query processing that uses a path that includes the mapping.

The Clio [14, 54] and Hyperion [3, 37] projects have developed tools for automating

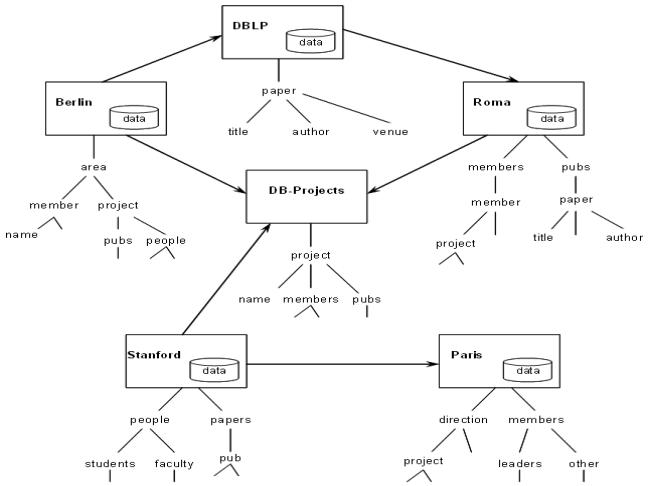

Figure 2.2: A PDMS for the database research domain. A fragment of each peer's XML schema is shown as a labeled tree (from [65]).

common data and structure management tasks underlying many data integration, translation, transformation, and evolution tasks. The thrust of these projects has been on supporting schema management, such as generating, matching, and mapping queries between schemas in multi-source and peer-to-peer systems [44, 53]. Semi-automatic techniques have been developed for general schema matching and for generating mappings between schemas [58, 72]. The projects introduce the *mapping table* approach to represent schema mappings, and discuss query processing in this environment [43, 44]. The architecture is similar to that of Piazza: A query is submitted at a peer, which passes it, possibly in translated form, to (some of) its acquaintances, which repeat this process.

2.4 Interoperability on XML Data

The authors of [46] present a lightweight infrastructure based on local semantic declarations for enabling interoperability across data sources. The main idea underlying this approach is based on the observation that ontology-specification frameworks such as RDF and OWL provide mechanisms for specifying metadata about a source: the metadata includes not only metadata such as the author and creation date, but also the semantic concepts present therein. The strength of this coordinator-based model is that it has inter-source query processing. However, its main disadvantage is that it doesn't scale very well, and the coordinator may become the bottleneck when people try to design large-scale data

sharing systems.

Our semantic-model approach extends the approach in the coordinator-based model by introducing multiple coordinators. It differs from data-warehousing approaches in that it does not generate a repository of all data, and hence is flexible and can dynamically accommodate sources and scale up to a large number of sources. It differs from information-integration systems that use a mediated schema in that it avoids the lengthy and error-prone process of schema mediation. Intuitively, our SM approach uses ready-made ontologies, or even a much simpler identification of binary relations to serve as an atomic decomposition of the application domain. Again, this allows incremental information integration and scalability. Finally, it differs significantly from peer data-management systems in its capabilities and efficiency in inter-source processing, and its ability to handle sources coming from different application domains.

Chapter 3

The Semantic-Model Approach

In this chapter we discuss the semantic-model approach to large-scale data integration and interoperability. Section 3.1 provides the multiple-coordinator and three-tiered hierarchical system based on the distributed system with only one coordinator proposed in [46]. In Section 3.2 we discuss the semantic-model approach to query processing and query optimization. Many examples are provided for this kind of three-tiered hierarchical distributed systems in Section 3.3. Finally we concentrate on the optimization challenges and opportunities in the semantic-model approach in Section 3.4.

3.1 System Architecture

The architecture of our system is shown in Figure 3.1. We introduce multiple coordinators (or "super peers") into the heterogeneous structure. The main advantage of this approach that we propose is to design large-scale data sharing system, without losing semantic interoperability among data sources. In general, we can have a network of communicating coordinators, where each coordinator is in charge of a set of sources and, possibly, other coordinators. In fact, a coordinator can be regarded as a source, or, more accurately, as a broker for the information under its oversight. An information source can also function as a coordinator. In the simplest form (see Figure 3.2), all information sources are connected to a single coordinator, which is in charge of coordinating query processing. A pure peer-to-peer system (see Figure 3.3) is a special case of a peer/coordinator system where every source is its own coordinator.

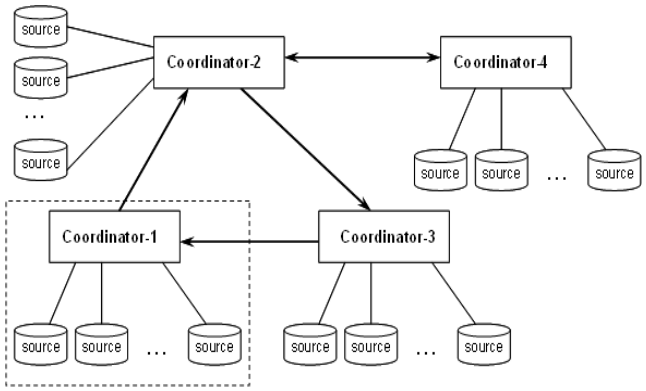

Figure 3.1: System architecture.

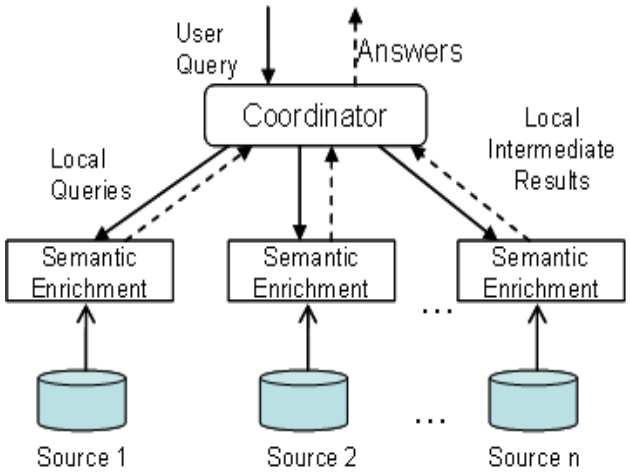

Figure 3.2: The simplest form.

3.2 Overview of the SM Approach

The *semantic-model* (SM) approach to information integration and interoperability was first introduced in [46]. In the SM approach, an information source joins an information-integration effort by providing a semantic-model view *(SM view)* of its information, as well as a mapping from its data to this model. The SM view for a source is a view of the information as a collection of binary relations, possibly based on an ontology for the source's application domain. These are basically a decomposition of the information into its "atomic components". (For practical simplicity, we allow views that combine binary relations with the same key into a single relation.)

The task of defining the view and providing data mappings is the responsibility of

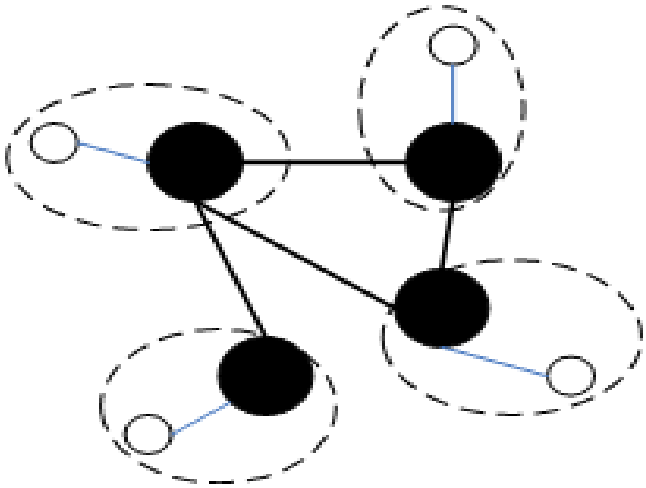

Figure 3.3: A pure P2P system.

the database administrator (DBA) and/or users of local sources. Tools, such as [52], can greatly simplify the task of defining mapping rules. An SM system contains many *sites*, where a site can be either an information source or a *coordinator* that oversees query decomposition and execution. A user query can be submitted at any site. It can be formulated in terms of the SM view, or in terms of the local (original) schema if submitted at an information source. Currently, the implementation of the SM system supports XML and relational sources, but the SM approach is applicable to any source type as long as mappings from the source data format (e.g., RDF, Microsoft Excel, etc.) to the SM view can be provided. The integration and interoperability system is responsible for processing the query in such a way that the answer corresponds to the answer of the query on the collection of all data in the participating information sources. There are many ways to achieve this goal, with widely varying performance characteristics. Query optimization, in particular, is critical to such an interoperability system. The SM approach is consistent with approaches based on ontological modeling [55], where applications are modeled by domain-specific concepts and their properties, which are, basically, binary relationships. The following examples (adopted from [46]) illustrate the SM approach.

Example 1 (The semantic model) *Consider a federation of catalog sales businesses. In this example we concentrate on their warehousing operations. A possible ontology for this application may use objects* (concepts) *such as* `item`*,* `warehouse`*,* `city`*,* `state`*, and relationships* (properties) *such as* `item-name`*,* `item-warehouse`*,* `warehouse-city`*, and*

`warehouse-state`.

The SM view consists of binary relations representing the relationships. Sources with heterogeneous models and schemas can model their warehousing operations using this SM view. For example, the DTDs of two XML sources are shown below [1]*. (We discuss the mappings from these schemas to the SM view in Example 2.)*

```
<!ELEMENT store (warehouse*)>
<!ELEMENT warehouse (city, state, item*)>
<!ELEMENT item (id, name, description)>
<!ATTLIST warehouse id ID #REQUIRED>

<!ELEMENT store (items, warehouses)>
<!ELEMENT items (item*)>
<!ELEMENT item (id, name, description)>
<!ELEMENT warehouses (warehouse*)>
<!ELEMENT warehouse (city, state)>
<!ATTLIST item warehouse-id IDREFS #REQUIRED>
<!ATTLIST warehouse id ID #REQUIRED>
```

The language we use to specify XML-to-SM mappings is based on (a subset of) XPath [71] and is similar to mapping languages, also called "transformation rules" or "source-to-target dependencies" in the literature (see, e.g., [4, 18]). A mapping for a binary relation p has the following general form:

```
p($X, $Y) <- path1 $G, $G/path2 $X, $G/path3 $Y.
```

where `$X` and `$Y` correspond to the arguments of p. The variable `$G` in the body of the rule is called the "glue" variable, and is used to restrict (`$X`,`$Y`) pairs to have the same `$G` ancestor element in the document.

Example 2 (Mapping rules) *Consider the first information source of Example 1. Some of the mapping rules that map data in this source to the SM view are as follows:*

```
item-name($I,$N) <-
      /store/warehouse/item $X, $X/id $I, $X/name $N.
item-warehouse($I,$W) <-
      /store/warehouse $X, $X/item/id $I, $X/@id $W.
warehouse-state($W,$S) <-
      /store/warehouse $X, $X/@id $W, $X/state $S.
```

[1] We omit declarations of elements of type `#PCDATA`.

3.3 Examples for the SM Approach

This section presents various examples for the SM approach from a simple system to a complex system with a three-tiered architecture.

3.3.1 A Simple System

A simple system consists of one coordinator and two data sources shown in Figure 3.4. This coordinator stores binary predicates for both data sources (A and B). The data format of A and B is XML. Note that sources can have different schemas and different (possibly overlapping) data and coverage (see Section 3.3.2). A user query is relational, which is based on binary predicates, such as $r \bowtie s$. For example, sources contain student information, and `id-lname`, `id-fname` and `id-major` are relations in the semantic-model view. A user query "List students' id, last name and major" is written in a relational query as following:

$$Q(id, lname, major) := (id - lname) \bowtie (id - major)$$

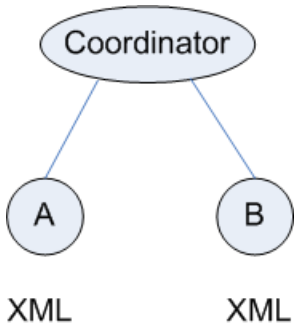

Figure 3.4: A simple system with only two sources.

The `subqueries` query processing (which will be described formally in Section 4.2) for this user query is the following:

1. A user query is posed on the coordinator.
2. The coordinator translates the user query into local subqueries and inter-source subqueries, whose format is XQuery. During the query translation, it should take the sources' schemas and mappings into consideration (see Section 4.2).

3. Those subqueries are executed at both Source A and Source B.

4. The coordinator collects the results of subqueries, and merges them into a final answer in a meaningful way (`Merge` operation is described in details in Section 4.6).

3.3.2 System with Heterogenous Sources

In the simple system, all sources are homogenous, although they may have various schemas. In a system with heterogenous sources, sources could store various formats of data, e.g., XML, relational tables, plain texts, etc.

Figure 3.5 shows that Source B stores relational tables instead of XML documents. From the user's point of view, the only difference between a simple system and system with heterogenous sources is mapping rules from sources to the SM view. XML-to-SM mappings are applied to a simple system; while in a system with heterogenous sources, Relation-to-SM mappings or mappings from other formats of data to SM view are used.

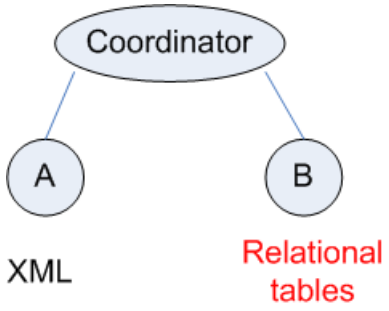

Figure 3.5: A system with heterogenous sources.

3.3.3 System with Multiple Coordinators

Figure 3.1 and Figure 3.6 show two examples of system with more than one coordinator. A system with multiple coordinators has a network of communicating coordinators, where each coordinator is in charge of a set of sources and, possibly, other coordinators. This architecture is basically a peer-to-peer arrangement of a set of single-coordinator systems, where each coordinator is in charge of its own community of sources. This system comprises a three-tiered architecture, with an extra group of mappings defined over SM views of a set of coordinators.

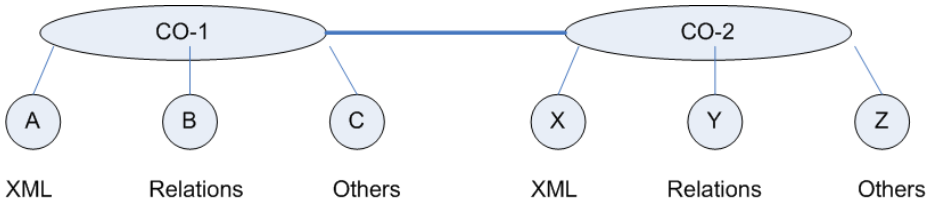

Figure 3.6: A system with two coordinators.

Multiple coordinators bring a new issue: how is a user query processed in the system with a three-tiered architecture?

Coordinators (or super peers) can serve as mapping providers, logical mediators, and/or mere query nodes. View mappings between disparate SM views are given locally between two (or a small set of) coordinators. Using these view mappings transitively, coordinators can make use of relevant data anywhere in the system. Consequently, queries in a system with multiple coordinators can be posed using the local SM view of the coordinator, without having to learn the SM view of other coordinators. This system with multiple coordinators are different from classic data integration systems, where queries are formulated via a global mediated schema, and all peers or sources must provide mappings from their schemas to this mediated schema.

3.4 Efficiency Issues and Optimization Opportunities

For each relation r in the SM view, a source i either stores the information for a *fragment* r_i of r, or has no data relevant to r. The fragment r_i is not materialized; only the mapping rule to generate r_i from the data at source i is available. We assume the data in the system corresponding to a relation r is the union of its fragments at every source, possibly subject to value mappings to reconcile heterogeneities.[2]

The answer to a user query Q should reflect the total data in information sources. That is, if Q mentions relation r, the answer is obtained as if the query were executed on $r = r_1 \cup \ldots \cup r_n$, where $r_1, \ldots, r_n$ are the fragments of r at the information sources in the federation. There are a number of ways to process user queries, with widely different

[2]The issue of data heterogeneity, such as different units of measurement, different scales, different terms for the same property, or the same term used for different properties, and ways to handle them are well known and will not be addressed in this dissertation.

performances. No single scheme is optimum for all queries and cases. Rather, an intelligent query-optimization approach needs to choose from a number of alternatives. We discuss our specific query-processing approaches and algorithms in Chapter 4, and present experimental results in Chapter 5. In the remainder of this section we discuss optimization challenges and opportunities in the semantic-model approach.

Consider a user query Q involving k relations $r^1, \ldots, r^k$ in the SM view.[3] Since each relation is the union of its fragments, Q can be regarded as a collection of n^k subqueries, where each subquery corresponds to one combination of single-source fragments of $r^1, \ldots, r^k$. In fact, one way of executing Q is to execute its corresponding subqueries, and then to merge the results. A subquery where all fragments are from the same source is called *local*; such subqueries can be executed at the source with no additional data transfer. A subquery with fragments from two or more sources is an *inter-source subquery*; its execution requires data transfer. Note that out of n^k subqueries, only n are local. The majority $(n^k - n)$ are inter-source subqueries.

The need for inter-source processing arises naturally in information integration. An application may need information from many sources with different kinds of data. For example, a security application may benefit from integrating many sources involving banking, travel, investment, employment, or taxes. There are important queries that need data from many of these sources simultaneously. Even when all sources belong to the same domain, there may be a need for inter-source processing. For example, academic data sources list information about their faculty, students, research, or publications. Each source has complete information about its personnel, but there may be collaboration between groups from different universities. There are queries where the results of local subqueries are only a strict subset of all answers. The need for inter-source processing poses a significant challenge to query optimization in data integration. Blindly executing all subqueries, or, alternatively, materializing the SM view relations by computing their fragments and unioning them, can be very costly. Our approach is based on using known integrity constraints, such as key and foreign-key constraints, to minimize the amount of inter-source processing that is needed. We present a method to determine when no inter-source processing is required to process a given query (Section 4.7). For queries that require some inter-source processing, we present a method to determine the minimum number of subqueries that are required (Section 4.8).

[3] We use superscripts for relations and subscripts for fragments. For example, r_i^j represents the fragment of relation r^j at source i.

These results are of general interest and can be used in any information-integration system that avoids loading all the data into a single repository.

Chapter 4

Algorithms and Theoretical Results

In this chapter we discuss the main query-processing algorithms and theoretical results that will play a significant role in query optimization. In addition to the basic *materialization* approach for query evaluation, we describe the *subqueries*, the *optimized-subqueries*, the *wrapper* approaches, and their semantically optimized versions. Then we discuss the theoretical results that allow us (1) to determine when no inter-source processing is needed (Section 4.7), and, (2) when inter-source processing cannot be avoided, to determine minimal equivalence sets of subqueries that are adequate to provide the complete answer to a user query (Section 4.8). These optimization techniques are then discussed in Chapter 5, along with the experimental results.

4.1 Query Processing I: Materialization

This is the *base* query-processing approach against which we evaluate other approaches. In the simple *materialization* approach, we materialize the SM view relations that appear in the user query, and execute the query on the materialized relations. Example 3 illustrates the approach.

Example 3 *Consider a system with four information sources and a mediator. Suppose query* $Q = \Pi_{B,C}\ \sigma_P(r \bowtie s)$ *involves relations* $r(A, B)$ *and* $s(A, C)$ *in the mediator schema, with selection condition* P*. Assume XML data sources; Figure 4.1 shows a small part of two of the sources. Note that the sources have different schemas.*

```
...                          ...
  <X>                          <Z>
     <A>a1</A>                    <C>c3</C>
     <B>b1</B>                    <W>
     <Y>                             <A>a2</A>
        <C>c1</C>                 </W>
        <C>c2</C>              </Z>
     </Y>                      <Z>
  </X>                            <C>c4</C>
  <X>                             <W>
     <A>a2</A>                       <A>a3</A>
     <B>b2</B>                       <B>b3</B>
  </X>                            </W>
...                            </Z>
                             ...
source 1                     source 2
```

Figure 4.1: Part of information in two data sources.

In the materialization approach, we create two materialized relations (fragments) for $r(A, B)$ *and* $s(A, C)$ *in each source. The conditions of predicate* P *that involve only* r *or* s *are enforced at this point. The queries to create these fragments are generated using the mapping rules for each source. These materialized relations (*r_1*,* s_1*,* r_2*,* s_2*,* r_3*,* s_3*,* r_4*,* s_4 *in this example) are sent to the mediator. The mediator merges these relations and executes the user query on them.* ■

Although the approach is not efficient in general, we were surprised to find that this approach was relatively efficient in certain situations, see discussion in Chapter 5.

4.2 Query Processing II: Subqueries

The *subqueries* approach is based on generating local and inter-source subqueries for the user query, executing the subqueries, and merging their (partial) results. A *local subquery* uses data from a single source and can be executed at the source. An *inter-source subquery* needs data from multiple sources, and requires some data transmission for its execution. The algorithm for generating local subqueries was presented in [46].

Example 4 *Consider the user query* $\Pi_{B,C}\ \sigma_P(r \bowtie s)$ *of Example 3. We have (up to)*

$4^2 = 16$ *subqueries.*[1] *Only four of these subqueries,* $\Pi_{B,C}\ \sigma_P(r_i \bowtie s_i), i = 1, \ldots, 4$, *are local; the remaining twelve are inter-source subqueries. The local subqueries are translated to queries on the source schemas and executed locally; the results are sent to the mediator. For an inter-source subquery such as* $\Pi_{B,C}\ \sigma_P(r_1 \bowtie s_2)$, *either the data for* r_1 *is sent to Source 2, or the data for* s_2 *is sent to Source 1. (We have developed algorithms for data transmission and subquery execution for XML data sources.) Finally, the mediator merges all partial results in order to obtain the final answer to the user query.* ■

The pseudocode for generating inter-source subqueries is shown in Algorithm 1.

Algorithm 1: Inter-source subquery generation

input : User query Q, order of data sources, set of binary predicate mappings

output: Inter-source subquery(XQuery) Q'

```
1  foreach binary predicate p in Q do
2      create one variable for p with specifying data locations by p's mapping
       and the order of data sources in Q';
3      foreach attribute attr in p do
4          create one variable for attr using the variable p above in the FOR
           clause of Q';
5      end
6  end
7  construct a WHERE clause in Q' if Q has constraints;
8  foreach element ele in Q's head do
9      specify ele which should be returned in Q';
10 end
11 return Q' with a FLWOR expression;
```

4.3 Query Processing III: Optimized Subqueries

The *optimized subqueries* approach uses formal results to eliminate, to the extent possible, inter-source processing. Based on key and foreign-key constraints that are relevant

[1] If source i has no data for relation r then all subqueries involving fragment r_i are empty and need not be executed.

to the query, all or some of inter-source subqueries may be redundant and will not be evaluated. The savings can be substantial; for instance, given a query involving k mediator-based relations in a system with n sources, there are only n local subqueries, while the number of inter-source subqueries can be as large as $n^k - n$. The details are presented in Sections 4.7-4.8.

Example 5 *In the setting of Example 3, suppose that attribute A is the key for r and that a foreign-key constraint holds from s.A to r.A. Then, by Theorem 2 (see Section 4.7), no inter-source processing is needed. This reduces the processing from sixteen to four (all local) subqueries that are translated and executed locally on the data sources.* ■

4.4 Query Processing IV: Wrapper

In the *wrapper* approach, we generate one subquery per information source; the subquery extracts from the source the minimum amount of information that is needed to answer the user query. We call this the "wrapper" approach because this extraction can be viewed as a (query-specific) wrapper that collects the needed information from each source. Compared to the *subqueries* and *optimized subqueries* approaches, the information extracted from each source in the wrapper approach is richer than the result of the local subquery on the same source, and makes it possible to obtain the full answer to user query by further processing. In a large class of applications, an efficient *chase*-based algorithm can be applied to the extracted information to obtain the full answer to the user query. The pseudocode for the wrapper algorithm is shown in Algorithm 2. The function to decide whether inter-source processing is needed mentioned in Line 6 is described in Algorithm 5.

Example 6 *Consider the user query $\Pi_{B,C}\ \sigma_P(r \bowtie s)$ of Example 3. Suppose that A is the key of r, but no foreign-key constraint holds from s to r (thus, according to Theorem 2 in Section 4.7, inter-source processing is needed.) In the wrapper approach, each source i generates a relation $t_i(A, B, C)$ corresponding to the full outer-join of r_i and s_i and sends it to the mediator. The mediator combines these relations, applies the chase, and enforces the query conditions and projections. In our example, source 1 (see Figure 4.1) has the following tuples (among others): (a1, b1), (a2, b2) for r_1 and (a1, c1) and (a1, c2) for s_1. Hence t_1 contains (a1, b1, c1), (a1, b1, c2), and (a2, b2, null). Source 2 has tuple (a3, b3) for r and tuples (a2, c3) and (a3, c4) for s. Then t_2 contains (a2, null, c3) and (a3, b3,*

c4). The result of unioning t_1 and t_2 (and t_3 and t_4) and chasing with respect to the key constraint generates a new tuple (a2, b2, c3) in the result, since A is the key of $r(A, B)$, null in (a2, null, c3) is replaced by b2 which comes from (a2, b2, null). The final step is to enforce predicate P and to project over B, C. The answer contains (b2, c3) (unless filtered out by P). ■

Algorithm 2: The Wrapper Algorithm

input : User query Q, set of sources $SRCS$ and their mappings MPS
output: Single XML document *Doc*

```
1  foreach source s in SRCS do
2  |  create a local subquery which allows nulls in the result for s;
3  |  execute the subquery locally, then send the local result to the
   |  coordinator;
4  end
5  merge the local results into query answer ans at the coordinator;
6  if inter-source processing is needed then
7  |  chaseSteps();
8  end
9  eliminate duplicates in ans;
10 save ans into an XML document Doc;
11 return Doc;

12 Function chaseSteps()
13 foreach constraint cstrt in Q and MPS do
14 |  replace nulls at ans using Chase algorithm with cstrt;
15 end
16 End Function
```

Note that in the subquery-based approaches, the answer (b2, c3) of Example 6 is generated by the inter-source subquery $\Pi_{B,C}\ \sigma_P(r_1 \bowtie s_2)$ (tuple (a2, b2) $\bowtie$ tuple (a2, c3)). Thus, while executing only local queries, the wrapper approach is able to generate the results of inter-source subqueries at the coordinator (mediator).

4.5 Semantic Optimization for XQuery

In the approaches presented in Sections 4.2–4.4, the system generates queries to be executed on local data (*e.g.*, XQuery queries on XML data). In our implementation framework of [70], user queries tend to have a relatively large number of joins of binary relations in the semantic-model view. When the above translation algorithms are adapted to this setting, they create one variable in the XQuery query for each binary relation in the user query. Our platform-specific semantic optimization rewrites XQuery queries into more efficient equivalent queries with fewer joins and variables. The algorithm uses information from mapping rules and from key constraints of the binary relations in the semantic model. Instead of creating one variable for each binary relation in the user query, the new algorithm generates a single variable for all binary relations that have a common "glue" variable in their mappings and the same key. Consider the following illustration of our semantic-optimization rewriting process.

Example 7 *Suppose an application involves multiple data sources with information on stocks (see Section 6.7.2 for the background). The semantic-model view for this application contains binary relations* `k-ticker, k-year, k-month, k-day, k-price, k-priceType`*, where* `k` *is the unique key. Each relation name refers to a non-key stock attribute. For example,* `k-ticker` *has two attributes:* `k` *is a unique key, and* `ticker` *is the ticker (stock) id. For each stock, we recorded four price types: open, close, high, and low. A mapping rule for* `k-ticker` *in an XML source could be:*

```
k-ticker($X, $Y) <-
      /stocks/stock $G, $G/@uid $X, $G/ticker $Y
```

As an example for the improvement obtained by semantic query rewriting, consider a user query that produces the average closing price for IBM in October 2005. This query uses five of the above binary relations, and a corresponding XQuery will have five variables on the XML stocks document. But since these five binary relations have the common glue variable `/stocks/stock` *and the same key* `k`*, we would obtain the following local subquery with just one variable:*

```
LET $br := /stocks/stock[year=2005 and month='Oct' and
  ticker='IBM' and prices/priceType = 'close']
RETURN avg($br/prices/price)
```

The effect of the semantic optimization, as demonstrated by the experimental evaluation in Section 5.5 and Section 7.4, can be significant (up to two orders of magnitude for certain queries). By utilizing this technique of semantic optimization, we develop new algorithms (semantically optimized versions) of *subqueries*, *optimized subqueries*, and *wrapper*, which are *subqueries**, *optimized subqueries**, and *wrapper** respectively.

4.5.1 Query Processing V: Subqueries*

Note that the *subqueries* approach is based on generating local and inter-source subqueries for the user query, executing the subqueries, and merging their (partial) results. The semantic optimization takes effect when local and inter-source subqueries are generated. The rest of query processing in the *subqueries** approach is exactly same as that in the *subqueries* approach. An example of the *subqueries** approach is shown in Section 5.5.1.

4.5.2 Query Processing VI: Optimized Subqueries*

The *optimized subqueries** approach is a semantically optimized version of the *optimized subqueries* approach, where local and inter-source subqueries which are needed to be evaluated in the *optimized subqueries* approach are semantically optimized.

4.5.3 Query Processing VII: Wrapper*

Recall that in the *wrapper* approach, we generate one subquery per information source; the subquery extracts from the source the minimum amount of information that is needed to answer the user query. In the *wrapper** approach, this subquery is transformed by generating a single variable for all binary relations that have a common "glue" variable in their mappings and the same key.

4.6 Merging XML Data

Given two or more XML documents on the same schema, our *merge algorithm* produces one XML document on the same schema; the document contains all the data from the input documents. In case merging the input documents is not possible, the algorithm outputs the discrepancies that hindered the merge operation.

As an example, consider several XML documents of *personnel* information. Certain natural consistency constraints are expected to hold on this kind of data. For example, each individual has a social security number that uniquely identifies the person's name and date of birth. A person may have multiple phone numbers, but the date of birth should be unique. The output of running the merge algorithm on such personnel data should contain all individuals mentioned in the input files. In addition, the information for one individual (determined by the same SSN in different inputs) is combined in the output document in the intuitive way: The date of birth of the same individual from different inputs, if known, should be identical. If not, merge is not possible and a discrepancy in the date of birth of this individual is identified. Further, phone numbers from multiple inputs for the same individual are all included in the merged information for that individual. In our prototype implementation, merge is halted if a discrepancy is detected. Many other approaches are possible, including approaches that use information about the degree of reliability of sources to guide the merge in presence of discrepancies, and can be incorporated with relative ease.

In general, assume D_1 and D_2 are two documents with the same schema S. Let E be an element type in S that has a logical identifier, and is at a maximal level (i.e., it does not have any ancestor with a logical identifier). Let e_1 be an element (instance) in D_1 of type E, and e_2 be an element of type E in D_2 that has the same value for its logical identifier as e_1. In order to retrieve the merged document $D(D = D_1 + D_2)$, we classify E into several types as follows:

- If E has a $single - valued, required$ child or descendent C that is a leaf (i.e., is an attribute or an element with no subelements), then e_1 and e_2 must have the same value for C (if not, issue an error message). The merged element $e = e_1 + e_2$ should have the same value for C. A descendent is called *single-valued, required* when all elements on the path from the ancestor to the descendent are single-valued and required. For Example, element $person$ may have a single-valued, required subelement $dateOfBirth$. It may also have s single-valued, required subelement name, with its own single-valued, required subelements $firstName$ and $lastName$.

- If E has a $single - valued, optional$ child or descendent C that is a leaf (i.e., is an attribute or an element with no subelements), then e_1 and e_2 must have the same value for C, or one or both can be empty (if not, issue an error message). The merged element $e = e_1 + e_2$ should have the same value for C (if both, or one of e_1 and

e_2 have a value). For example, Element *person* may have a single-valued, required subelement *name*, which has a single-valued, optional subelement $middleInitial$ (as well as required subelements $firstName$ and $lastName$).

- If E has a *single-valued, required* child or descendent C that is not a leaf, but has a logical identifier of its own, then the instances c_1 and c_2 belonging to e_1 and e_2, respectively must be logically equal. This means, they should have the same identifiers. The merged element e will have a subelement c obtained by (recursively) merging e_1 and e_2.

- E is a *single-valued, optional* child or descendent that is not a leaf, where e_1 or e_2 is optional, or e_1 and e_2 must have same value.

- For $multi-valued$ children and descendants (e.g., children declared with a + or * in DTD schema, or maxOccurs with the value of more than 1 in the XML Schema), the merge rule is somewhat similar to combining these children with duplicate elimination. For example, each when merging *person* elements, the books written by a person (from different documents) are combined, keeping one instance of each book. To determine duplicates, logical equivalence discussed above is used. For example, a book element is a duplicate of another book element if it has the same value for its logical identifier. The resulting book element in the merged document is the merge of the two book elements. The $multi-valued$ class is divided into a few subclasses according the "maxOccurs" and "minOccurs" attributes in the XML Schema: Optional (minOccurs=0), Unique (no duplicate instance is allowed), Reduplicate (duplicate instances are allowed).

The pseudocode for the merge algorithm is shown in Algorithms 3 and 4.

4.7 Eliminating Inter-Source Subqueries

In this subsection and in Section 4.8, we present theoretical results that play a significant role in query optimization in the semantic-model approach. Our first result addresses the question "when is inter-source processing not needed?" To motivate this investigation, let us first obtain an intuition about the amount of inter-source processing that may be needed: Consider a system with n information sources, and a user query

Algorithm 3: The Merge Algorithm

input : XML documents $xmls$ and their schema sma

output: Single XML document Doc

```
1  retrieve keys, unique nodes and other constraints from sma;
2  treewalk all elements from the root and classify elements into types such as
   single − valued, required, leaf, non − leaf, multi − valued, and so on;
3  Doc = fetchUnVisitedDocumentFrom(xmls);
4  while xmls has any other document which has not been visited do
5      nextDoc = fetchUnVisitedDocumentFrom(xmls);
6      while (E = getElementFromTreewalk(sma)) !=null do
7          e1 = getInstanceFromDoc(Doc, E);
8          e2 = getInstanceFromDoc(nextDoc, E);
9          mergeElements(e1, e2);
10     end
11 end
12 return Doc
```

involving k relations in the semantic model. The total number of possible subqueries, where the data for each of the k relations come from one of the n sources, is n^k. Only n of these are local, in the sense that all data come from the same source. The remaining $n^k - n$ may require some degree of inter-source processing. This is, of course, a worst-case scenario. In practice, even when the total number of sources is very large, a specific relation in the SM view has a limited number of sources with data pertaining to that relation, reducing the possible inter-source queries to $m^k - n$, where $m << n$ is the number of sources with data for a given relation, on the average. Nevertheless, in large-scale information integration, where n can be in the hundreds or even thousands or higher, this number can still be quite large. If we are able to identify the minimum amount of inter-source processing that is required, and restrict our query evaluation to avoid any extra work, we can potentially achieve orders of magnitude faster query processing in large-scale integration.

Before we introduce a new theorem of eliminating inter-source subqueries, we study the characteristics of binary relations in the SM approach. Recall that binary relations specify metadata and semantic concepts about data sources. Some binary relations with

Algorithm 4: The Merge Algorithm (continued)

```
Procedure mergeElements(e1, e2)
typeVal = getType(e1);
switch typeVal do
    case single-valued and required
        if e1 != e2 then
            print ERROR;
        end
        break;
    end
    case single-valued and optional
        if (e1 != null) AND (e2 != null) then
            if e1 != e2 then
                print ERROR;
            end
        end
        if e1 == null then
            e1 = e2;
        end
        break;
    end
    case multi-valued
        append e2 and its descendants to e1;
        eliminate duplicates according to E's attribute;
        break;
    end
end
```

a key are generated to explain complex relationship between attributes in a source. For example, a source has a relational table with its schema $R(A, B, C)$, where a set of attributes $R.A$ and $R.B$ is the key of R. In the SM view for this source, a unique id *uid* is created, and the following binary relations as well: UID-A (*uid*, A), UID-B (*uid*, B), and UID-C (*uid*, C).

In [46], the authors study user queries that involve the natural join of two relations of the SM view and defines cases when inter-source processing is not needed. We generalize their result to user queries with any number of relations. First, we state the result from [46]:

Theorem 1 *Consider a user query involving the natural join of relations $r^i(A, B)$ and $r^j(B, C)$. No inter-source processing is needed for this query if all the following conditions hold:*

1. *Key constraint: For every source k, B is the key for the fragment r^j_k.*
2. *Foreign-key constraint: For every source k, there is a foreign-key constraint from $r^i_k(B)$ to $r^j_k(B)$.*
3. *Consistency constraint: If $r^j_k(b, c)$ and $r^j_l(b, c')$ hold at two sources k and l, then $c = c'$.*

■

In our generalization we use the following definition:

Definition 1 (Local-Join Graph) Given $r^i(A, B)$ and $r^j(B, C)$, if the three conditions of Theorem 1 hold then we say r^i and r^j have the *local-join property*, and their relationship is illustrated in Figure 4.2.

Figure 4.2: Relationship between two binary relations

Let $r^1, \ldots, r^m$ be all the relations in the SM view. The *local-join graph* is a directed graph $G = (N, E)$, where the set of nodes N corresponds to the relations $r^1, \ldots, r^m$, and $(r^i, r^j) \in E$ if r^i and r^j have the local-join property. ■

A new theorem follows:

Theorem 2 *Given a user query involving the natural join of two or more relations $r^1, \ldots, r^k$, if the local-join graph restricted to the query relations $\{r^1, \ldots, r^k\}$ contains a directed spanning tree, then no inter-source processing is needed for this query.* ■

In the proof of Theorem 2 we use the following lemma.

Lemma 1 *If there is an edge from r^i to r^j in the local-join graph, then $(r^1_{s_1} \bowtie \cdots \bowtie r^i_x \bowtie \cdots \bowtie r^j_x \bowtie \cdots \bowtie r^k_{s_k})$ subsumes $(r^1_{s_1} \bowtie \cdots \bowtie r^i_x \bowtie \cdots \bowtie r^j_y \bowtie \cdots \bowtie r^k_{s_k})$ for all s_i, x, and y.*

Proof (Lemma 1): By the local-join property, since r^i has a foreign-key constraint to r^j, then all tuples of the fragment r^i_x at a source x participate in the join with the fragment r^j_x at the same source. Further, by the consistency condition, $r^i_x \bowtie r^j_y$ cannot generate any tuple that is not already in $r^i_x \bowtie r^j_x$. Then, $r^i_x \bowtie r^j_x \supseteq r^i_x \bowtie r^j_y$ for all x and y. Join both sides with $r^1_{s_1} \cdots r^k_{s_k}$. The result of the lemma follows. ■

Proof of Theorem 2:

Let G be the local-join graph of the predicates, and H be the restriction of G to the query relations. Then we want to show that if H has a directed spanning tree T', then the query does not require inter-source processing. Without loss of generality, assume r^1 is the root of T, and $r^1, r^2, \ldots, r^k$ is the depth-first search order of T (any ordering compatible with the parent-child ordering of T will work, including the DFS order). The user query can be written as $r^1 \bowtie \cdots \bowtie r^k$. We will show that each inter-source subquery $r^1_{s_1} \bowtie r^2_{s_2} \bowtie \cdots \bowtie r^k_{s_k}$ is subsumed by the local subquery at source s_1, namely $r^1_{s_1} \bowtie r^2_{s_1} \bowtie \cdots \bowtie r^k_{s_1}$. For simplicity, we use $(s_1, s_2, \ldots, s_k)$ to represent the subquery $r^1_{s_1} \bowtie r^2_{s_2} \bowtie \cdots \bowtie r^k_{s_k}$. We now show, by (backward) induction, from $j = k$ to $j = 1$, that $(s_1, \ldots, s_1) \supseteq (s_1, \ldots, s_1, s_{j+1}, \ldots, s_k)$

Basis $j = k$: Obviously, $(s_1, s_1, \ldots, s_1) \supseteq (s_1, s_1, \ldots, s_1)$.

Induction: Assume the inductive hypothesis holds for $j+1$, that is, $(s_1, \ldots, s_1) \supseteq (s_1, \ldots, s_1, s_1, s_{j+1}, \ldots, s_k)$. We want to show $(s_1, \ldots, s_1) \supseteq (s_1, \ldots, s_1, s_j, s_{j+1}, \ldots, s_k)$. Let r_i be the parent of r_j in the directed spanning tree. Then, since $r_1, \ldots, r_k$ is the depth-first ordering of the tree, we must have $i < j$. By Lemma 1, we have $(r^{s_1}_1 \bowtie \cdots \bowtie r^{s_1}_i \bowtie \cdots \bowtie r^{s_1}_j \bowtie \cdots \bowtie r^{s_k}_k)$ subsumes $(r^{s_1}_1 \bowtie \cdots \bowtie r^{s_1}_i \bowtie \cdots \bowtie r^{y}_j \bowtie \cdots \bowtie r^{s_k}_k)$, for all y. Let $y = s_j$. Written in our notation: $(s_1, \ldots, s_1, s_{j+1}, \ldots, s_k) \supseteq (s_1, \ldots, s_j, s_{j+1}, \ldots, s_k)$. Combined with the inductive hypothesis and by transitivity of subsumption we get: $(s_1, \ldots, s_1) \supseteq (s_1, \ldots, s_j, s_{j+1}, \ldots, s_k)$. This completes the proof of the induction.

Hence, the n local subqueries $(s_1, \ldots, s_1)$, $s_1 = 1, \ldots, n$ (n is the number of sources) subsume all inter-source subqueries. Thus, no inter-source processing is needed in the computation of the user query. ■

Table 4.1 presents the position of a node (binary relation) in a local-join graph when no inter-source processing is needed. For example, if a binary relation r has no primary key, but has a foreign-key constraint to other relation's attribute, the position of r is a root in the local-join graph when no inter-source processing is needed.

Table 4.1: Position of nodes in a local-join graph when no inter-source processing is needed.

Primary Key	Foreign-key Constraint	Position at the graph
-	-	N/A
-	√	root
√	-	leaf
√	√	any

Example 8 *According to Theorem 2, no inter-source processing is needed for a user query ($r \bowtie s \bowtie t$) with a local-join graph shown in Figure 4.3 (a) and (b); while in Figure 4.3 (c), inter-source processing is needed for this user query (we will see in Section 4.8 that, even when inter-source subquereis are needed, we are still able to determine, based on key and foreign key constraints, a subset of all subquereis that are adequate for the evaluation of user query).* ■

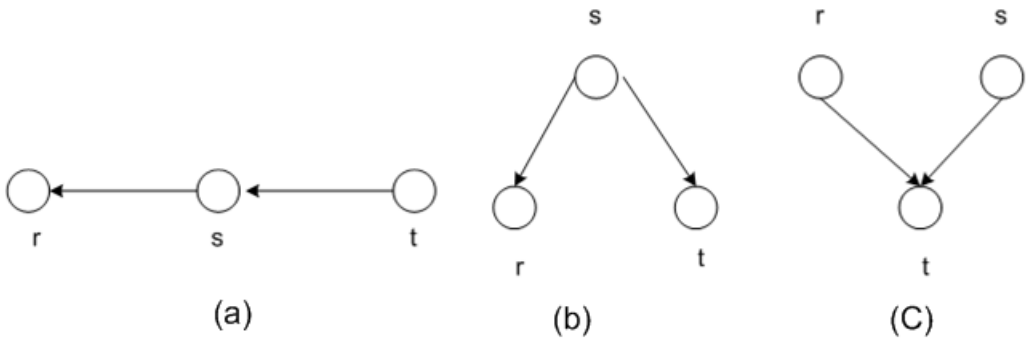

Figure 4.3: Examples for local-join graphs

Theorem 2 gives a sufficient condition, based on key and foreign-key constraints, where no inter-source processing is needed for the evaluation of a user query. The question

naturally arises as to whether the condition of Theorem 2 is also necessary. In other words, if the restriction of the local-join graph to query relations does not have a directed spanning tree, does the evaluation of the query require evaluation of some inter-source subqueries? We should first mention that there are weaker semantic constraints (than key, foreign-key constraints) that may provide conditions for Theorem 2 [46]. However, these semantic constraints are, to the best of our knowledge, not available as standard features in commercial databases. (At the same time, it is possible to enforce such conditions by means of triggers.) Hence, we restrict ourselves to key and foreign-key constraints. This means that if there is no edge from r_i to r_j in the local-join graph, then no constraints of any form exist between r_i and r_j. The following theorem addresses the issue of whether the conditions of Theorem 2 are also necessary in the positive.

Theorem 3 *Given a user query involving the natural join of two or more relations $r_1, \ldots, r_k$, if the local-join graph restricted to the query relations $\{r_1, \ldots, r_k\}$ does not contain a directed spanning tree, then a database instance exists where at least one inter-source subquery is not subsumed by any local subqueries.* ■

Proof: Let G be the local-join graph restricted to query relations. We show the nodes N of G can be partitioned to 3 subsets: N_1 contains a node r_i and all nodes reachable from r_i; N_2 contains a node r_j that can not reach r_i and is not reachable from r_i, plus all nodes that can reach r_j; and N_3 contains the remaining nodes. We show that N_1 and N_2 are nonempty. Now, we build an instance involving two information sources, where the fragments (1) satisfy the constraints dictated by G, while, (2) there is at least one tuple t in the inter-source join involving relations corresponding to N_1 from source 1, and relations corresponding to N_2 from source 2, such that t is not in the result of any local subquery. ■

The pseudocode for how to decide whether inter-source processing is needed is shown in Algorithm 5. This algorithm is a modified Depth-First Search (DFS) over G: firstly, check each vertex in turn and, when an un-visited vertex is found, add it into a set S and visit it using the following modified DFS-VISIT: if a vertex in the set S is visited from vertexes who are finished after it (that's, are out of its tree), this vertex is removed from this set S. Secondly, check the amount of elements in S: if the amount is more than one, return true; otherwise, return false. The running time is exactly same to that of Depth-First Search, Θ(V+E), where V is the number of nodes (binary relations in the user query), and E is the number of edges in the local-join graph of this user query.

Algorithm 5: Algorithm to decide whether inter-source processing is needed

input : A local-join graph G

output: A boolean value: true or false. If true, inter-source processing is needed; otherwise, it is not needed.

```
1  foreach vertex u in V[G] do
2  |  color[u] = WHITE;
3  end
4  S = null;
5  foreach vertex u in V[G] do
6  |  if color[u] == WHITE then
7  |  |  S.add(u);
8  |  |  DFS − VISIT(u);
9  |  end
10 end
11 if S.size() > 1 then
12 |  return true;
13 else
14 |  return false;
15 end
16 Function DFS-VISIT(u)
17 color[u] = GRAY;
18 foreach vertex v in Adj[u] do
19 |  if S.find(v) == true then
20 |  |  S.erase(v);
21 |  end
22 |  if color[v] == WHITE then
23 |  |  DFS − VISIT(v);
24 |  end
25 end
26 color[u] = BLACK;
27 END Function
```

4.8 Partitioning Inter-Source Subqueries

Given a user query, if the condition of Theorem 2 does not hold then some inter-source processing is needed. In this section we discuss the problem of determining the set of subqueries that are needed for the evaluation of the user query. In particular, we will show a counterintuitive result, namely that the set of needed subqueries is not unique, rather, there can be multiple *equivalence sets* of subqueries. More specifically, we show by a simple example that the set of subqueries can be partitioned into (1) required subqueries, (2) redundant subqueries, with each subquery in this group being subsumed by a subquery in the required group, and (3) zero or more sets of equivalent subqueries, where we need to execute only one subquery from each equivalence class.

Example 9 *Consider a system with two sources ($n = 2$), and a user query involving the join of three relations $r(A, B)$, $s(A, C)$, $t(A, D)$. Further, assume A is the key for r, and foreign-key constraints hold from $s.A$ and $t.A$ to $r.A$. Also assume the consistency condition (condition 3 in Theorem 1) holds for r. Note that the local-join graph, restricted to r, s, and t, has edges from s and t to r, and does not have a directed spanning tree.*

There are $2^3 = 8$ subqueries. Two of them are local subqueries, namely, $r_1 \bowtie s_1 \bowtie t_1$ and $r_2 \bowtie s_2 \bowtie t_2$ (where r_i represents the fragment of r that comes from source i, similarly for s and t.) It is easy to verify that (see Section 4.8.1 below), for this query,

- *$r_1 \bowtie s_1 \bowtie t_1$ and $r_2 \bowtie s_2 \bowtie t_2$ are required.*
- *$r_1 \bowtie s_2 \bowtie t_2$ and $r_2 \bowtie s_1 \bowtie t_1$ are redundant: $r_1 \bowtie s_2 \bowtie t_2$ is subsumed by $r_2 \bowtie s_2 \bowtie t_2$, and $r_2 \bowtie s_1 \bowtie t_1$ is subsumed by $r_1 \bowtie s_1 \bowtie t_1$.*
- *$r_1 \bowtie s_1 \bowtie t_2$ and $r_2 \bowtie s_1 \bowtie t_2$ are equivalent, and so are $r_1 \bowtie s_2 \bowtie t_1$ and $r_2 \bowtie s_2 \bowtie t_1$.*

Hence, the user query can be evaluated fully by evaluating four subqueries (out of the total 8). There are four sets of such minimally-sufficient subqueries: Each set includes the two required subqueries, plus one subquery from each of the two equivalence classes in the third bullet above. ■

4.8.1 Determining the partition

We use a graph $G = (N, E)$, which we call the *subsumes graph,* to determine required, redundant, and equivalence classes of subqueries. Each node $q \in N$ represents

a subquery. There is also a special node ϕ, intended to represent subqueries with empty results. There is an edge from q_i to q_j if q_i subsumes (i.e., is a superset of) q_j. Further, there is an edge from node ϕ to a node q if the subquery represented by q includes an empty fragment — that is, q represents a subquery involving $r^1_{s_1} \bowtie \cdots \bowtie r^k_{s_k}$ where at least one of the sources s_i does not provide a mapping for the corresponding relation r^i, and hence, $r^i_{s_i}$ is empty. The following procedure can be used to partition the set of subqueries:

- Eliminate all nodes reachable from ϕ (these are the subqueries with empty answers);
- A node with in-degree zero in the remaining graph represents a required subquery;
- Nodes reachable from the required subqueries represent the redundant subqueries;
- Eliminate all nodes representing the required and redundant subqueries; the remaining nodes, if any, must be on cycles; all nodes on a given cycle represent equivalent subqueries.

Finally, how do we determine if one subquery subsumes another? We demonstrate the idea of the algorithm with an example.

Example 10 *Consider the same user query and constraints as in Example 9. Consider the edge from s to r in the local-join graph. The following is immediate from the key, foreign-key, and consistency conditions:*
$r_i \bowtie s_i \supseteq r_j \bowtie s_i$, for all $1 < i < n$ and $1 < j < n$ (n is the number of sources). It follows that $r_i \bowtie s_i \bowtie t_k \supseteq r_j \bowtie s_i \bowtie t_k$, for all $1 < i < n$, $1 < j < n$, and $1 < k < n$. Hence, there is an edge from $r_i \bowtie s_i \bowtie t_k$ to $r_j \bowtie s_i \bowtie t_k$, for all $1 < i < n$, $1 < j < n$, $j \neq i$, and $1 < k < n$, in the subsumes graph. ■

Chapter 5

Optimization Techniques and Experimental Results

Query-processing performance is a key issue in large-scale information integration. In this section we discuss query-optimization techniques for the semantic-model approach and present experimental results to evaluate the impact of the techniques on the query-processing performance and on the network traffic. We present our experimental results in Section 5.5, after discussing, in Section 5.1 - Section 5.2, the optimization techniques we used in our approach, and illustrating our experimental setup in Section 5.3 and Section 5.4.

Our experimental results show that (a) our methods are competitive with respect to the baseline materialization, and further more, the semantically optimized versions perform much better, and (b) the performance is query-type and environment dependent, suggesting learning approach can be used to establish robust rules or to develop cost functions to be used for dynamically selecting the best query processing method for a given query in a specific integration instance.

5.1 Reducing Inter-Source Processing

The results presented in Sections 4.7 and 4.8 allow us (1) to determine whether inter-source processing is needed, and, (2) if such processing is needed, to determine equivalent sets of inter-source subqueries that are required and adequate for answering the user query. These results can go a long way toward reducing the overall query-execution costs. Constraints (key and foreign-key) used in these results are quite common and should, in

many practical cases, enable us to avoid a large portion of costly inter-source processing. If no inter-source processing is required, the amount of processing needed is $O(n)$, where n is the number of sources, and the processing can be performed locally at each source. Only the last step, merging partial results, needs data transmission. Compare this to a naive execution of all subqueries, where the processing effort is significantly larger — $O(n^k)$, where k is the number of relations in the user query. The amount of data transfer in this case is also significant due to additional data transmission required for inter-source processing.

5.2 The Chase Approach

Our wrapper approach generates for each source local subqueries that, intuitively, extract the minimum amount of information needed to answer the user query. More specifically, if the user query involves the join of relations $r^1, \ldots, r^k$, the wrapper extracts the *outer join* of $r_i^1, \ldots, r_i^k$ from each source i. (Recall that r_i^j denotes the fraction of r^j stored at source i.) With care, selections can also be pushed down to the wrapper. But we have to be careful not to miss any valid answers because of null values. Intuitively, this means we need to treat comparisons with nulls as *true* at the wrapper level, and repeat the comparison once the null has been replaced by a value in our chase process.

An important optimization approach in the wrappers technique is the application of the well-known chase method [68]. In some cases we can obtain the complete answer to the user query by unioning the extracted data, performing a chase with respect to key constraints, enforcing the remaining conditions and projecting on the desired attributes. The chase (with respect to a key constraint, or, more generally, with respect to a functional dependency) can be used to fill in values for nulls in certain cases. The following example illustrates the idea of our algorithm.

Example 11 *Consider a user query on the SM view that involves $r(A, B) \bowtie s(A, C)$, and assume A is the key of r. Assume the wrapper generates (among other tuples) $(a, b, null)$ from source 1 and $(a, null, c)$ from source 2. The result of unioning these two answers and chasing with respect to the key constraint generates the tuple (a, b, c), since the A-value a is associated with B-value b in the first tuple, hence the null in the second tuple can be replaced by b. Note that (a, b, c) is in the (inter-source join) $r_1 \bowtie s_2$ and belongs in the answer to the user query.* ■

An efficient sort-based or hashing-based algorithm can be used to implement the chase with respect to functional dependencies. The question arises as to when this simple processing would generate *all* answers to the user query. The answer is, not surprisingly, when the key constraints guarantee the *lossless join* property for the join in the user query.

5.3 Scenario I: DB-Research

```
[proceeding]
proceeding-title
proceeding-year
proceeding-location
proceeding-gc(general-chair)
proceeding-pc(program-chair)
proceeding-member(program-committee)

[paper]
paper-title
paper-author
paper-conference
paper-cite
paper-status

[project and person]
project-member
project-paper
project-area
project-topic
person-adviser

[review]
person-advisee
review-reviewer
person-affiliation
review-rating
person-homepage
review-paper
review-comment
```

Figure 5.1: Semantic model for DB-Research.

In our experiments we used the setup of [65] [1], with the modification that in

[1] We are grateful to Igor Tatarinov and Alon Halevy for providing us with their experimental setup for

addition to the queries used in [65] we defined several extra queries that would require more inter-source processing. Our first set of experiments used the "DB-Research" dataset, which contains data sources pertaining to several universities such as UC Berkeley and University of Washington, research organizations, and publications information from sources such as CiteSeer [13], DBLP [19], and SIGMOD [63]. Our semantic model for the DB-research dataset, shown in Figure 5.1, is based on the *academic department ontology* [1] from the DAML ontology library [17].

```
<!-- paper -->
<bp name="paper-title">
  <key>paper</key>
  <fkeys>paper-conference</fkeys>
</bp>
<bp name="paper-author">
  <fkeys>paper-title</fkeys>
  <fkeys>paper-conference</fkeys>
</bp>
<bp name="paper-conference">
  <key>paper</key>
  <fkeys>paper-title</fkeys>
  <fkeys>proceedings-title</fkeys>
  <fkeys>proceedings-year</fkeys>
</bp>
<bp name="paper-status">
  <key>paper</key>
  <fkeys>paper-title</fkeys>
  <fkeys>paper-conference</fkeys>
</bp>
```

Figure 5.2: Configuration for local-join property.

A configuration XML document is used to store key, foreign-key and consistency constraints for each binary relation in the SM view in DB-research. If binary relation R and S have the `local-join property`, and there is a foreign-key constraint from R to S, then S is an "*fkey*" of R. Figure 5.2 illustrates the key constraint and `local-join property` for binary relations involving paper information. In the following subsections, we examine each source in this scenario and user queries we will evaluate.

[65], including the data, schemas, and queries.

5.3.1 Data Sources

In this scenario, two coordinators share the same SM view, and therefore, view mappings between these two SM views are easy to generate. Using the view mapping transitively, coordinators can make use of relevant data anywhere in the system. Consequently, queries in a system with multiple coordinators can be posed by using the local SM view of one coordinator. Data sources are connected to one of two coordinators through binary relation mappings from the SM view to data sources (see Figure 5.3). In this experiment, sources are of various XML schemas shown in Figure 5.4 and Figure 5.5.

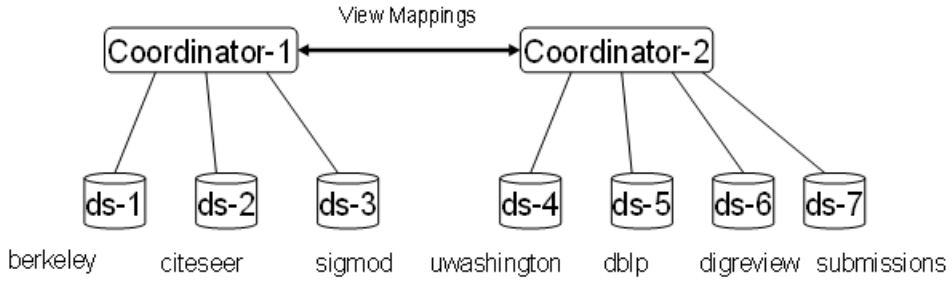

Figure 5.3: Scenario I: DB-research.

`Coordinator-1` is in charge of a set of sources: `berkeley`, `citeseer`, and `sigmod`. Actually, this coordinator can be regarded as a source, or as a broker for the information under its oversight.

DS-1 presents data of "database research" groups at UC Berkeley including projects, group members and their publications. The following is a part of data from the DS-1's XML document:

```
...
<direction>
  <name>Adaptive Information Systems</name>
  <project>
    <name>Telegraph XML</name>

    <people>
      <faculty>Joseph Hellerstein</faculty>
      <faculty>Michael Franklin</faculty>
      <student>Sirish Chandrasekaran</student>
      <student>Amol Deshpande</student>
```

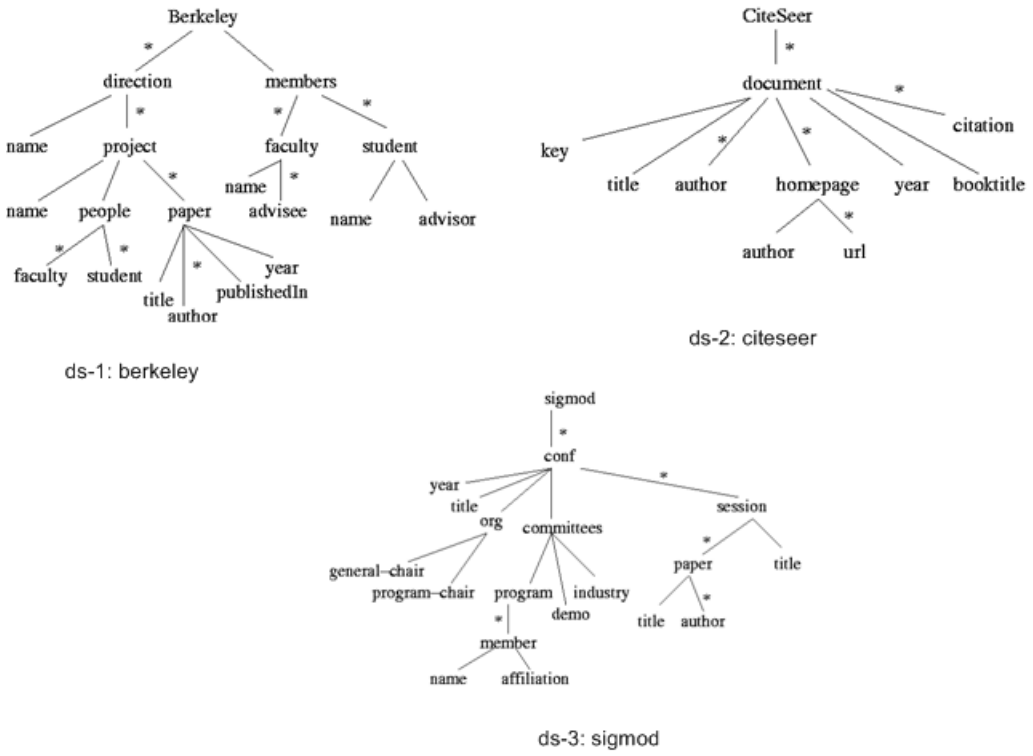

Figure 5.4: Schemas of data sources in Scenario I.

```
    <student>Ryan Huebsch</student>
    <student>Sailesh Krishnamurthy</student>
    <student>Boon Thau Loo</student>
    <student>Sam Madden</student>
    <student>Fred Reiss</student>
    <student>Mehul Shah</student>
  </people>

  <paper>
    <id>ChandrasekaranF02</id>
    <title>Streaming Queries over Streaming Data</title>
    <author>Sirish Chandrasekaran</author>
    <author>Michael Franklin</author>
    <author></author>
    <publishedIn>VLDB02</publishedIn>
    <year>2002</year>
  </paper>

</project>
```

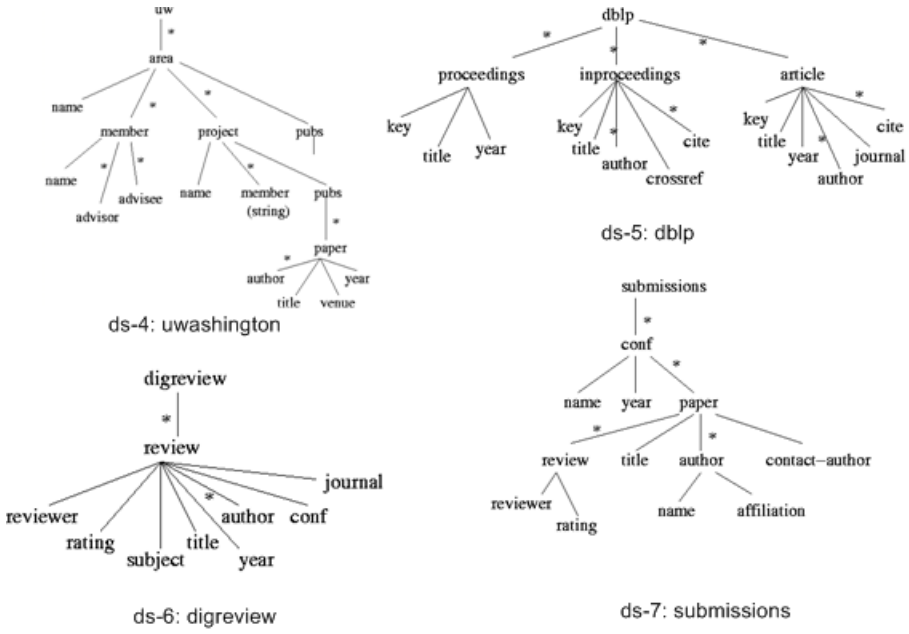

Figure 5.5: Schemas of data sources in Scenario I (continued).

```
</direction>
...
```

We also generate a set of mappings (discussed in Section 3.2) from the SM view to DS-1. For example, we create one rule for each binary relation in DS-1:

```
...
 <mapping>
   <predicate>paper-title</predicate>
   <pathGlue>berkeley/direction/project/paper</pathGlue>
   <firstarg type="variable">id</firstarg>
   <secondarg type="variable">title</secondarg>
 </mapping>
  <mapping>
   <predicate>paper-author</predicate>
   <pathGlue>berkeley/direction/project/paper</pathGlue>
   <firstarg type="variable">id</firstarg>
   <secondarg type="variable">author</secondarg>
 </mapping>
  <mapping>
   <predicate>paper-conference</predicate>
   <pathGlue>berkeley/direction/project/paper</pathGlue>
```

```
  <firstarg type="variable">id</firstarg>
  <secondarg type="variable">publishedIn</secondarg>
 </mapping>
 ...
```

DS-2 contains publications indexed by CiteSeer including title, author, year and citation, etc. We create mappings for DS-2 as well. Samples from DS-2's mapping configuration are shown as follows:

```
...
 <!-- paper  -->
 <mapping>
  <predicate>paper-title</predicate>
  <pathGlue>citeseer/document</pathGlue>
  <firstarg type="variable">key</firstarg>
  <secondarg type="variable">title</secondarg>
 </mapping>
 <mapping>
  <predicate>paper-author</predicate>
  <pathGlue>citeseer/document</pathGlue>
  <firstarg type="variable">key</firstarg>
  <secondarg type="variable">author</secondarg>
 </mapping>
 <mapping>
  <predicate>paper-conference</predicate>
  <pathGlue>citeseer/document</pathGlue>
  <firstarg type="variable">key</firstarg>
  <secondarg type="variable">conference</secondarg>
 </mapping>
 <mapping>
  <predicate>paper-cite</predicate>
  <pathGlue>citeseer/document</pathGlue>
  <firstarg type="variable">key</firstarg>
  <secondarg type="variable">citation</secondarg>
 </mapping>
 ...
```

DS-3 stores SIGMOD's organizations and publications. DS-3 provides mappings for each binary relation: `proceeding-title`, `proceeding-year`, `proceeding-pc`, and `proceeding-gc`, etc.

```
...
 <!-- proceeding -->
 <mapping>
   <predicate>proceeding-title</predicate>
   <pathGlue>sigmod/conf</pathGlue>
   <firstarg type="variable">id</firstarg>
   <secondarg type="variable">title</secondarg>
 </mapping>
 <mapping>
   <predicate>proceeding-year</predicate>
   <pathGlue>sigmod/conf</pathGlue>
   <firstarg type="variable">id</firstarg>
   <secondarg type="variable">year</secondarg>
 </mapping>
 <mapping>
   <predicate>proceeding-gc</predicate>
   <pathGlue>sigmod/conf</pathGlue>
   <firstarg type="variable">id</firstarg>
   <secondarg type="variable">org/general-chair</secondarg>
 </mapping>
 <mapping>
   <predicate>proceeding-pc</predicate>
   <pathGlue>sigmod/conf</pathGlue>
   <firstarg type="variable">id</firstarg>
   <secondarg type="variable">org/program-chair</secondarg>
 </mapping>
...
```

`Coordinator-2` is in charge of Source `uwashington`, Source `dblp`, Source `digreview`, and Source `submissions`. Each source provides mappings for the SM view as sources do in `Coordinator-1`.

5.3.2 User Queries

The user queries we evaluated are shown in Figure 5.6. The first seven queries are from [65]; Q8 and Q9 are our extra queries.

The full queries are described at a high level here and are provided in full in Figure 5.7 as relational queries.

Q1: Find all XML related projects. This requires a search for projects whose area contains a substring "XML". The binary relation `project-area` is used for Q1. The *local-join graph* is restricted to the only one relation `project-area`, so that it contains a

```
Q1: Find all XML related projects
Q2: Find all projects a given person was involved in
Q3: Find all co-authors of a given researcher
Q4: Find all papers by a given pair of authors
Q5: Find papers written by researchers who lead an XML project
Q6: Find Jayavel Shanmugasundaram's paper on VLDB'99
Q7: Find PC chairs of recent conferences and their papers
Q8: Find people at UC Berkeley and their home pages
Q9: List persons who are PC chairs at a conference both in 2003 and in 2004.
```

Figure 5.6: Queries for the DB-research experiments.

directed spanning tree. According to Theorem 2, no inter-source processing is needed for this query and all subqueries in the *subqueries* approach (see Section 4.2) are local subqueries.

Q2: Only binary relation `project-member` can satisfy this query where the member is a given person (for example, "Alon Halevy"). Similarly, no inter-source processing is needed for this query with only one relation.

Q3: Here we perform a join between `paper-author` and `paper-author`. As `paper` is not the key of `paper-author`, the *local-join graph* restricted to these two relations doesn't contain a directed spanning tree. Therefore, we can't eliminate inter-source processing for Q3 according to Theorem 2. For example, one of inter-source subqueries is made up of `paper-author` in `berkeley` join `paper-author` in `citeseer`.

Q4: Find all papers by a given pair of authors. This query is similar to Q3, and also contains a join between `paper-author` and `paper-author`. Inter-source processing is needed due to Q4's *local-join graph* without a directed spanning tree.

Q5: Find papers written by researchers who lead an XML project. For this query, we join `project-leader` and `paper-author` together. Unfortunately, there is no *local-join property* between them and the *local-join graph* restricted to `project-leader` and `paper-author` doesn't contain a directed spanning tree, so that inter-source subqueries are necessarily generated for Q5.

Q6: Find someone's paper on the specified conference. `paper-conference` and `paper-author` are used in Q6. Note that `paper-conference.paper` is the key and `paper-conference`

and `paper-author` have the *local-join property*. Q6's *local-join graph* contains a directed spanning tree, so no inter-source processing is mandatory for Q6's evaluation.

Q7: `proceeding-pc`, `paper-author`, and `proceeding-year` are involved in Q7. Since we can't generate a directed spanning tree for the *local-join graph* restricted to these three relations, we can't eliminate inter-source processing for Q7. However, if we apply the technique of partitioning inter-source subqueries described in Section 4.8 to Q7's processing, only some of inter-source subqueries are mandatory to be evaluated, since some redundant subqueries could be removed from Q7's evaluation.

Q8: Find people at UC Berkeley and their home pages. This query needs a join of `person-affiliation` and `person-homepage`. Similar to Q3, inter-source processing is needed for this query.

Q9: Finally, we include a little complex query which contains four binary relations. The local-join graph restricted to the four query relations contains a directed spanning tree, so no inter-source processing is needed for this query according to Theorem 2.

5.4 Scenario II: XML.org

Our second set of experiments used schemas from XML.org. We generated the data for these schemas, since the original data are no longer available on the Internet. As our experiments compare *relative* performance of our algorithms, the comparison is valid no matter what data are used.

In this experiments, we created one coordinator, and under this coordinator, three sources: `customers` (buyers and sellers), `xcbl1` (E-commerce), and `papi1` (Accounting). Our semantic model for the XML.org dataset is shown in Figure 5.8. The queries are shown in Figure 5.9; the last query is our extra query.

5.5 Experimental Results on the SM Approach

Now that we have described our two benchmark datasets and the queries that we run over them, we compare and study their performance in seven different algorithms:

- The *materialization* technique generates the relations in the user query at the coordinator, and executes the query on the materialized data.
- The *subqueries* technique executes *all* local and inter-source subqueries, and merges the results at the coordinator.
- The *subqueries** technique is the semantically optimized version of the *subqueries* technique.
- The *optimized-subqueries* technique is similar to the subqueries approach, except that it avoids executing redundant inter-source subqueries.
- The *optimized-subqueries** technique is the improved *optimized-subqueries* technique where local and inter-source subqueries are semantically optimized.
- The *wrapper* technique extracts the minimum required information from each source, combines them at the coordinator, and applies the chase.
- The *wrapper** technique is the semantically optimized version of the *wrapper* technique.

All the experiments were executed using PostgreSQL [59] version 8.1.3 and SAXON [61] version 8 on a 2.0 GHz Pentium M computer with 768 MB memory and 40 GB hard disk running Windows XP Pro.

5.5.1 Semantic Optimization

As described in Section 4.5, we propose a new algorithm that rewrites XQuery queries into more efficient equivalent queries with fewer joins and variables. Instead of creating one variable for each binary relation in the user query, the new algorithm generates a single variable for all binary relations that have a common "glue" variable in their mappings and the same key.

Example 12 *When Q3 in Section 5.3.2 is processed by the subqueries approach, one variable is created in the XQuery query for each binary relation in Q3. A local subquery at Source berkeley for Q3 is the following:*

```
FOR $g0 in /berkeley/direction/project/paper,
  $g00 in $g0/id, $g01 in $g0/author,
  $g1 in /berkeley/direction/project/paper,
  $g10 in $g1/id, $g11 in $g1/author
```

```
WHERE $g00 = $g10 AND $g01 = 'Alon Halevy' AND $g11 != 'Alon Halevy'
RETURN <author> {$g11} </author>
```

After semantic optimization is applied, that's, Q3 is processed by the subqueries approach, only one variable is generated for two binary relations in Q3 that have a common "glue" variable (*`/berkeley/direction/project/paper`*) and the same key (*`id`*). As a result, an optimized version of this local subquery at Source berkeley turns to be:*

```
FOR $br IN /berkeley/direction/project/paper[author='Alon Halevy']
LET $coauthor := $br/author
WHERE $coauthor != 'Alon Halevy'
RETURN <author> {$coauthor} </author>
```

■

5.5.2 Results

Our experimental results for the DB-Research dataset are shown in Figures 5.10-5.11 and Figures 5.13-5.14, and the results for the XML.org dataset are shown in Figure 5.12.

The horizontal (X) axis of each of Figures 5.10-5.14 lists the queries, and the vertical (Y) axis — the respective execution times in milliseconds. (The 9 queries used in the DB-Research experiments are shown in Figure 5.6, and the 6 queries used in the XML.org experiments are shown in Figure 5.9.)

Figures 5.10-5.11 show the results of four algorithms without semantic optimization. From Figures 5.10-5.11 and Figure 5.12, we know that although the optimized-subqueries technique seems to be the algorithm of choice for most cases, it is not always the most efficient. In fact, we should not write off any of the algorithms. Rather, we expect that, depending on the information sources and the user query, any of the algorithms may outperform the other three.

In Section 4.5, we proposed a new algorithm for subquery generation in the *subqueries* approach, the *optimized-subqueries* approach and the *wrapper* approach. Figures 5.13 and 5.14 illustrate the performance of the semantically optimized versions of these three algorithms for the DB-Research dataset experiments. The semantically optimized versions, shown in the graphs as `subqueries*, optimized subqueries*` and `wrapper*,` apply the semantic optimization to the queries then execute them.

As shown in the figures, the semantically optimized versions perform much better,

and the `subqueries*` and `optimized-subqueries*` techniques seem to be the algorithms of choice for most cases.

For example, Q3 and Q4, which used to take 7.648 and 4.859 seconds respectively on the *Subqueries* approach, now take about two seconds (see Table 5.1) when semantic optimization is applied.

Table 5.1: Query times (in milliseconds) for Q3 and Q4 after semantic optimization is applied. %faster = $\frac{100|original-new|}{original}$.

	Q3	Q4
Subqueries	2301 (69.9%)	2251 (53.7%)
Optimized Subqueries	2906 (61.6%)	2108 (56.0%)
Wrapper	4806 (71.7%)	1769 (69.2%)

5.5.3 Observations

Our experimental environment may have had an impact on the results, which should be carefully considered when interpreting these results. For XML processing, which included local and inter-source subqueries, and the subqueries for the extraction of minimum required information in the wrapper technique, we used XQuery on SAXON [61]. In fact, the entire wrapper algorithm, including chase and duplicate elimination, is implemented in XQuery. On the other hand, for the materialization technique, we used PostgreSQL [59]. This seems to have penalized the efficiency for the wrapper and, to a lesser degree, for the subqueries approach, while favoring materialization.

We have experimentally established that our proposed methods are very competitive, although, in some cases, the basic materialization method turns out to be the best. The *subqueries* approach is, in general, more efficient than *materialization*. The *optimized subqueries* technique is more efficient than the other methods without semantic optimization for most (but not all) user queries. This is witnessed, in particular, in Q6 of the DB-Research experiments as well as in most XML.org experiments. Despite the handicap of the XQuery-engine inefficiencies, the *wrapper* technique has the best performance for some of the XML.org experiments. Our conjecture is that when data is very regular at every source, in the sense that each source contributes to all relations in a user query, then the *wrapper* method is the most efficient. For queries where semantic constraints enable us to eliminate all or most of the inter-source processing, the *optimized subqueries* is expected to

outperform the other methods. *Materialization* is preferred for queries with large number of relations where all or most inter-source subqueries are required. Finally, *subqueries* method is preferable for queries with small to moderate number of relations. If we take semantic optimization into consideration, the semantically optimized versions perform much better, and the *optimized subqueries** techniques seem to be the algorithms of choice for most cases.

From the result analysis in Section 5.5.2, although the *optimized subqueries** technique seems to be the best algorithm for most cases, it is not always the most efficient. Many aspects have an impact on the experimental results. What we should consider are the followings: data sources, user queries and experimental tools such as SAXON, PostgreSQL and XMLBeans. If we want to choose the best approach dynamically, we must investigate and classify the experimental environments. The following observations comes from both Scenario I and Scenario II.

The *materialization* approach may run good under one or more of the following rules:

- Small number of unique binary relations in the user query.
- Small size of data in each source.
- The user query with some "like" clauses.

As for the subqueries-based approaches, e.g., the *subqueries* or *subqueries** approach and the *optimized subqueries* or *optimized subqueries** approach, they may be:

- Large size of data in each source.
- The user query with only 2 to 6 binary predicates.
- The query without any "like" clause.
- Few required inter-source queries for the Semantic Model in the *optimized subqueries* or *optimized subqueries** approach.

The *wrapper* or *wrapper** approach could be better when the environment matches one or more of the followings:

- Small size of local results before chase steps.
- Chase steps are not needed, according to our theorems.

- Few chase steps are needed, when less key constraints and less equalities in the user query.
- Each source has all predicates that appear the user query. As a result, less nulls are in local results.

```
Q1: select project-area.project from project-area
    where project-area.project like '%XML%'
Q2: select project-member.project from project-member
    where project-member.member = 'Alon Halevy'
Q3: select p2.author from paper-author as p1, paper-author as p2
    where p1.paper = p2.paper and p1.author = 'Alon Halevy' and
    p2.author != 'Alon Halevy'
Q4: select p1.paper from paper-author as p1, paper-author as p2
    where p1.paper = p2.paper and p1.author = 'Alon Halevy' and
    p2.author = 'Zack Ives'
Q5: select paper-author.paper from paper-author, project-leader
    where paper-author.author = project-leader.leader and
    project-leader.project like '%XML%'
Q6: select paper-conference.paper from paper-conference,
    paper-author where paper-conference.paper = paper-author.paper and
    paper-author.author ='Jayavel Shanmugasundaram' and
    paper-conference.conference = 'VLDB99'
Q7: select proceeding-pc.pc, paper-author.paper from proceeding-pc,
    paper-author, proceeding-year where proceeding-pc.pc =
    paper-author.author and proceeding-pc.proceeding =
    proceeding-year.proceeding and proceeding-year.year like '%200%'
Q8: select person-affiliation.person, person-homepage.homepage from
    person-affiliation, person-homepage where person-affiliation.person
    = person-homepage.person and person-affiliation.affiliation =
    'UC Berkeley'
Q9: select pp1.pc from proceeding-pc as pp1, proceeding-pc as pp2,
    proceeding-year as py1, proceeding-year as py2 where pp1.proceeding
    = py1.proceeding and pp2.proceeding = py2.proceeding and pp1.pc =
    pp2.pc and py1.year = 2003 and py2.year = 2004
```

Figure 5.7: Relational-like queries for the DB-research experiments.

```
[person]                [order]              [item]
person-name             order-buyer          item-description
person-address          order-seller         item-price
person-phone            order-price          item-qty
                        order-item
                        order-itemNum
```

Figure 5.8: Semantic model for the XML.org schemas.

```
Q1.  Find all customers
Q2.  Find the parts supplied by a given supplier
Q3.  Find customers of a given supplier
Q4.  Find suppliers who sell a given part
Q5.  Find customers who also supply parts
Q6.  Find customers' ids and phones of a given supplier
```

Figure 5.9: Queries for the XML.org experiments.

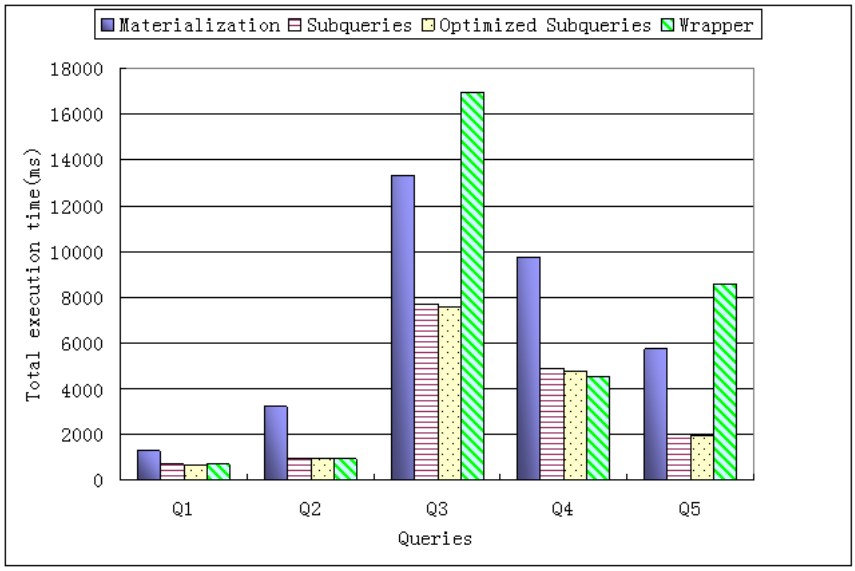

Figure 5.10: Experimental results for the DB-Research dataset. All query times are given in milliseconds.

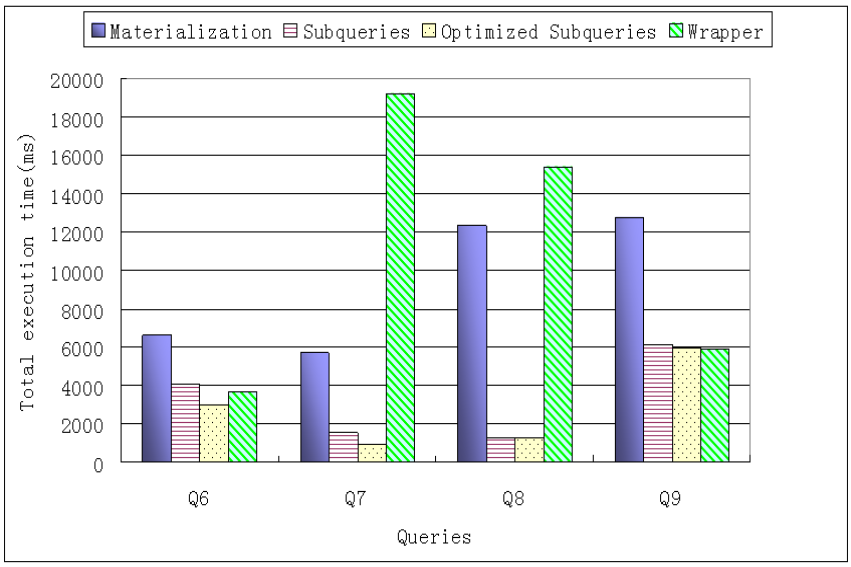

Figure 5.11: Experimental results for DB-Research (continued). All query times are in milliseconds.

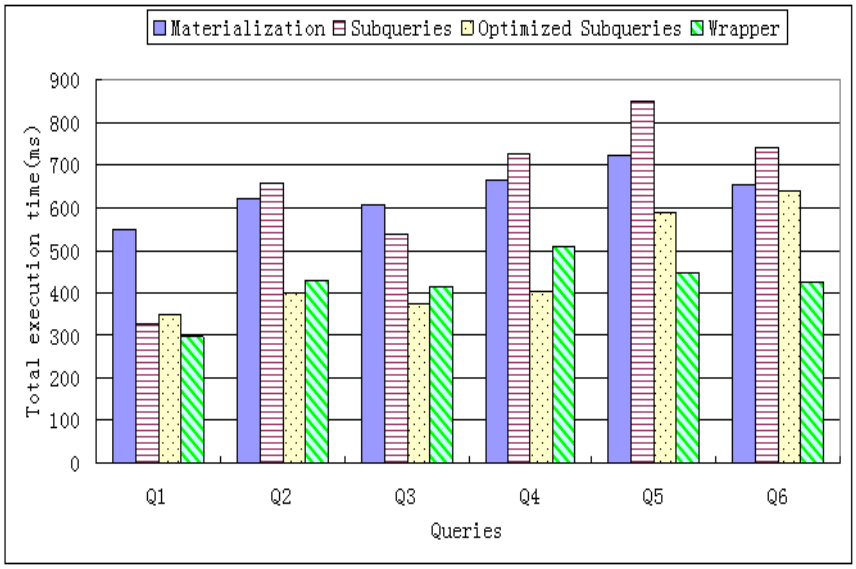

Figure 5.12: Experimental results for the XML.org dataset. All query times are given in milliseconds.

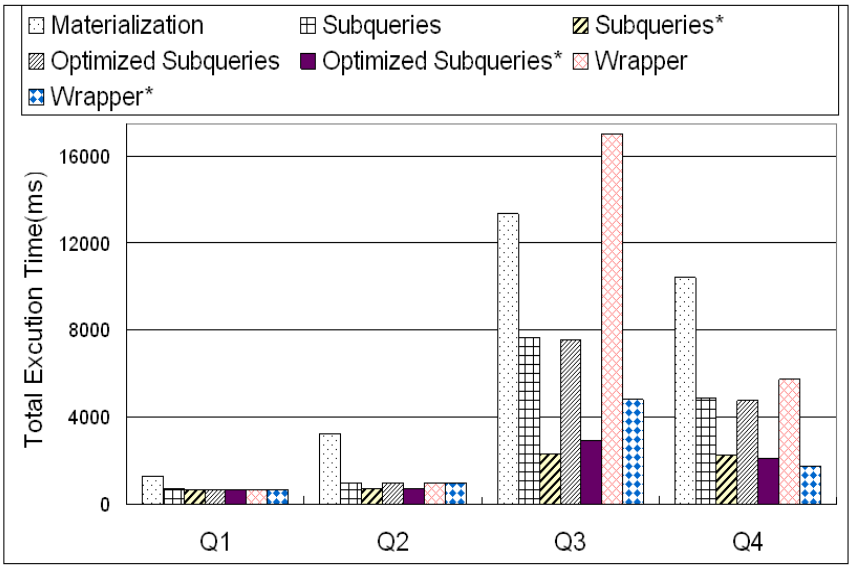

Figure 5.13: Semantic optimization in the DB-Research dataset. All query times are given in milliseconds.

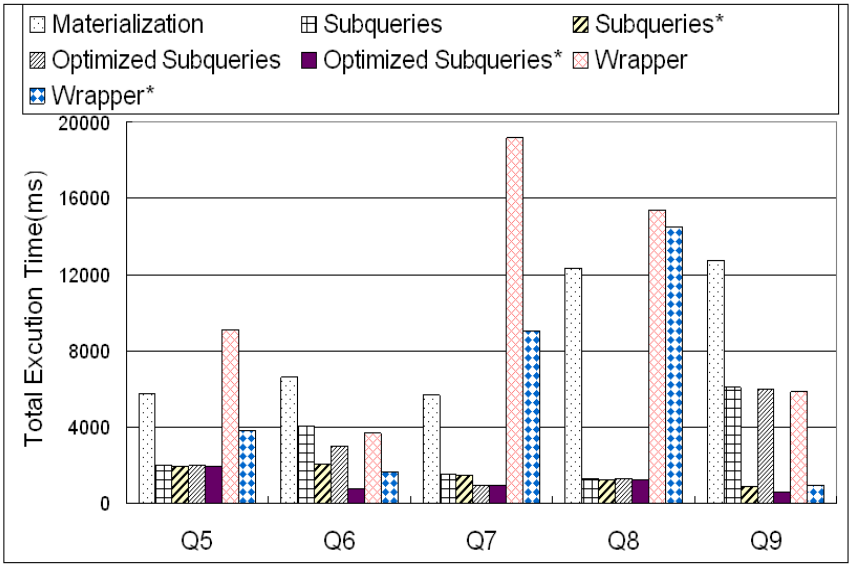

Figure 5.14: Semantic optimization in DB-Research (continued). All query times are in milliseconds.

Chapter 6

Restructured Views: Preliminaries

Classical query optimization using restructured views can result in significant additional savings in query-evaluation costs, and some commercial database systems are adding facilities for the definition and materialization of restructured views. In this chapter, we first present the background of restructured views (Section 6.1), and then propose our approach to define restructured views and rewrite queries using restructured views in the remainder of this chapter. Section 6.2 provides some definitions and presents examples for restructured views. In Section 6.3, we present conditions for determining when a (regular or restructured) view is usable in answering a SQL query, and develop algorithms for rewriting queries in terms of usable views. We develop these results for select-project-join queries and views without aggregation, as well as for queries and views involving group by and aggregation. Next, we focus on our ultimate goal of combining query optimization using regular and restructured views into a single framework for relational databases. As such, materialized restructured views should be stored as relations, and should participate in SQL query evaluation. In Section 6.4 we present our approach to enhancing an SQL system with query optimization using restructured views. In Section 6.5 we present techniques to derive such semantic information, specifically the primary-key constraints, for a restructured view. In Section 6.6 we show how to use integrity constraints on restructured views to further optimize rewritten queries. Finally, we carry out experiments to evaluate the effectiveness of query optimization using restructured views, and illustrate that our approach is simple and easily understood in Section 6.7.

In the next chapter (Chapter 7), we will incorporate the concept of materialized *restructured views* into query processing and optimization in the semantic-model approach.

6.1 Background and Motivation

We study optimization of relational queries using materialized views, where views may be regular or *restructured.* In a restructured view, some data from the base table(s) are represented as metadata — that is, schema information, such as table and attribute names — or vice versa. Often, data from one or multiple tables can be restructured into a new representation with significant reduction in the overall size of data. Intuitively, by moving data values that appear frequently and repeatedly in a relation to metadata in the view, we can represent the information in the view more compactly than by using a regular view.

Consider, for example, the `salesInfo` relation of Figure 6.1, which lists stores and their monthly sales information over a number of years. Figure 6.2 shows a restructured view of `salesInfo.` This `salesInfoView` relation represents the same information as `salesInfo,` but the months (`Jan`, `Feb`, ... , `Dec`) now play the role of attribute names, and the sales values for the months of each year are organized "horizontally" into a single tuple for each store. Note that each tuple of the view `salesInfoView` represents the same information that is represented by 12 tuples of the base table `salesInfo`. Figure 6.3 shows another view, `annualSalesView,` which involves aggregation and restructuring. Each tuple of `annualSalesView` represents the complete annual sales values for each store for all years.

storeID	year	month	sales
1	2005	Jan	120000
1	2005	Feb	100000
...	...	...	...
1	2005	Dec	150000
2	2005	Jan	300000
...	...	...	...

Figure 6.1: Base relation salesInfo.

storeID	year	Jan	Feb	···	Dec
1	2005	120000	100000	···	150000
2	2005	300000	···	···	···

Figure 6.2: Restructured view salesInfoView.

storeID	···	2003	2004	2005	···
1	···	1500000	1650000	1770000	···
2	···	3000000	2870000	2800000	···

Figure 6.3: Restructured view annualSalesView.

Using restructured views in query optimization opens up a new spectrum of views that were not previously available, and can result in significant additional savings in query-evaluation costs. The savings can be obtained due to a significantly larger set of views to choose from, and may involve reduced table sizes, elimination of self-joins, clustering produced by restructuring, and horizontal partitioning.

Commercial database systems are adding facilities for the definition and materialization of restructured views [16]. Our goal is to tap into this new capability for query optimization. Semantic information, such as knowledge of the key of a view, can be used to further optimize a rewritten query. We consider the problem of determining the key of a (regular or restructured) view, and show how this information can be used to further optimize a rewritten query.

Restructuring operations *unfold*, *fold*, *split*, and *unite* were introduced in [29]; the first two operations are also known in database literature as *pivot* and *unpivot*, respectively. Combined with relational algebra, these operations form an extended algebra capable of querying data and metadata uniformly, and of defining restructured views within the relational data model [48].

Consider the relation `salesInfo` of Figure 6.1. The *unfold* operation takes two attributes (column names) as parameters. Performing the operation $unfold_{month,sales}$ on `salesInfo` generates the `salesInfoView` relation shown in Figure 6.2. The values in column *month* of `salesInfo` are moved to schema (attribute names) in the view, and corresponding values in column *sales* are organized (horizontally) as values for these newly formed attribute names. The *fold* operation is the converse of *unfold*.

The *split* operation takes one attribute as parameter, and performs a horizontal partitioning according to the values of the specified attribute. Each partition, minus the specified attribute column, is made into a new relation, and the value of the specified attribute is used as the name of the relation. The *unite* operation is the converse of *split*.

Restructuring operations together with regular aggregation and group by opera-

tions can define a broad spectrum of views that enhance regular views with restructuring. Figure 6.3 shows one such view, `annualSalesView`. It represents aggregated annual sales values for stores in a restructured view that has years as attributes (column names).

Using the full spectrum of regular and restructured views for query optimization can result in a significant reduction in query-processing costs over using only regular views. The extra efficiency is due to two factors:

1. Information can be represented more compactly in restructured views, hence reducing the *size* (number of disk blocks) of the relation, which can benefit all queries.

2. Restructuring amounts to different *physical clusterings* of data. Some classes of queries can be processed more efficiently as a result of the clustering.

The following example demonstrates some of the advantages of using materialized restructured views in query optimization.

Example 13 (Query Optimization Using Materialized Restructured Views) We have shown a regular relation, `salesInfo`, and two restructured views, `salesInfoView` and `annualSalesView`, in Figures 6.1, 6.2, and 6.3, respectively. View `salesInfoView` pivots sales amounts by month, that is, it uses months as attributes and records sales figures of the months of each year "horizontally". View `annualSalesView` aggregates (sums) the sales amounts for the months of each year, and then pivots the sum by year. The effect of restructuring should be evident in these views: Each tuple of `salesInfoView` records the same information that is represented by twelve tuples of `salesInfo`, and thus the size of the relation for `salesInfoView` will be smaller than that of `salesInfo` (generally two to four times smaller). Similarly, each tuple of `annualSalesView` records the same information that is represented by n tuples of a regular view that aggregates monthly sales for each year, where n is the number of years sales data are available in the base table `salesInfo`. Again, `annualSalesInfo` is, in general, smaller than a regular view that does the same aggregation. Size reduction in this case may amount to one or two orders of magnitude. (Note that `salesInfoView` clusters monthly sales for each year into one tuple, while `annualSalesView` clusters all annual sales for each store into a single tuple.)

Now consider the following queries:

Q1. List stores that have a sales volume of at least one million dollars in January of 2006.

Q2. List stores that show a rapid decline (10% or more per month) in the fourth quarter of 2005.

Q3. List stores that doubled their annual sales between 1995 and 2005.

Q4. List hardware stores that doubled their annual sales between 1995 and 2005.

These four queries can all be expressed in SQL; for instance, a SQL definition for Q1 is

```
Q1: SELECT storeID FROM salesInfo WHERE year = 2006
    AND month = 'January' AND sales >= 1000000;
```

Q1 is a simple query whose evaluation requires a full scan of the table `salesInfo` in the absence of an index on month or year. It can be processed more efficiently using the restructured view `salesInfoView` due to its smaller size.

Evaluating the query Q2 can benefit from the clustering that `salesInfoView` provides, as the view clusters the information of interest (sales amounts for the fourth quarter of 2005) into a single tuple for each store. Evaluating Q2 on the base table `salesInfo` requires two self-joins, which can be eliminated using `salesInfoView`; in addition, the size of the view is smaller than the size of `salesInfo`. As a result, the savings in evaluation costs for Q2 using `salesInfoView` can amount to an order of magnitude or more. We discuss Q2 in detail in Example 22 in Section 6.6.

Q3 can benefit from the clustering `annualSalesView` provides. Similarly to Q2, evaluating Q3 using a restructured view is more efficient than evaluating it on a regular view that aggregates monthly sales amounts for each year.

While similar to Q3, Q4 has an additional restriction to only one type of store: hardware stores. The second view of Example 17 (Section 6.2.1) is usable in answering this query. Note that this view is basically a horizontal partitioning, on the *store type* attribute, of a restructured view that lists stores and their annual sales (such as `annualSalesView`). The size of the materialized view for `hardware` stores can be orders of magnitude smaller than the size of the base table or of a regular view, due to horizontal partitioning as well as to representing in a single tuple all annual-sales figures for a store. Hence, our general framework also accommodates optimization techniques based on horizontal partitioning. (Compare this to the current practice that deals with horizontal partitioning separately from query optimization using materialized views.)

■

6.2 Restructured Views: Definitions and Examples

Traditionally, the relational model distinguishes between the *schema* (*intention*) and *data* (*extension*). But in a restructured view, data values can play the role of schema elements — such as attribute and relation names — and vice versa. (As an illustration, consider the two relations in Figures 6.1 and 6.2.) We use the following definitions to accommodate restructured views as an extension of the relational model.

Definition 2 (Flexible Schema; Fact) A *flexible schema* (*schema*, for short) is a pair (A, D) where A is a set of attributes, and D is a set of values (a *domain*). A schema is *regular* if it has attributes only.

Let $S = (A, D)$, $A = \{A_1, \ldots, A_n\}$, $D = \{d_1, \ldots, d_m\}$, be a flexible schema. A *fact* r on the schema S is a statement of the form $p(A_1 : u_1, \ldots, A_n : u_n, d_i : v)$, where p is a predicate symbol, $u_1, \ldots, u_n, v$ are values, and $d_i \in D$. ■

Our formalism makes it convenient to discuss in a uniform way views (regular and restructured), view definitions, view usability, and query rewriting (Sections 6.2.1 and 6.3). This is achieved by separating the issue of representing data from that of relational (tabular) storage of data. Our ultimate goal, however, is to use these results for query optimization in a practical setting in an SQL system, where materialized views are stored and accessed as relations. Hence, in this section we also address relational storage of restructured views. To accommodate relational (tabular) representation, we borrow from [47] the notion of *molecule*. A molecule combines multiple facts into a single representation.

Definition 3 (Molecule) Let $S = (A, D)$, $A = \{A_1, \ldots, A_n\}$, $D = \{d_1, \ldots, d_m\}$, be a flexible schema. A *partial molecule* m on the schema S is a statement of the form $p(A_1 : u_1, \ldots, A_n : u_n, e_1 : v_1, \ldots, e_k : v_k)$, where p is a predicate symbol, $u_1, \ldots, u_n, v_1, \ldots, v_k$ are values, and $e_1, \ldots e_k \in D$, $e_i \neq e_j$, $i, j = 1, \ldots, k$. We denote $\{e_1, \ldots, e_k\} \subseteq D$ by D_m and call it the *domain* of the molecule m. A molecule m is *total* if $D_m = D$, *i.e.*, every value of the domain D appears in the molecule. Note that a fact is (a special case of) a molecule. Given a molecule m, we write $m(A_i)$ (or $m(d_i)$) to refer to A_i (or d_i) component of m. We use $m(A)$ as a shorthand for $m(A_1), \ldots, m(A_n)$. ■

Example 14 (Regular and Restructured Relations) Consider a regular schema `salesInfo(storeID,year,month, sales)` and the relation of Figure 6.1 on `salesInfo`. This relation is represented by the following facts:

salesInfo(storeID: 1, year: 2005, month: Jan, sales: 120000)

...

salesInfo(storeID: 1, year: 2005, month: Dec, sales: 150000)

salesInfo(storeID: 2, year: 2005, month: Jan, sales: 300000)

...

Now consider a restructured view `salesInfoView` that "unfolds" (or "pivots") `salesInfo` on month. The tabular representation of this view is shown in Figure 6.2. In our abstraction, each tuple of this relation is represented by twelve facts. For example, the first tuple is represented as follows:

salesInfoView(storeID: 1, year: 2005, Jan: 120000)

...

salesInfoView(storeID: 1, year: 2005, Dec: 150000)

The same tuple is represented by the following molecule:

salesInfoView(storeID: 1, year: 2005, Jan: 120000, Feb:100000, ..., Dec:150000)

■

We need the following definitions to bridge the gap between facts and molecules:

Definition 4 (Mergeable Molecules) Two molecules m_1 and m_2 on the schema $S = (A, D)$, $A = \{A_1, \ldots, A_n\}$ are *mergeable* if (i) $m_1(A) = m_2(A)$, and (ii) $D_{m_1} \cap D_{m_2} = \phi$. The *merge* of two mergeable molecules m_1 and m_2, written $m_1 \oplus m_2$, is a molecule m on S where $m(A) = m_1(A)$, $D_m = D_{m_1} \cup D_{m_2}$, and for all $d \in D_m$, $m(d) = m_1(d)$ if $m_1(d)$ is defined, otherwise $m(d) = m_2(d)$. ■

Definition 5 (Maximal Molecule) Given a molecule m and a set of molecules M, we say m is *maximal* with respect to M if there is no molecule in M that is mergeable with m. A set of molecules M is called *maximal* if every molecule in M is maximal with respect to M. ■

Definition 6 (Canonical Representation) Given a set F of facts on the schema $S = (A, D)$, we define a *canonical (molecular) representation* of F as a set of maximal molecules obtained by repeatedly merging facts of F. ■

In general, a given set of facts F on a schema $S = (A, D)$ can have more than one canonical representation. But if A is the key of S, then the canonical representation is unique:

Lemma 2 *(*Uniqueness of Canonical Representation*) A set of facts F on a schema $S = (A, D)$ has a unique canonical representation if A is the key for S.*

Proof: We can construct a canonical representation for F by (1) partitioning facts F according to their A values, and (2) merging the facts in each partition. Since A is the key, each partition can have at most one fact with a value for a given $d \in D$. Hence, all facts in a partition can be merged into a single molecule. The set of molecules obtained in this way is maximal since each has a different A value. ■

Given a set of facts F on schema $S = (A, D)$, a canonical representation M of F can be represented in relational (tabular) form. Each molecule $m \in M$ is represented by a tuple on the (regular) schema $(A_1, \ldots, A_n, d_1, \ldots, d_k)$, where $\{A_1, \ldots, A_n\} = A$ and $\{d_1, \ldots, d_k\} = D$. If molecule m is partial, it should be padded with a special value for missing values in D to make it into a full tuple. We denote this special (padding) value by π. The reason we use this special value instead of a null is to be able to distinguish between nulls that may exist in the base relation and the padding values used to represent partial molecules by full tuples. In this way, the relational representation is lossless, meaning that we can obtain the original set of facts from the relational representation.

Example 15 (Relational Representation) Suppose a store with *storeID* 567 was opened in February 2003. It would have sales figures for February through December of 2003, but no sales figure for January 2003. Its canonical representation for `salesInfoView` will have the following partial molecule (plus possibly total and partial molecules for 2004 and later years).

salesInfoView(storeID: 567, year: 2003, Feb: 300000, Mar: 330000, ..., Dec: 400000)

The corresponding relational representation will have the following tuple for this store for 2003:

storeID	year	Jan	Feb	Mar	$\cdots$	Dec
567	2003	π	300000	330000	$\cdots$	400000

6.2.1 View Definition

We use *S-LOG*, a Prolog-like language, to define views in our framework. In this subsection we first present some examples, then give a brief discussion of *S-LOG*.

Example 16 (View Definition) The view definition for `salesInfoView` in Example 14 is the following *S-LOG* rule (uppercase letters represent variables):

salesInfoView(storeID: I, year: Y, M: S) ←

salesInfo(storeID: I, year: Y, month: M, sales: S)

Note that while `salesInfo` has the regular schema (`storeID, year, month, sales`), `salesInfoView` has the (flexible) schema ({`storeID, year`}, `D`), with domain `D = {Jan, Feb, ..., Dec}`. This is achieved by using variable *M*; *M* is bound to data (months `Jan, Feb, ...`) in the body, while playing the role of an attribute name in the head of the rule. ■

Example 17 (Restructured spj views) This example demonstrates that *S-LOG* can combine select-project-join (SPJ) view capability with a variety of aggregation and restructuring operations. Suppose relation `storeType(store ID, type)` specifies a type (*e.g.*, grocery, hardware, etc.) for each store. The following definition generates multiple relations, one per store type. It is basically a partitioning of `storeInfoView` of Example 16 according to store type.

T(storeID: I, year: Y, M: S) ←

salesInfo(storeID: I, year: Y, month: M, sales: S), storeType(storeID: I, type: T)

As another example, consider view definition

T(storeID: I, Y: sum(S)) ←

salesInfo(storeID: I, year: Y, month: M, sales: S), storeType(storeID: I, type: T)

It generates multiple relations, one per store type. Each relation lists for each store all total annual sales figures in a single tuple, arranged horizontally. ■

Introduction to *S-LOG*

S-LOG is a simplified version of *schemaLOG* [47]. The main difference between *S-LOG* and *schemaLOG* is that *schemaLOG* facts contain a *tuple-id* component while *S-LOG* facts do

not, which makes *S-LOG* simpler and closer to the relational model. Atoms of *S-LOG* correspond to facts on (flexible) schemas (see Definition 2), where components can be constants or variables. View definitions use *S-LOG* rules, defined as follows:

Definition 7 An *S-LOG rule* has the form $h(x) \leftarrow b(x, y)$, where (1) $h(x)$, the *head* of the rule, is a single *S-LOG* atom containing the set of variables x, and (2) $b(x, y)$, the *body* of the rule, is a conjunction of *S-LOG* atoms and comparison conditions containing the set of variables x and y. The set of variables x in the head of the rule is called *distinguished* variables. Note that we require the "safety condition" — that is, in an *S-LOG* rule, distinguished variables must also appear in the atoms of the body. ■

Semantics of *S-LOG*

The valuation-based semantics of *schemaLOG* [47] is applicable to *S-LOG* as well. For example, given the fact

salesInfo(storeID: 1, year: 2005, month: Jan, sales: 120000)

the instantiation of the definition for `salesInfoView` of Example 16 binds variables I, Y, M, and S to 1, 2005, Jan, and 120000, respectively, which generates this fact in the view:

salesInfoView(storeID: 1, year: 2005, Jan: 120000)

It is easy to see that applying the view-definition rule of Example 16 to `salesInfo` facts produces the `salesInfoView` facts represented in Example 14.

Aggregation

A query or view definition can involve aggregation. To accommodate aggregation, we extend *S-LOG* rules by allowing aggregate functions in the head of the rules. We adopt the convention of [15, 56] that an implicit "*group by*" by the non-aggregated variables in the head is applied to the query or view definition.

Example 18 (Aggregation) This definition

annualSalesInfo(storeID: I, year: Y, annualSales: sum(S)) ←
salesInfo(storeID: I, year: Y, month: M, sales: S)

of a *regular* view is equivalent to the SQL view definition

```
CREATE VIEW annualSalesInfo(storeID,year,annualSales) AS SELECT
storeID, year, SUM(sales) FROM salesInfo GROUP BY storeID, year;
```

Now consider this definition of a *restructured* view:

annualSalesView(storeID: I, Y: sum(S)) ←

salesInfo(storeID: I, year: Y, month: M, sales: S)

Similarly to `annualSalesInfo`, `annualSalesView` performs a group by storeID I and year Y, and aggregates the sales figures. But the schema of `annualSalesView` is ({`storeID`}, `D`), where domain `D` contains the years that appear in the base relation `salesInfo`. A tabular representation of `annualSalesView` has the schema (`storeID,...,2003,2004,...`). That is, it lists the annual-sales figures "horizontally" in columns labeled by the year. Figure 6.3 depicts this view. Note that each tuple of the table in Figure 6.3 represents the information of n tuples of the regular view `annualSalesInfo`, where n is the number of years for which sales information is available in the base table `salesInfo`. ■

6.3 View Usability and Query Rewriting

We would like to extend view usability and query optimization using materialized views [64] to restructured views. This provides the database system (and the DBA) with a significantly larger set of views to choose from for materialization and utilization in query optimization, which can result in significant improvements in query-answering performance. In this section we concentrate on view usability and query rewriting for restructured views. As regular views are special cases of restructured views, our results form the basis for a unified approach for query optimization using materialized regular and restructured views in database systems. We formulate our view-usability results first for unaggregate queries and views (Section 6.3.1), and then for aggregate queries and views (Section 6.3.2), with aggregate functions `SUM, COUNT, MAX,` and `MIN`.

6.3.1 Unaggregate Queries and Views

Let Q be a query in *S-LOG*, and V be a view defined by an *S-LOG* rule r. Let V_r, V_Q, and C_Q denote the set of variables in r, the set of variables in Q, and the set of constants in Q, respectively. The following definitions, from [8], apply equally to our framework:

Definition 8 (Valid Renaming) A *valid renaming* σ of r with respect to Q is a mapping from V_r to $V_Q \cup C_Q$, $\sigma : V_r \rightarrow V_Q \cup C_Q$, such that

(1) If $v \in V_r$ is a distinguished variable (see Definition 7), then $\sigma(v) \in V_Q \cup C_Q$.

(2) If $v \in V_r$ is a non-distinguished variable, then $\sigma(v) \in V_Q$, and $\sigma(v) \neq \sigma(v')$ for any other variable $v' \in V_r$. ■

The purpose of renaming is to make the body of the view (with the exception of comparison predicates) to become equal to (part of) the body of the query.

Definition 9 (Safe Occurrence; Safe Substitution) Given a view V (defined as an *S-LOG* rule r) and a query Q, we say Q has a *safe occurrence* of V if there is a valid renaming of r with respect to Q, such that

1. The renamed rule has the form $V(x) \leftarrow L(x,y), I(x)$, where $I(x)$ is a conjunction of comparison conditions on (possibly a subset of) variables in x. Note that $L(x,y)$ may contain comparison conditions on variables in y (but not x).

2. The query Q has the form $Q(u) \leftarrow L(x,y), I'(x), G(v)$, where y is disjoint from x, v, and u.

3. Conditions $I'(x)$ of the query logically imply conditions $I(x)$ of the view: $I'(x) \Rightarrow I(x)$.

The *safe substitution* corresponding to the above safe occurrence is: $Q'(u) \leftarrow V(x), I'(x), G(v)$. ■

We can now state the following result for *S-LOG* queries and views without aggregation.

Theorem 4 (View Usability) *Let Q and V be select-project-join* S-LOG *query and view, respectively. Then view V is* usable *for query Q if Q has a safe occurrence of V, in which case Q can be rewritten into a (multi-set [7]) equivalent query that uses the view V in its body. The rewritten query is the safe substitution mentioned above.*

Proof: Let Q be a query that has a safe occurrence of view V. Then Q has the form $Q(u) \leftarrow L(x,y), I'(x)$, $G(v)$, and V can be renamed as $V(x) \leftarrow L(x,y), I(x)$. Consider query $Q'(u) \leftarrow V(x), I'(x), G(v)$, which is the safe substitution of V in Q. We show $Q \equiv Q'$ by showing a 1:1 correspondence between the facts generated for Q and those generated for

Q'. Suppose that values x_1, y_1, and v_1 in facts corresponding to L and G generate a fact $Q(u_1)$ by valuation $Q(u_1)\leftarrow L(x_1, y_1), I'(x_1), G(v_1)$. Then, since $I'(x) \implies I(x)$, conditions $I(x_1)$ are also true, and $V(x_1)$ is generated by $V(x_1) \leftarrow L(x_1, y_1), I(x_1)$ in the view V. Hence $Q'(u_1)$ will be generated for the query Q' by $Q'(u_1)\leftarrow V(x_1)$, $I'(x_1), G(v_1)$.

Conversely, assume values x_1, y_1, and v_1 generate $V(x_1)$ and $Q'(u_1)$ by instantiations $V(x_1)\leftarrow L(x_1, y_1), I(x_1)$ and $Q'(u_1)\leftarrow V(x_1), I'(x_1), G(v_1)$, respectively. Then they also generate $Q(u_1)$ by $Q(u_1)\leftarrow L(x_1, y_1), I'(x_1), G(v_1)$. ■

Theorem 4 gives us a sound algorithm (that is, each output of the algorithm is valid) for obtaining equivalent rewritings of SQL queries using restructured views. Note that the soundness result holds (1) for all of SQL select-project-join queries and of regular or restructured select-project-join views, all possibly with arithmetic comparisons, and (2) under each of set, bag (*i.e.*, multi-set [7]), and bag-set [7] semantics for query evaluation.

6.3.2 Queries and Views with Aggregation

We now extend our approach to rewriting SQL queries with aggregation `SUM`, `COUNT`, `MAX`, or `MIN` using restructured views. We begin by giving a motivating example.

Example 19 Consider a SQL query Q "Return the sum of sales per store since the beginning of the year 2003." In *S-LOG*, Q can be defined as

q(storeID: I, since2003sales: sum(S)) ←
salesInfo(storeID: I, year: Y, month: M, sales: S), Y ≥ 2003

We can evaluate the query Q by using the following two equivalent rewritings of Q, R_1 and R_2, which use restructured views. Rewriting R_1 uses view `salesInfoView`, and rewriting R_2 uses view `annualSalesView`; see Examples 16 and 18 for the definitions of these two restructured views.

r_1*(storeID: I, since2003sales: sum(S)) ←*
salesInfoView(storeID: I, year: Y, M: S), Y ≥ 2003

r_2*(storeID: I, since2003sales: sum(S)) ←*
annualSalesView(storeID: I, Y: S), Y ≥ 2003

As discussed above, using restructured view `salesInfo View` instead of `salesInfo` may result in a reduction in evaluation costs for Q. Moreover, using instead the *aggregate* restruc-

tured view `annualSalesView` may result in further significant reductions in the evaluation costs for the query. ■

Our goal in this subsection is to provide an analog of Theorem 4 for aggregate SQL queries and restructured views with and without aggregation. These results will allow us to obtain equivalent rewritings of aggregate queries using restructured views, such as rewritings R_1 and R_2 in Example 19. Our approach, which is an extension of the contributions of [15, 56] to restructured views, is twofold. First, we define rules (Lemma 3) that tell us when an aggregate rewriting using a restructured view is equivalent to its expansion in terms of the base relations. As has been shown in [15], such equivalence-to-expansion problems need to be addressed essentially on a case-by-case basis, by defining and using rewriting templates that allow only certain types of views in the body of the rewriting and only certain group-by and aggregation attributes in the head of the rewriting.

Second, we use our rules to reduce the problem of determining equivalence of an aggregate query to a rewriting to the problem of determining equivalence of the query to the expansion of the rewriting (Theorem 5). The latter problem has been solved in [56] for queries with aggregate functions `SUM, COUNT, MAX,` and `MIN`.

We now formulate Lemma 3 that spells out our rules that define when an aggregate rewriting using a restructured view is equivalent to its expansion in terms of the base relations. (In this subsection we use the term "rewriting" to denote queries defined in terms of views.) Our rewriting templates are an extension of central rewritings of [2] to restructured views. Intuitively, in central rewritings the unaggregated *core* is a SPJ query, and only one view (the *central view*) contributes to computing the aggregated query output. Both rewritings in Example 19 are central rewritings. Central rewritings are a natural choice in many applications; for instance, in the star-schema framework [6] the fact table provides the aggregate view, and the dimension tables provide the other views in the rewritings. In our analysis, we chose central rewritings for their simplicity. As there is a natural relationship between some classes of rewritings in [2] and in [15], our results also apply to rewritings of [15].

Lemma 3 uses the notion of an expansion R^{exp} of a central rewriting R. We obtain R^{exp} by (1) replacing all view literals in R by the bodies of their definitions, using fresh variables for non-distinguished variables as usual, and by (2) replacing the aggregate function in the head of R by the aggregate function of the central view. (As a special case, if

the central view is unaggregated, the heads of R and R^{exp} use the same aggregate function.) For instance, the expansion of rewriting R_2 in Example 19 is defined as

r_2^{exp}*(storeID: I, since2003sales: sum(S))* $\leftarrow$

salesInfo(storeID: I, year: Y, month: M, sales: S), $Y \geq 2003$

Lemma 3 (Rewriting Templates) *Let R be a regular central rewriting with aggregation* `SUM, COUNT, MAX`*, or* `MIN` *that uses no more than one restructured view and zero or more regular views. Then $R^{exp} \equiv R$ provided that (1) R is a* `MAX` *or* `MIN` *rewriting, or the following rule is observed: (2) In case R is a* `SUM` *or* `COUNT` *rewriting, all noncentral views are unaggregated and are evaluated using bag projection.*

Proof: The key intuition behind restructured aggregate views is that they are just a different tabular representation such as based on a pivot operation behind some regular aggregate view. (For an example, consider view annualSalesView.) Thus, the result of the lemma follows directly from the results in [2] on aggregate rewritings with regular views, and from Theorem 4 on view usability for restructured views. Note that rewriting R can be viewed as the result of doing a safe substitution of the central view of R into the expansion R^{exp} of R. ■

We refer to rewritings that satisfy the equivalence test of Lemma 3 as *admissible aggregate rewritings.* Note that the rules of Lemma 3 work for rewritings that use restructured (and regular) views, as well as for rewritings that use only regular views. Thus, the role of Lemma 3 in our general framework is to extend the applicability of central rewritings of [2] to restructured as well as regular views.

We now use the rules of Lemma 3 to reduce the problem of determining equivalence of an aggregate query Q to a rewriting R to the problem of determining equivalence of Q to the expansion R^{exp} of R. As we have already mentioned, once the reduction has been done we can decide whether Q is equivalent to R^{exp} (and thus whether Q is equivalent to R) by using the results of [56].

Theorem 5 *Let Q and V be select-project-join* S-LOG *query and view, respectively, such that Q has aggregation* `SUM, COUNT, MAX,` *or* `MIN`*, and V may or may not have aggregation. Then V is* usable *for Q if (1) Q has a safe occurrence of V, and (2) the result R of rewriting Q using V is an admissible aggregate rewriting. If both (1) and (2) are satisfied, the rewriting R is equivalent to the query Q.*

Proof: The result of the theorem is straightforward from the definition of central rewritings [2], from Theorem 4, and from Lemma 3. ■

Theorem 5 gives us a sound algorithm for equivalently rewriting aggregate queries using regular and restructured views. Note that the soundness result holds for all of SQL aggregate select-project-join queries and of SQL regular or restructured select-project-join views (possibly with aggregation), all possibly with arithmetic comparisons. In particular, by Theorem 5 both rewritings (R_1 and R_2) of Example 19 are equivalent to the query Q in the example. As a result, each of R_1 and R_2 can be used to reduce the evaluation costs of the query Q on a relational database where the views used in the rewritings (`salesInfoView` in R_1 and `annualSalesView` in R_2) are materialized as stored relations.

6.4 SQL Optimization

In this section we address the additional details needed to apply our results in an SQL database system. In this setting, all materialized views — including restructured views — should be stored and accessed as relations, and queries (including rewritten queries) should be definable in SQL. The additional capability required is the ability to define and materialize restructured views, which has been incorporated into some commercial database systems [16].

Given an SQL query, we can use our usability results of Section 6.3 to rewrite the query with respect to a usable view. But the resulting query may not be expressible in SQL. The reason is that *S-LOG* is more powerful than SQL in its ability to treat data and metadata (schema information) uniformly. To make sure the rewritten query is expressible in SQL, the variables in $V(x)$ in the body of the rewritten query should only correspond to data. That is, no variable should appear in $V(x)$ in the position of a schema element (attribute name, relation name). Or, if a variable v does appear in a metadata position, then a condition of the form $v = constant$ should also exist in the rewritten query. The following result guarantees that the rewritten query is in SQL. Let V be a restructured view defined by the *S-LOG* rule r, and Q be an SQL query. Let V_r be the set of variables in the rule r, and $V_r^m \subseteq V_r$ be those variables in the head of r that are in metadata positions.

Theorem 6 (SQL Usability) *V is SQL-usable in answering Q if it is usable in answering Q with respect to a valid renaming σ, and, further, for all variables $v \in V_r^m$, if $\sigma(v)$ is a*

variable, then the rewritten query contains a condition $\sigma(v) = c$ for some constant c, or such a condition is implied by the conditions of the rewritten query.

Proof: Let Q' be the rewritten query w.r.t. the renaming σ. For each variable $v \in V_r^m$, we replace $\sigma(v)$ in Q' by the constant c from $\sigma(v) = c$, and remove any condition that involves $\sigma(v)$. The result is an *S-LOG* query with variables in data positions only, which is expressible in SQL. ■

Finally, query rewriting needs one additional detail in an SQL database setting. Conditions should be added to the rewritten query to ensure that "padding" values π do not impact the answer to a query.

We notice that padding values can be generated in the relational representation of a view only when the view-definition rule has a variable in an attribute-name position in the head of the rule. Let Q be an SQL query, and V be an SQL-usable view for Q. Let $V(x)$ be the head of the rule defining the view V, and x^a be the set of variables that appear in attribute-name positions in $V(x)$. Let σ be the renaming function used for testing SQL-usability and for rewriting the query, and let Q' be the rewritten query according to Theorem 4 or to Theorem 5.

Example 20 (SQL Query Rewriting) Consider the relational representation of the view `salesInfoView` defined on the base relation `salesInfo` (Example 16). The query "List maximal sales for the month of January for each store for their sales since 2000" is written as follows:

maxJanSales(storeID: I, janSales: max(S)) ←
salesInfo(storeID:I,year:Y,month:'Jan',sales:S), Y>1999

The view `salesInfoView` is SQL-usable in answering this query. According to Theorem 5, the rewritten query is:

maxJanSales(storeID: I, janSales: max(S)) ←
salesInfoView(storeID: I, year:Y, Jan: S), Y > 1999

The rewritten query for the relational representation is obtained by adding the condition $S \neq \pi$ to the body:

maxJanSales(storeID: I, janSales: max(S)) ←
salesInfoView(storeID:I,year:Y,Jan:S), Y>1999, $S \neq \pi$

Or, in SQL:

```
SELECT storeID, MAX(Jan) FROM salesInfoView WHERE year>1999 AND
Jan<>padding-value GROUP BY storeID
```

6.5 Deriving Integrity Constraints in Restructured Views

Integrity constraints, such as key and foreign-key constraints, play an important role in database design and in query optimization. They become even more important for query optimization using materialized restructured views. As we will discuss in Section 6.6, knowledge of integrity constraints for restructured views can be used for certain classes of queries to further optimize the rewritten query with significant improvements in the query-evaluation efficiency.

In this section we concentrate on the problem of deriving integrity constraints in views. Two factors determine integrity constraints in views: (1) the view definition, and (2) integrity constraints in the corresponding base relations.

6.5.1 Integrity Constraints in Aggregate Views

The group-by operation of aggregate views naturally generates key constraints in the view. For example, consider the views `annualSalesInfo` and `annualSalesView` of Example 18. The key of `annualSalesInfo` is `(storeID, year)`, as a result of grouping by these two attributes. The case of `annualSalesView`, which is a restructured view, is somewhat different. Here, the group-by is also by `storeID` and `year`. However, `year` (or rather the variable that is bound to `year`), is used as attribute name in the aggregation part of the head of the view-definition rule. The key of `annualSalesView` is the single attribute `storeID`, see Figure 6.3.

We can state the following result regarding integrity constraints in aggregate views. Let an aggregate view V be defined by an *S-LOG* rule. The head of the rule has the form *view-name(target)*, where *target* contains non-aggregation components of the form *att:var* and aggregation components of the form *att:agg-function(var)*, where *agg-function* is an aggregate function such as `MIN, MAX, SUM,` etc.

Theorem 7 (Key of an Aggregate View) *Let view V be as above. The key of V is the set of attributes that appear in non-aggregation components in the head of the view-definition rule.* ■

We should note that attribute field (*att* in *att:var*) can be a constant or a variable. By "set of attributes" in Theorem 7 we mean the union of (1) the set of constant attributes, and of (2) the domain of the variable attribute (if any) in the non-aggregation components in the head. In Example 18, the view definition for `annualSalesView` has only one (constant) attribute, `storeID`, in the non-aggregation components of its head. Hence, `storeID` is the key of `annualSalesView`.

6.5.2 Constraints Implied by Base Relations

Given key constraints on base tables, or, more generally, a set of functional dependencies (FDs) on base tables, we would like to derive the constraints that hold in a restructured view defined by an *S-LOG* rule r. We will use an *S-LOG*-based declaration for FDs. For example, consider the relation `salesInfo(storeID, year, month, sales)`. Suppose that `(storeID, year, month)` is the key of `sales Info`, that is, the FD `storeID, year, month` $\rightarrow$ `sales` holds in `salesInfo`. We express this FD as follows:

salesInfo(storeID: I, year: Y, month: M, sales: S),
salesInfo(storeID: I, year: Y, month: M, sales: S') $\Rightarrow$ S = S'

which is stating that two *salesInfo* facts with the same values for `storeID`, `year`, and `month` must also have the same value for `sales`. This representation is suitable for our flexible-schema framework. For example, consider the view `salesInfoView` of Example 16. The above FD is expressed as follows for `salesInfoView`:

salesInfoView(storeID: I, year: Y, M: S),
salesInfoView(storeID: I, year: Y, M: S') $\Rightarrow$ S = S'

For certain restructured views, integrity constraints implied by FDs in base tables cannot be expressed as FDs. Yet, our *S-LOG* formalism allows us to represent these constraints. In the rest of this section, we briefly discuss the closed-world semantics of view-definition rules, then present a logical process to derive integrity constraints in views.

Semantics of View-Definition Rules

We follow the *closed-world* semantics for view-definition rules. This is the usual semantics for query processing and query optimization using materialized views. In contrast, the *open-world* semantics has been used for data-integration applications. Interested readers

are referred to [30] for a comprehensive discussion.

Intuitively, the closed-world semantics states that a view defined in *S-LOG* consists of the facts derived by the rule and nothing else. We can represent this semantics for a view V defined as $v(x) \leftarrow b(x, y)$ by the logical statement

$$\forall x(v(x) \implies \exists y(b(x, y)))$$

The following example demonstrates our logical method for verifying integrity constraints that hold in a view.

Example 21 Consider the view `salesInfoView` of Example 16. The corresponding base relation is `salesInfo(store ID, year, month, sales)`. Suppose that `(storeID, year, month)` is the key of `salesInfo`.

We can verify that the following FD holds in the view:

salesInfoView(storeID: I, year: Y, M: S),
salesInfoView(storeID: I, year: Y, M: S') $\implies$ S = S'

We observe that, by the closed-world semantics:

salesInfoView(storeID: I, year: Y, M: S),
salesInfoView(storeID: I, year: Y, M: S') $\implies$
salesInfo(storeID: I, year: Y, month: M, sales: S),
salesInfo(storeID: I, year: Y, month: M, sales: S')

But, since (`storeID, year, month`) is the key for `salesInfo`:

salesInfo(storeID: I, year: Y, month: M, sales: S),
salesInfo(storeID: I, year: Y, month: M, sales: S') $\implies$ S = S'

Hence, we have verified the following FD in the view:

salesInfoView(storeID: I, year: Y, M: S),
salesInfoView(storeID: I, year: Y, M: S') $\implies$ S = S' ■

6.6 Optimizing Rewritten Queries

In Section 6.5 we discussed how to determine the key of a view. This knowledge can be used to optimize rewritten queries, with a potential for significant improvements in the evaluation efficiency for certain classes of queries.

As we saw in Section 6.1, the added efficiency due to restructured views is possible because (1) a restructured view can represent the information of a regular view in a much smaller relation, and (2) restructured views naturally perform *clustering* of data by organizing many tuples of a regular view into a single tuple, or by horizontal partitioning. Optimization of rewritten queries discussed in this section allows us to maximize the benefit from the second factor.

We demonstrate this further optimization by the following example. The optimizer uses semantic knowledge (key constraints) to determine that information needed for answering a user query is clustered within single tuples, and hence, it is able to eliminate costly self-joins:

Example 22 (Optimization of Rewritten Queries) Consider a query "List stores that show a rapid decline in sales (10% or more per month) in the fourth quarter of 2005." Using the relation `salesInfo`, the query is:

decliningStores(storeID: I) ←
  salesInfo(storeID: I, year: 2005, month: 'Oct', sales: S1),
  salesInfo(storeID: I, year: 2005, month: 'Nov', sales: S2),
  salesInfo(storeID: I, year: 2005, month: 'Dec', sales: S3),
  $S2 \leq 0.9 \times S1$, $S3 \leq 0.9 \times S2$

The view `salesInfoView` is SQL-usable for this query. The rewriting is applied three times to replace the three occurrences of `salesInfo` in the body with `salesInfoView`. The rewritten query is:

decliningStores(storeID: I) ←
  salesInfoView(storeID: I, year: 2005, Oct: S1),
  salesInfoView(storeID: I, year: 2005, Nov: S2),
  salesInfoView(storeID: I, year: 2005, Dec: S3),
  $S2 \leq 0.9 \times S1$, $S3 \leq 0.9 \times S2$

In the absence of key constraints, the rewritten query for relational representation of `salesInfoView` view is the same as the query above with the addition of constraints $S1 \neq \pi$, $S2 \neq \pi$, and $S3 \neq \pi$. The rewritten query is more efficient than the original query due to the reduced size of the view `salesInfoView` (which is, in general, two to four times smaller than `salesInfo`), resulting in an evaluation time possibly 6 to 12 times faster than

the original query, assuming linear time for join operation.

However, the optimizer also knows that `(storeID, year)` is the key for `storeInfoView` (see Example 21). Then the rewritten query can be further optimized into:

decliningStores(storeID: I) ←

salesInfoView(storeID:I, year:2005, Oct:S1, Nov:S2, Dec:S3),

$S1 \neq \pi$, $S2 \neq \pi$, $S3 \neq \pi$, $S2 \leq 0.9 \times S1$, $S3 \leq 0.9 \times S2$

The reason is that the three atoms in the body of the first rewritten query all have the same values for (`storeID, year`), which is the key for `salesInfoView`. Hence, these atoms are represented by a *single* tuple in the view. In relational terms, this optimization eliminates (multiple) self-joins of `salesInfoView`, further reducing the execution time by a factor of 3 or more, for an overall reduction of 18 to 36 times in execution time. Note that this additional optimization becomes possible by the knowledge of the key for the restructured view, which was determined by techniques discussed in the previous section.

■

6.7 Experiments on Restructured Views

We have carried out extensive experiments to evaluate the effectiveness of query optimization using restructured views. All the experiments were executed on a 2.0 GHz Pentium M computer with 768 MB memory and 40 GB hard disk running Windows XP Pro. To improve the accuracy of the results, each query was executed 12 times, then the smallest and largest execution times were eliminated, and the remaining 10 execution times were averaged to obtain the final results.

In the first set of experiments, we used the `salesInfo` database (Section 6.7.1) introduced in Section 6.1. We present the second set of experiments on the `Stocks` database in Section 6.7.2. In Section 6.7.3, we repeated our experiments with the `salesInfo` database using the IBM DB2 to evaluate the effects of our proposed optimization approaches using restructured views with a commercial-strength database.

6.7.1 Scenario III: SalesInfo

In this subsection we report the results of our experiments that were run using PostgreSQL [59] version 8.1.3. In the first set of experiments we used the salesInfo database

schema introduced in Section 6.1. Different database sizes were generated for 5000, 10000, and 20000 stores, corresponding to database sizes 69MB, 137MB, and 275MB, respectively. Although these databases are not too large, the experiment results clearly demonstrate the relative merits of using restructured views in query optimization. We added two queries to the four listed in Example 13. The queries are listed below for convenience. The results of our experiments are summarized in Table 6.1.

Q1. List stores that have a sales volume of at least one million dollars in January of 2006.

Q2. List stores that show a rapid decline (10% or more per month) in the fourth quarter of 2005.

Q3. List stores that doubled their annual sales between 1995 and 2005.

Q4. List hardware stores that doubled their annual sales between 1995 and 2005.

Q5. List sales of storeID "500" for January 2000.

Q6. List annual sales of storeID "500" for the year 2000.

Table 6.1: Results of our PostgreSQL experiments using the SalesInfo databases.

Query	5K stores (69MB)		10K stores (137MB)		20K stores (275MB)		average speedup
Q1	748.7	128.3	1325.8	245.0	2506.0	358.8	6
Q2	2178.1	118.2	4443.3	218.2	8690.9	247.4	25
Q3	1740.9	32.9	3450.4	67.1	6725.6	138.5	51
Q4	1259.7	6.7	2492.3	13.9	4799.6	27.8	180
Q5	683.6	107.4	1250.8	220.4	2336.5	338.5	6
Q6	88.3	4.8	192.2	9.4	333.6	19.0	19

Discussion:

In Table 6.1 two execution times are shown for each query and each database size. The first value is the execution time using materialized regular views. The second value shows the execution time using materialized restructured views. The last column, labeled "speedup", shows the average ratio of execution time using regular view to the execution time using restructured view. Hence, Q1 executes 6 times faster on the average when restructured views are used, Q2 executes 25 times faster, and so forth.

Queries Q1 and Q5 are simple, involving a scan of `salesInfo` table (for regular view) or a scan of the restructured view `salesInfoView` (See Figures 6.1 and 6.2). The speedups for these queries are due to the reduced size of `salesInfoView`, which is about 6 times smaller than `salesInfo`. Query Q6 is similar, except it scans the regular view `annualSales`, which aggregates sales annually and has the schema `(storeID, year, annualSales)`, versus the restructured view `annualSalesView` of Figure 6.3. In this case the restructured view is about 19 times smaller than the regular view, resulting in a speedup of 19. Queries Q2 and Q3 benefit from restructured views in two ways: First the smaller sizes of restructured views (compared to corresponding regular views) makes query processing more efficient. In addition, the *clustering* archived by restructuring, and elimination of self joins discussed in Section 6.6, increase the efficiency of the rewritten query using restructured views. The speedups are 25 and 51 times, respectively. Finally, query Q4 demonstrates the impact of the above two factors, in combination with the well-known horizontal partitioning. In this case the restructured view is further partitioned on the `storeType` attribute. The combined effects result in the query to run 180 times faster than when only regular views were used.

6.7.2 Scenario IV: Stocks

For the second set of experiments, we wrote code to download historical stock information from *Yahoo! Finance* (http://finance.yahoo.com/) for various stocks (tickers) over a period of 20 years and to upload it into a PostgreSQL database. The URL
http://itable.finance.yahoo.com/table.csv?a=0&b=1&c=1987&d=10&e=24&f=2006&g=d&s=ibm
illustrates the structure of the specific URLs we used for the downloads; the parameters specify the start and end dates and the stock (ticker) ID. In the above URL the dates are 1987/01/01 to 2006/10/24 and the stock ID is "IBM". The information was loaded in a relation `stocks(year, month, day, ticker, price, priceType)`. Price type is one of "open, close, high, low". Different database sizes were obtained by downloading stock prices for 20, 40, 60, and 80 tickers. Database sizes for these experiments are not large either (largest table, for 80 tickers, was slightly over 100 MB), but the experiments still demonstrate the relative improvement in query processing speed when materialized restructured views are used. For each database size, a number of regular and restructured views were materialized and used in query evaluation.

The following queries were used in these experiments:

Q'1. List the closing price of IBM stock on Nov. 21, 2006.

Q'2. List the average closing price of IBM stock for October 2006.

Q'3. List stocks (tickers) that had a high value 20% or more than their low value on a day in 2005.

Q'4. List dates in 2005 when IBM had a closing value 50% or more than the closing value of Microsoft.

Q'5. List stocks (tickers) that show a rapid decline in the fourth quarter of 2005 (their average closing value drops at least 10% in November and December 2005 compared to previous month.)

The results of our experiments are summarized in Table 6.2.

Table 6.2: Results of our PostgreSQL experiments using the Stocks databases.

Query	20 tickers		40 tickers		60 tickers		80 tickers		speedup
Q'1	366.7	22.4	572.9	40.1	803.3	44.6	992.1	46.8	17
Q'2	5.5	0.6	10.5	1.1	15.4	1.6	20.9	2.2	9
Q'3	661.0	91.7	1252.9	213.7	1823.4	280.2	2385.2	355.8	7
Q'4	554.5	24.1	978.2	41.9	1404.2	46.9	1823.1	49.3	28
Q'5	14.0	0.6	28.2	1.0	42.1	1.5	56.0	2.2	26

6.7.3 Experiments Using IBM DB2

To evaluate the effects of our proposed optimization approaches using restructured views with a commercial-strength database that uses various optimization techniques, we repeated our experiments with the salesInfo dataset using the IBM DB2 database-management system.[1] We generated the databases and executed the queries (see Section 6.7.1 for the details on each) in the following settings:

- Base tables, materialized regular and restructured views with no indexes.

[1]The authors are grateful to the IBM University Relations Division (http://www-304.ibm.com/jct09002c/university/scholars/ur/index.html) for providing DB2 for our experiments.

- Base tables, materialized regular and restructured views with indexes on primary keys.
- Base tables, materialized regular and restructured views with indexes on primary keys and additional secondary indexes on relevant attributes.
- We also used "window queries" for Q3 and Q4 on the base table. Window functions and window queries, introduced in the 1989 and 2003 SQL standards [40], facilitate the formulation of data-analysis queries. Commercial databases have incorporated special optimization techniques for window queries.

These experiments were executed using IBM DB2 9.1 on an IBM x340 Pentium III 997 MHz, with 3GB RAM and 108GB SCSI disk, running Red Hat Enterprise Linux 5 (RHEL5).

Table 6.3: IBM DB2 experiments (best performances) for the salesInfo databases.

Query	5K stores (69MB)		10K stores (137MB)		20K stores (275MB)		average speedup
Q1	816	78	1692	170	3291	337	10
Q2	1148	133	2338	264	4599	552	9
Q3	850	25	2201	50	4437	98	41
Q4	342	5.4	467	11	1368	20	58
Q5	0.62	0.54	0.77	0.64	0.79	0.69	1.1
Q6	0.78	0.51	0.84	0.58	1.07	0.61	1.6

Our results are summarized in Table 6.3. We report the best performance obtained for each of regular views and restructured views. For Q3 and Q4, the best performance for regular views was obtained when we used window queries. All other queries achieved their best performance with indexed (both base and view) tables. The performance gains for Q1-Q4 are similar to those obtained in our PostgreSQL experiments (see Section 6.7.1), with speedups of an order of magnitude and higher. For Q5 and Q6, the speedup is smaller, with 10% to 60% gain. These experiments clearly show that adding query optimization using restructured views to a commercial database-management system has the potential of significantly improving query-processing performance for some queries.

Chapter 7

Query Optimization Using Restructured Views

The experimental results in Section 6.7 show that classical query optimization using restructured views can result in significant additional savings in query-evaluation costs. In this chapter we incorporate the concept of materialized *restructured views* into query processing and optimization in the semantic-model approach, and present a suite of algorithms for efficient query processing in presence of materialized restructured views.

In Section 7.1 we describe our eighth query processing approach: *restructured view*. Note that we have discussed the following seven query processing approaches: *materialization* (Section 4.1), *subqueries* (Section 4.2), *optimized subqueries* (Section 4.3), *wrapper* (Section 4.4), *subqueries** (Section 4.5.1), *optimized subqueries** (Section 4.5.2), and *wrapper** (Section 4.5.3). We present the experimental setup and restructured views in DB-Research in Section 7.2 and Stocks in Section 7.3. Our experimental results in Section 7.4 demonstrate that using restructured views may result in orders-of-magnitude improvement in query-processing time for certain classes of queries.

7.1 Query Processing VIII: Restructured View

In the *restructured view* approach, we materialize restructured views in coordinators, rewrite user queries on the semantic-model view to equivalent queries on restructured views, and then execute them over restructured views. In the remainder of this section we discuss our approach to integrating restructured views into our semantic-model framework,

and address the following questions:

- How are the restructured views represented in the semantic-model approach?
- What are the queries that materialize restructured views?
- How do we rewrite user queries using restructured views?

7.1.1 Defining restructured views

Restructured views can be defined by S-LOG mappings described in Section 6.2.1 from the mediator schema. For example, the `stocksByTicker` view (see Section 7.3 for the background) is defined as:

```
stocksByTicker (year:Y,month:M,day:D,priceType:R,T:P) <-
k-ticker(k:K,ticker:T),k-year(k:K,year:Y),k-month(k:K,month:M),
k-day(k:K,day:D),k-priceType(k:K,priceType:R),k-price(k:K,price:P).
```

In order to further reduce the size of restructured view, we decompose it into a set of restructured views, one for each ticker: `stocksByIBM`, `stocksByMSFT`, and so on. The s-LOG rule for `stocksByIBM` is as follows:

```
stocksByIBM (year:Y,month:M,day:D,priceType:R,ibm:P) <-
k-ticker(k:K,ticker:‘ibm’),k-year(k:K,year:Y),k-month(k:K,month:M),
k-day(k:K,day:D),k-priceType(k:K,priceType:R),k-price(k:K,price:P).
```

Similarly, if we take `priceType` (whose value is one of “open”, “close”, “high” and “low”) and `price` as parameters for the pivot operation, we get a restructured view `stocksByPriceType (ticker,year,month,day,open,close,high,low)`. We can also define *aggregate* restructured views. As an example, consider the following view definition that stores the monthly average closing prices for the tickers.

```
monthlyAvgByTicker (year:Y,month:M,T:avg(P)) <-
k-ticker(k:K,ticker:T), k-year(k:K,year:Y), k-month(k:K,month:M),
k-priceType(k:K,priceType:‘close’), k-price(k:K,price:P).
```

7.1.2 View materialization

Once restructured views are defined (e.g., by the database administrator), queries are needed for materializing the view answers. The algorithm that generates queries for view materialization composes the mappings from each data source to the mediator schema

with the mapping from the mediator schema to the restructured view; the latter can be obtained automatically from S-LOG view definitions.

As an illustration, for the view `stocksByIBM` defined above one would obtain the query:

```
<stocksByIBM>{
FOR $br in /stocks/stock WHERE $br/ticker = 'IBM'
RETURN
  <stock>
    {$br/year}
    {$br/month}
    {$br/day}
    {$br/priceType}
    <ibm>{$br/price/text()}</ibm>
  </stock>
}</stocksByIBM>
```

7.1.3 Query rewriting

We now propose an algorithm for taking advantage of restructured views, by using them to rewrite user queries. Algorithms for view usability and query rewriting for materialized restructured views have been presented in Section 6.3, and Algorithm 6 summarizes our approach.

Example 23 *The following user query lists the average closing price for IBM in October 2005:*

```
Q(monthlyAvg:avg(P)) <-
   k-ticker(k:K, ticker:'IBM'), k-year(k:K, year:2005),
   k-month(k:K, month:'Oct'), k-priceType(k:K, priceType:'close'),
   k-price(k:K, price:P)
```

Our proposed algorithm can be used to rewrite Q into a query that uses view `monthlyAvgByTicker`*:*

```
  select t.ibm
  from   monthlyAvgByTicker t
  where  t.year = 2005 and t.month = 'Oct'
```

or into a query that uses view `monthlyAvgByMonth`*:*

```
  select m.oct
  from   monthlyAvgByMonth m
  where  m.year = 2005 and m.ticker = 'IBM'
```

■

Algorithm 6: Query rewriting using restructured views

input : User query Q and set of restructured views V

output: Equivalent query Q' that uses views in V

```
1 foreach view v in V do
2     foreach safe substitution σ from v to Q do
3         if (Q is an unaggregate query) AND (rewriting Q' of Q satisfies
          conditions of Theorem 4) then
4             return Q';
5         else if (Q is an aggregate query) AND (rewriting Q' of Q satisfies
          conditions of Theorem 5) then
6             return Q';
7         end
8     end
9 end
```

7.2 Scenario V: DB-Research with Restructured Views

In Scenario V, we define three restructured views in the DB-Research dataset (see Section 5.3 for the background): `paperByAHalevy`, `paperByVLDB99`, and `proceeding-title-2001-2002-2003-2004`.

The `paperByAHalevy` view represents all information about paper whose author is "Alon Halevy", where horizontal partitioning based on `paper-author.author = 'Alon Halevy'` is performed. The `paperByAHalevy` view is defined as:

```
paperByAHalevy(title:T, conference:C, coauthor:A) <-
     paper-title(paper:P, title:T),  paper-conference(paper:P, conference:C),
     paper-author(paper:P, author:'Alon Halevy'),
     paper-author(paper:P, author:A), A !='Alon Halevy'.
```

The `paperByVLDB99` view contains all publications in VLDB'99, which is defined in S-LOG as:

```
paperByVLDB99(title:T, author:A) <-
   paper-title(paper:P, title:T), paper-author(paper:P, author:A),
   paper-conference(paper:P, conference:'VLDB99').
```

We take *year* and *pc* (program-chair) as parameters for the pivot operation, we get a restructured view `proceeding-title-2001-2002-2003-2004` which is defined as: (note that the value of *year* is between 2001 and 2004.)

```
proceeding-title-YEARS(title:T, Y:PC) <-
    proceeding-title(proceeding:P, title:T),
    proceeding-year(proceeding:P, year:Y),
    proceeding-pc(proceeding:P, pc:PC).
```

Once restructured views are defined, queries are generated for materializing the view answers. For example, the following query is created for the `paperyByAHalevy` view in Source `berkeley`:

```
for $br in /berkeley/direction/project/paper[author='Alon Halevy'],
    $coauthor in $br/author
where $coauthor != 'Alon Halevy'
return
<rs>
  <id>{$br/id/text()}</id>
  <title>{$br/title/text()}</title>
  <coauthor>{$coauthor/text()}</coauthor>
  <conf>{$br/publishedIn/text()}</conf>
</rs>
```

There are four queries (Q3, Q4, Q6, and Q9 in Figure 5.6) which are evaluated using restructured views. The `paperyByAHalevy` view is usable for Q3 and Q4. Q6 can be rewritten by using the `paperByVLDB99` view. The `proceeding-title-2001-2002-2003-2004` view is usable for Q9. The equivalent queries using restructured views for these four user queries are:

```
Q3: SELECT id, coauthor FROM paperByAHalevy
Q4: SELECT id FROM paperByAHalevy WHERE coauthor = 'Zack Ives'
Q6: SELECT title FROM paperByVLDB99 WHERE author = 'Jayavel Shanmugasundaram'
Q9: SELECT title, y2003 FROM proceedingPCByYear WHERE y2003=y2004
```

7.3 Scenario VI: Stocks with Restructured Views

We modified the Stocks dataset described in Section 6.7.2. Instead of loading stock information into a relation `stocks(year, month, day, ticker, price, priceType)`, we

loaded it into two sources with various XML schemas (see Figure 7.1). We also developed a simple semantic-model view based on an ontology for stocks consisting of binary relations `k-ticker`, `k-year`, `k-month`, `k-day`, `k-price`, and `k-priceType`, where `k` is a unique key.

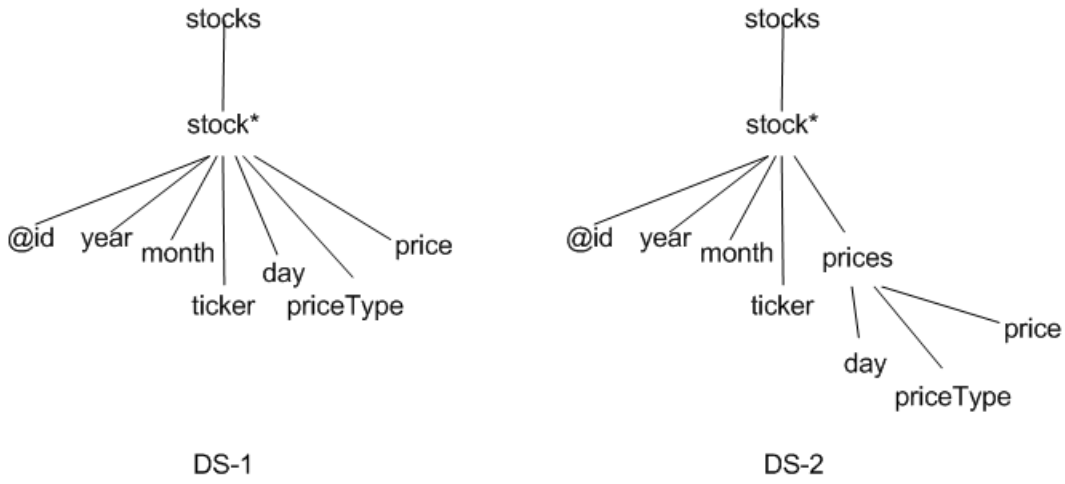

Figure 7.1: Schemas of data sources in Scenario VI.

We materialized four restructured views in this scenario: `stocksByTicker`, `stocksByPriceType`, `monthlyAvgByTicker`, and `monthlyAvgByMonth`. The `stocksByTicker` view, a restructured view which represents the same information as the base relation `Stocks`, but the tickers (`ibm`, `msft`, ... , `dell`) now play the role of attribute names, and the stock values for a given price type are organized "horizontally" into a single tuple for each day. Note that each tuple of the view `stocksByTicker` represents the information represented by many tuples of the base table `Stocks`. For example, if `Stocks` table stores information about 20 different tickers, then each tuple of `stocksByTicker` represents the same information as the corresponding 20 tuples in the `Stocks` table. Similarly, the price types (`open`, `close`, `high`, and `low`) play the role of attribute names in `stocksByPriceType`. `monthlyAvgByTicker` and `monthlyAvgByMonth` are aggregate restructured views.

`stocksByTicker` and `monthlyAvgByTicker` are defined in Section 7.1.1. The S-LOG rules for `stocksByPriceType` and `monthlyAvgByMonth` are as follows:

```
stocksByPriceType (year:Y,month:M,ticker:T,day:D,R:P) <-
k-ticker(k:K,ticker:T),k-year(k:K,year:Y),k-month(k:K,month:M),
k-day(k:K,day:D),k-priceType(k:K,priceType:R),k-price(k:K,price:P).
```

```
monthlyAvgByMonth (year:Y,ticker:T,M:avg(P)) <-
k-ticker(k:K,ticker:T), k-year(k:K,year:Y), k-month(k:K,month:M),
k-priceType(k:K,priceType:'close'), k-price(k:K,price:P).
```

User queries (the first five come from Section 6.7.2) and their equivalent queries using restructured views are listed as follows:

Q1. List the closing price of IBM stock on Nov. 21, 2005.
SQL: SELECT ibm FROM stocksByTicker WHERE year = 2005 AND month = 'Nov' AND day = 21 AND priceType = 'close'

Q2. List the average closing price of IBM stock for October 2005.
SQL: SELECT ibm FROM monthlyAvgByTicker WHERE month = 'Oct' AND year = 2005

Q3. List stocks (tickers) that had a high value 20% or more than their low value on a day in 2005.
SQL: SELECT DISTINCT ticker FROM stocksByPriceType WHERE year = 2005 AND high >= low*1.2

Q4. List dates in 2005 when IBM had a closing value 50% or more than the closing value of Microsoft.
SQL: SELECT month, day FROM stocksByTicker WHERE year = 2005 AND priceType = 'close' AND ibm >= msft*1.5

Q5. List stocks (tickers) that show a rapid decline in the fourth quarter of 2005 (their average closing value drops at least 10% in November and December 2005 compared to previous month.)
SQL: SELECT ticker FROM monthlyAvgByMonth WHERE year = 2005 AND dec >= nov*1.1 AND nov >= oct*1.1

Q6. List the maximum closing price for ibm in October 2005.
SQL: SELECT max(ibm) FROM stocksbyTicker WHERE year = 2005 AND month = 'Oct' AND pricetype = 'close' GROUP BY year, month, pricetype

Q7. List days when IBM's price decreased rapidly(i.e., by 10% or more) two consecutive days.
SQL: SELECT v1.year, v1.month, v1.day FROM stocksbyTicker v1, stocksbyTicker v2 WHERE v1.year = v2.year AND v1.month = v2.month AND v1.day = v2.day-1 AND v1.pricetype = 'close' AND v2.pricetype = 'close' AND v1.ibm = v2.ibm * 1.1

7.4 Experimental Results Using Restructured Views

In this section we present the results of our evaluation of the impact of query optimization using restructured views in our information-integration system shell.

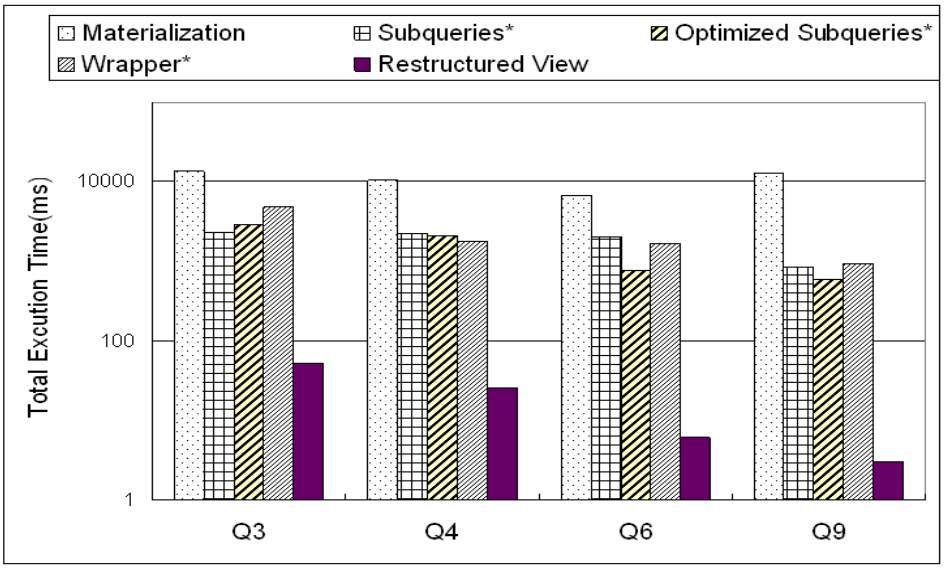

Figure 7.2: Performance comparison (on the DB-Research dataset) for optimization using restructured views.

Our experimental results for Scenario V are shown in Figure 7.2, and the results for Scenario VI are shown in Figure 7.3. Note that the Y-axis, the execution time (measured in milliseconds), is logarithmic.

As shown in Figure 7.2, optimization using materialized restructured views is applicable and yields better performance. Figure 7.3 shows that the improvement over the other techniques amounts to one to three orders of magnitude. Note also that the semantic optimization, the *optimized subqueries** approach, which uses key-constraint information to optimize the XQuery subqueries, yields an order of magnitude or better performance than that of the (native) optimized subqueries approach. We notice that user queries in the Stocks dataset are much more "complicated" than queries in the DB-Research dataset according to the amount of binary relations in user queries. Depending on the allocated memory, large numbers of variables in an XQuery query may have a pronounced effect on execution time by SAXON.

We expect the gains obtained by the optimized algorithm to be typical for all user queries with moderate to large number of joins. In addition, our optimization using restructured views can be applied whenever the data exhibits certain regularity properties, thus resulting in substantial size reduction via restructured views.

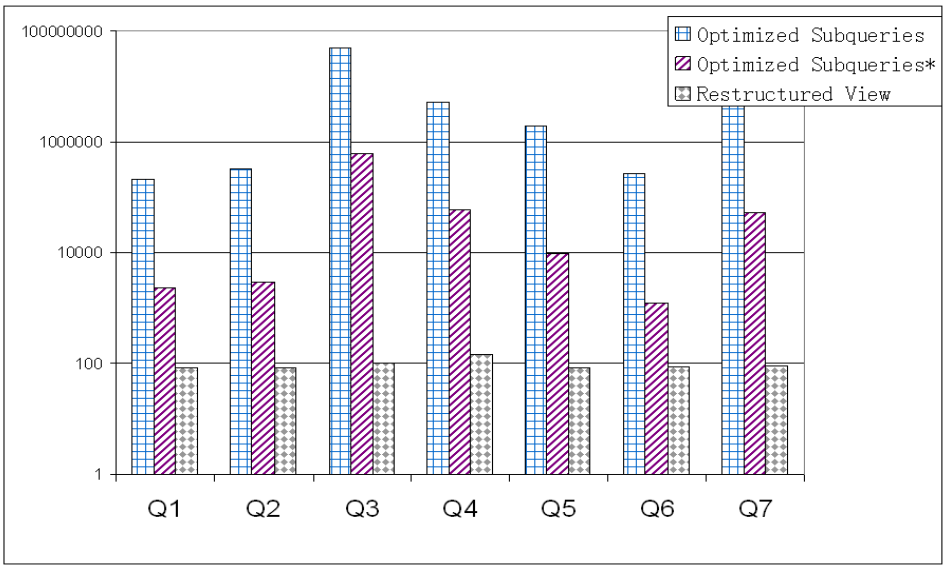

Figure 7.3: Performance comparison (on the Stocks dataset) for optimization using restructured views.

Chapter 8

Conclusions

We have presented a set of algorithms for query processing and optimization in the semantic-model approach to large-scale information integration and interoperability. In addition to supporting gradual large-scale information integration and efficient inter-source processing, the SM approach eliminates the need for mediation in deriving the global schema, thus addressing the main limitation of data-integration systems.

To the best of our knowledge, our methods are the first to account for the practical issues of information overlap across data sources and of inter-source processing. While most of algorithms presented in this dissertation are platform- and implementation-independent, we also proposed XML-specific optimization techniques that allow for system-level tuning of query-processing performance.

We discussed a general query-optimization framework that treats regular and restructured views in a uniform manner and is applicable to select-project-join queries and views without or with aggregation. Within the framework we provided (1) algorithms to determine when a view (regular or restructured) is usable in answering a query, and (2) algorithms to rewrite queries using usable views. Semantic information, such as knowledge of the key of a view, can be used to further optimize a rewritten query. Within our general query-optimization framework, we developed techniques for determining the key of a (regular or restructured) view, and showed how this information can be used to further optimize a rewritten query.

We then incorporated the concept of materialized *restructured views* into query processing and optimization in the semantic-model approach. Using restructured views in query optimization opens up a new spectrum of views that were not previously available in

information-integration projects, and can result in significant additional savings in query-evaluation costs. We developed algorithms for materializing restructured views for XML information sources, and for rewriting user queries on the semantic-model view to equivalent queries on restructured views. It has been shown that classical query optimization using restructured views can improve query-processing efficiency significantly for certain classes of queries, sometimes by an order of magnitude or more. Our experimental results demonstrated that query optimization using restructured views can and does result in similarly significant performance gains.

Currently we are continuing our study on the performance and improvements of algorithms for query processing and optimization in the semantic-model approach to large-scale information integration and interoperability. Because of the complexity of information integration problem itself, no single query-processing strategy would be optimum for all queries and cases. The *restructured view* approach also gives us a problem of (materialized) restructured view maintenance. Restructured views shall be updated correctly and efficiently whenever data in sources are changed. This is also an interesting topic for future research directions. Besides, as for our implementation of an information-integration system shell, we are incorporating schema mapping, data exchange and query translation in Peer Data Management Systems into "inter-coordinator processing" modules.

Bibliography

[1] Academic Department Ontology. `http://www.daml.org/ontologies/65`.

[2] F. Afrati and R. Chirkova. Selecting and using views to compute aggregate queries. In *Proc. ICDT*, 2005.

[3] Marcelo Arenas, Vasiliki Kantere, Anastasios Kementsietsidis, Iluju Kiringa, Renée J. Miller, and John Mylopoulos. The Hyperion project: From data integration to data coordination. *SIGMOD Record*, 32(3):53–58, 2003. Special issue on Peer to Peer Data Management.

[4] Marcelo Arenas and Leonid Libkin. XML data exchange: Consistency and query answering. In *Proceedings of ACM Symposium on Principles of Database Systems*, pages 13–24, 2005.

[5] Tim Berners-Lee, James Hendler, and Ora Lassila. The Semantic Web. *Scientific American*, May 2001.

[6] S. Chaudhuri and U. Dayal. An overview of data warehousing and OLAP technology. *SIGMOD Record*, 26(1):65–74, 1997.

[7] S. Chaudhuri and M. Y. Vardi. Optimization of real conjunctive queries. In *Proc. PODS*, pages 59–70, 1993.

[8] Surajit Chaudhuri, Ravi Krishnamurthy, Spyros Potamianos, and Kyuseok Shim. Optimizing queries with materialized views. In *Proceedings of IEEE International Conference on Data Engineering*, pages 190–200, 1995.

[9] Dongfeng Chen, Rada Chirkova, and Fereidoon Sadri. Designing an Information In-

tegration and Interoperability System — First Steps. Technical Report NCSU CSC TR-2006-30. Available at `http://www4.ncsu.edu/~rychirko/Papers`, October 2006.

[10] R. Chirkova and F. Sadri. Query optimization using restructured views. In *CIKM*, pages 642–651, 2006.

[11] Vassilis Christophides, Gregory Karvounarakis, I. Koffina, G. Kokkinidis, Aimilia Magkanaraki, Dimitris Plexousakis, G. Serfiotis, and Val Tannen. The ICS-FORTH SWIM: A powerful Semantic Web integration middleware. In *Proceedings of VLDB Workshop on Semantic Web and Databases*, pages 381–393, 2003.

[12] Vassilis Christophides, Gregory Karvounarakis, Aimilia Magkanaraki, Dimitris Plexousakis, and Val Tannen. The ICS-FORTH Semantic Web integration middleware (SWIM). In *IEEE Data Engineering Bulletin*, pages 11–18, 2003.

[13] CiteSeer. `http://citeseer.ist.psu.edu/`.

[14] The Clio project. `http://www.cs.toronto.edu/db/clio`.

[15] Sara Cohen, Werner Nutt, and Alexander Serebrenik. Rewriting aggregate queries using views. In *Proceedings of ACM Symposium on Principles of Database Systems*, pages 155–166, 1999.

[16] Conor Cunningham, Goetz Graefe, and César A. Galindo-Legaria. PIVOT and UNPIVOT: Optimization and execution strategies in an RDBMS. In *Proceedings of International Conference on Very Large Databases*, pages 998–1009, 2004.

[17] DAML Ontology Library. `http://www.daml.org/ontologies/`.

[18] Susan Davidson, Wenfei Fan, Carmem Hara, and Jing Qin. Propagating XML constraints to relations. In *Proceedings of IEEE International Conference on Data Engineering*, 2003.

[19] DBLP. `http://www.informatik.uni-trier.de/~ley/db/index.html`.

[20] Robin Dhamankar, Yoonkyong Lee, AnHai Doan, Alon Y. Halevy, and Pedro Domingos. iMAP: Discovering complex semantic matches between database schemas. In *Proceedings of ACM SIGMOD International Conference on Management of Data*, pages 383–394, 2004.

[21] Data Integration in Life Sciences, 2007. `http://dils07.cis.upenn.edu`.

[22] Hong Hai Do and Erhard Rahm:. COMA - a system for flexible combination of schema matching approaches. In *Proceedings of International Conference on Very Large Databases*, pages 610–621, 2002.

[23] AnHai Doan, Pedro Domingos, and Alon Halevy. Reconciling schemas of disparate data sources: A machine-learning approach. In *Proceedings of ACM SIGMOD International Conference on Management of Data*, 2001.

[24] AnHai Doan, Pedro Domingos, and Alon Halevy. Reconciling schemas of disparate data sources: A multistrategy approach. *Machine Learning*, 50(3):279–301, 2003.

[25] AnHai Doan and Alon Halevy. Efficiently ordering query plans for data integration. In *Proceedings of IEEE International Conference on Data Engineering*, pages 393–402, 2002.

[26] AnHai Doan, Jayant Madhavan, Pedro Domingos, and Alon Halevy. Learning to map between ontologies on the semantic web. In *Proceedings of the International WWW Conference*, pages 662–673, 2002.

[27] Xin Dong and Alon Halevy. Indexing dataspaces. In *Proceedings of ACM SIGMOD International Conference on Management of Data*, pages 43–54, 2007.

[28] Xin Dong, Alon Y. Halevy, and Igor Tatarinov. Containment of nested XML queries. In *Proceedings of International Conference on Very Large Databases*, pages 132–143, 2005.

[29] Mark Gyssens, Laks V. S. Lakshmanan, and Iyer N. Subramanian. Tables as a paradigm for querying and restructuring. In *Proceedings of ACM Symposium on Principles of Database Systems*, pages 93-103, 1996. .

[30] Alon Y. Halevy. Answering queries using views. *The VLDB Journal*, 10(4):270–294, 2001.

[31] Alon Y. Levy, Anand Rajaraman, and Joann J. Ordille. Querying heterogeneous information sources using source descriptions. In *VLDB*, pages 251–262, 1996.

[32] Alon Halevy, Oren Etzioni, AnHai Doan, Zachary Ives, Jayant Madhavan, Luke McDowell, and Igor Tatarinov. Crossing the structure chasm. In *Proceedings of the Biennial Conference on Innovative Data Systems Research (CIDR)*, 2003.

[33] Alon Halevy, Michael Franklin, and David Maier. Principles of dataspace systems. In *Proceedings of ACM Symposium on Principles of Database Systems*, pages 1–9, 2006.

[34] Alon Y. Halevy, Zachary G. Ives, Jayant Madhavan, Peter Mork, Dan Suciu, and Igor Tatarinov. The Piazza peer data management system. *IEEE Transactions on Knowledge and Data Engineering*, 16(7):787–798, 2004.

[35] Alon Y. Halevy, Zachary G. Ives, Peter Mork, and Igor Tatarinov. Piazza: data management infrastructure for semantic web applications. In *Proceedings of the International WWW Conference*, pages 556–567, 2003.

[36] Alon Y. Halevy, Zachary G. Ives, Dan Suciu, and Igor Tatarinov. Schema mediation in peer data management systems. In *Proceedings of IEEE International Conference on Data Engineering*, pages 505–516, 2003.

[37] The Hyperion project. `http://www.cs.toronto.edu/db/hyperion`.

[38] EDBT 2006 Workshop on Information Integration in Healthcare Applications (IIHA), 2006. `www6.informatik.uni-erlangen.de/events/wsIIHAedbt2006/`.

[39] Workshop on Information Integration Methods, Architectures, and Systems (IIMAS), in conjunction with ICDE'08, 2008. `http://daks.ucdavis.edu/IIMAS08/`.

[40] International Organization for Standardization (ISO), Technology C database languages C SQL (December 2003).

[41] Zachary G. Ives, Alon Y. Halevy, and Daniel S. Weld. Adapting to source properties in processing data integration queries. In *Proceedings of ACM SIGMOD International Conference on Management of Data*, pages 395–406, 2004.

[42] Gregory Karvounarakis, Sofia Alexaki, Vassilis Christophides, Dimitris Plexousakis, and Michel Scholl. RQL: A declarative query language for RDF. In *Proceedings of the International WWW Conference*, pages 592–603, 2002.

[43] Anastasios Kementsietsidis and Marcelo Arenas. Data sharing through query translation in autonomous sources. In *Proceedings of International Conference on Very Large Databases*, pages 468–479, 2004.

[44] Anastasios Kementsietsidis, Marcelo Arenas, and Renée J. Miller. Mapping data in peer-to-peer systems: Semantics and algorithmic issues. In *Proceedings of ACM SIGMOD International Conference on Management of Data*, pages 325–336, 2003.

[45] Maurizio Lenzerini. Data integration: A theoretical perspective. In *PODS*, pages 233–246, 2002.

[46] Laks V. S. Lakshmanan and Fereidoon Sadri. Interoperability on XML data. In *Proceedings of the International Semantic Web Conference (ISWC)*, pages 146–163, 2003.

[47] Laks V. S. Lakshmanan, Fereidoon Sadri, and Iyer N. Subramanian. Logic and algebraic languages for interoperability in multidatabase systems. *Journal of Logic Programming*, 33(2):101–149, November 1997.

[48] Laks V. S. Lakshmanan and Fereidoon Sadri and Subbu N. Subramanian. On Efficiently Implementing SchemaSQL on an SQL Database System. In *Proceedings of International Conference on Very Large Databases*, pages 471-482, 1999

[49] Jayant Madhavan, Philip A. Bernstein, AnHai Doan, and Alon Y. Halevy. Corpus-based schema matching. In *Proceedings of IEEE International Conference on Data Engineering*, 2005.

[50] Jayant Madhavan, Shirley Cohen, Xin Luna Dong, Alon Y. Halevy, Shawn R. Jeffery, David Ko, and Cong Yu. Web-scale data integration: You can afford to pay as you go. In *Proceedings of the Biennial Conference on Innovative Data Systems Research (CIDR)*, pages 342–350, 2007.

[51] Jayant Madhavan and Alon Y. Halevy. Composing mappings among data sources. In *Proceedings of International Conference on Very Large Databases*, pages 572–583, 2003.

[52] Douglas Markland. An XML mapping tool. Project Report, Department of Mathematical Sciences, University of North Carolina at Greensboro, April 2004.

[53] Renée J. Miller, Laura M. Haas, and Mauricio A. Hernández. Schema mapping as query discovery. In *Proceedings of International Conference on Very Large Databases*, pages 77–88, 2000.

[54] Renée J. Miller, Mauricio A. Hernández, Laura M. Haas, Ling-Ling Yan, C. T. Howard Ho, Ronald Fagin, and Lucian Popa. The Clio project: Managing heterogeneity. *SIGMOD Record*, 30(1), 2001.

[55] Natalya F. Noy and Deborah L. McGuinness. Ontology development 101: A guide to creating your first ontology, 2001. `http://ksl.stanford.edu/people/dlm/papers/ontology-tutorial-noy-mcguinness.pdf`.

[56] Werner Nutt, Yehoshua Sagiv, and Sara Shurin. Deciding equivalences among aggregate queries. In *Proceedings of ACM Symposium on Principles of Database Systems*, pages 214–223, 1998.

[57] The Piazza project. `http://data.cs.washington.edu/p2p/piazza`.

[58] Lucian Popa, Yannis Velegrakis, Renée J. Miller, Mauricio A. Hernández, and Ronald Fagin. Translating web data. In *Proceedings of International Conference on Very Large Databases*, 2002.

[59] PostgreSQL. `http://www.postgresql.org/`.

[60] Erhard Rahm and Philip A. Bernstein. A survey of approaches to automatic schema matching. *The VLDB Journal*, 10(4):334–350, 2001.

[61] SAXONICA XSLT and XQuery Processing. `http://www.saxonica.com/`.

[62] Semantic Web. `http://www.w3c.org/2001/sw/`.

[63] SIGMOD. `http://www.sigmod.org/`.

[64] Divesh Srivastava, Shaul Dar, H. V. Jagadish, and Alon Y. Levy. Answering queries with aggregation using views. In *Proc. VLDB*, pages 318–329, 1996.

[65] Igor Tatarinov and Alon Halevy. Efficient query reformulation in peer data management systems. In *Proceedings of ACM SIGMOD International Conference on Management of Data*, pages 539–550, 2004.

[66] Igor Tatarinov, Zachary G. Ives, Jayant Madhavan, Alon Y. Halevy, Dan Suciu, Nilesh N. Dalvi, Xin Dong, Yana Kadiyska, Gerome Miklau, and Peter Mork. The piazza peer data management project. *SIGMOD Record*, 32(3):47–52, 2003.

[67] The TSIMMIS project home page, in *http://www-db.stanford.edu/tsimmis/tsimmis.html*.

[68] Jeffrey D. Ullman. *Principles of Database and Knowledge-Base Systems, Volume I.* Computer Science Press, 1988.

[69] Jeffrey D. Ullman. Information integration using logical views. In *Proceedings of International Conference on Database Theory*, pages 19–40, 1997.

[70] NCSU-UNCG Information-Integration Project `http://dbgroup.ncsu.edu/?page_id=205`.

[71] XML Path Language (XPath). `http://www.w3c.org/TR/xpath`.

[72] Ling-Ling Yan, Renée J. Miller, and Laura M. Haas. Data-driven understanding and refinement of schema mappings. In *Proceedings of International Conference on Very Large Databases*, 2001.