Supporting Lineage Tracing in Mediator-Based Information Integration Systems

Ali Taghizadeh-Azari

A Thesis in the
Department of Computer Science and Software Engineering

Presented in Partial Fulfillment of the Requirements
For the Degree of Master of Computer Science

Concordia University
Montreal, Quebec, Canada

March 2005

Your file Votre référence
ISBN: 0-494-04452-7
Our file Notre référence
ISBN: 0-494-04452-7

Abstract

Supporting Lineage Tracing in Mediator-Based Information Integration Systems

Ali Taghizadeh-Azari

Information integration provides users with a uniform interface to multiple (possibly heterogeneous) data sources. Two main approaches to information integration are data warehousing and mediator-based. The problem of providing explanation for a query answer is referred to as lineage tracing. This problem has been studied extensively in the context of data warehouse systems, however, for mediator-based systems, this is identified as a research problem [HC'03]. In such a system, the mediator does not store data. This means for query processing as well as for tracing, the mediator has to "communicate" with the data sources. While this communication could be expensive, the real issue is that after a query is being processed, lineage tracing could be more difficult or even impossible, if the structure of some contributing sources changes, or if the content of such sources change or a source become unavailable. This means, to support lineage tracing, we need to collect "enough" data and metadata information during query processing. In this work, we study this problem, and introduce data structures and algorithms to support lineage tracing in two modes: batch and interactive. We have successfully developed a prototype, called *ELIT*, for *Exploration and LIneage Tracing*. We also study query optimization in the content of ELIT and implemented some basic optimization techniques. While more sophisticated techniques are required in this context, we believe the ideas proposed in this work lend themselves to useful analysis and tracing tools in mediator-based systems.

Acknowledgment

I am deeply grateful to my supervisor, Professor Nematollaah Shiri, for his encouragement, support, valuable advice, generous attention, constant help, and proofreading many drafts of the thesis throughout my studies at Concordia University. He patiently guided me with his profound knowledge and encouraged me in tackling various difficult issues. I am greatly honored to study under his supervision.

This research was supported in part by Natural Sciences and Engineering Research Council (NSERC) of Canada and Concordia University, ENCS Faculty Research Support.

My thanks also go to Halina Monkiewicz and Veronica Jacobo-Gutierrez for their support.

I am extremely grateful to my friend Ali Kiani, who helped me a lot in different stages of the development of this work.

My special thanks are also due to the faculties and staffs in the Computer Science Department at Concordia University, who provided me a good opportunity to learn and progress.

Finally, I would like to thank my parents and my wife for their support and encouragement in pursing my studies.

Table of Content

1. Introduction 1

1.1 Motivating Example 3

1.2 Contributions of the Thesis 7

1.3 Thesis Outline 9

2. Background and Related Work 11

2.1 Data Warehouse 12

2.2 Mediator Based Systems 16

2.3 Related Work 18

2.3.1 Lineage Tracing 19

3. Metadata Collection 25

3.1 Collecting Schema Information 25

3.2 Collecting Schema Information Algorithm 30

4. Query Formulation and Processing 32

4.1 Visual Query Building 33

4.2 Query Type 34

4.3 Query Parsing 36

4.4 Query Components 37

4.4.1 Collecting Query Components 41

4.5 Query Evaluation 42

4.6 Lineage Tracing 43

4.6.1 Lineage Tracing Algorithm 44

5. ELIT: A System Prototype 46

5.1 System Design 46

5.2 System Illustration 48

5.3 Architecture of ELIT 53

5.4 Performance Evaluation 55

6. Lineage Tracing in Distributed Database Systems 60

6.1 Distributed Database Systems 60

6.2 Lineage Tracing and Distributed Database Systems 65

7. Conclusions and Future Work 68

7.1 Conclusions 68

7.2 Future Work 69

References 70

Appendices 73

Appendix A: System Requirements 74

Appendix B: User Interface Specifications 75

Appendix C: List of Tables 76

Appendix E: List of Views 81

Appendix F: List of User Interfaces in ELIT 84

F-1: Table Definition 84

F-2: XML Source Selection 85

F-3: Query Component Creation 87

F-4: Lineage Tracing Process 90

List of Figures

Figure 1-1: SALES table 4

Figure 1-2: Fragments of the XML schema XSD for ITEM 5

Figure 2-1: Star schema 14

Figure 2-2: Data warehouse system 15

Figure 2-3: Mediator-based system 18

Figure 2-4: Lineage tracing result for SALES and ITEM data sources 20

Figure 4-1: Steps in lineage tracing – *IMDS* creation and query parsing 37

Figure 4-2: Transformation functions execution plan 38

Figure 4-3: Preparation steps for lineage tracing 43

Figure 5-1: Storing query information in internal metadata tables 47

Figure 5-2: User query GUI 48

Figure 5-3: *DRT* after inserting metadata and atomic data values 49

Figure 5-4: Query result 50

Figure 5-5: *DRT* after applying One-Side-Condition 51

Figure 5-6: *DRT* after applying Two-Side-Condition 52

Figure 5-7: ELIT processing model 54

Figure 5-8: Lineage tracing evaluation result 57

Figure 5-9: Lineage tracing evaluation result – non-optimized 58

Figure 5-10: Lineage tracing evaluation result – partial *DRT*, optimized 58

Figure 5-11: Lineage tracing evaluation result – optimized 59

Figure 6-1: Distributed database system architecture 62

Figure A-1: Entity relationship diagram 80

Figure A-2: Table definition GUI 85

Figure A-3: XML source selection GUI 86

Figure A-4: XML structure view GUI 86

Figure A-5: View definition GUI 87

Figure A-6: Table selection GUI 88

Figure A-7: Column selection GUI 88

Figure A-8: Where condition GUI 89

Figure A-9: Where condition, column selection GUI 89

Figure A-10: Final select statement GUI 90

Figure A-11: Lineage tracing GUI 91

List of Tables

Table 1: List of forms and related data blocks and their tables 75

Table 2: List of internal tables 78

Table 3: List of internal views and their definition.. 83

Chapter 1

Introduction

Information integration systems provide users with a uniform interface to a multitude of (possibly heterogeneous) data sources [GMUW'01]. As in database systems, a user in an information integration system poses his/her query to the system and system responds by returning the result back to the user. In order to formulate the query, all the user needs to know is the structures of the data, i.e., the global schema or a view.

It is the responsibility of the underlying system to check the query for its syntax and the semantics. If all is in order, then the system dispatches the query to the appropriate sources to be processed. The results from sources are then combined and returned to the user as the answer to the query. Data warehouse and mediator-based systems are among the main approaches for information integration. We will review these two approaches and study their support for *lineage tracing*, chapter 2.

In many applications, it is desired to know which atomic data contributed to the query result, for instance for data analysis which traces data back to the original data. This is known as *the lineage tracing problem* [CW'01]. A question, which may arise at this point, is: "what do we need to do in order to support lineage tracing?" To address this, we need the following capabilities:

a) To have *all* the information about the schema and locations of the data sources involved. This is required in particular to allow the user to interact with the system to select the desired data items to be traced and displayed.

b) To have the query itself to define the transformation functions. These functions are introduced to the system, according to query specifications. For each condition in the query, a transformation function is used to transfer an input data set to the same or different output data set.

c) To deal with multiple data sources because of the nature of the mediator-based system environment.

Lineage tracing has been studied extensively in the context of data warehouse systems [CW'01, FP'02]. A data warehouse is a centralized database system, defined from data sources through operations of extraction, transformation, load, and refresh. In a mediator-based information integration system, on the other hand, we may submit a query to the system through the mediator and receive the result; no data is explicitly stored in the mediator and hence the actual data sources will be involved in answering the query.

In such a system, accessing atomic data contributing to the query answer is time consuming and sometimes impossible. For example, if it takes T_1 to retrieve the data from data sources and combine them, and it takes T_2 for the user to request for lineage tracing and for the system to apply the linage tracing functions, then, the total time $T = T_1 + T_2$ must be considered as a gap between the first attempt to access the data from sources and the second attempt to provide linage tracing. Considering this gap, a problem, which may arise in case of some data update in the contributing source(s), after

T_1 and before T_2, lineage-tracing will return the atomic data which have already been changed in the original data sources. Data caching would be a solution for this problem, but could be too expensive if the sources involved are huge.

In both data warehouse and mediator-based approaches, we need to identify the atomic data contributed to the answer. However, due to the dynamic nature of query answering in a mediator-based approach, there are some new issues here, which we may face in query processing and lineage tracing as identified as a challenge research [HC'03]. In this case, we need to study the problem and the various issues involved and develop new techniques for tracing the data origins.

In this thesis, we attempt to study lineage tracing in mediator-based information integration systems and develop a framework to address these issues. Our work takes a *fine-grained* approach, which means we consider lineage tracing at an instance level. Other approach is *coarse-grained*, which produces the lineage tracing answer in a schema level. Having instance level requires having the schema level itself. This is why in a mediator-based system some kind of schema mapping to the original data sources should be applied to obtain the schema information.

1.1 Motivating Example

We consider a simple sales application example to illustrate the problem and our solution approach. In this example, we have a heterogeneous mediator-based information integration system consisting of two different types of data sources.

The first data source is a relational database, which contains SALES table with the attributes StoreID, ItemID, NumSold, and Price as shown in Figure 1-1. The second data source is an XML data file, ITEM, with the attributes ItemID, ItemName, and Category as shown in Figure 1-2 (XSD) and 1.3 (data file).

StoreID	***ItemID***	***NumSold***	***Price***
2	1	800	5
2	2	2000	2
2	4	800	35
3	3	1500	45
3	4	600	60
4	3	2100	50
4	4	1200	70
4	5	200	30
1	1	1000	4
1	2	3000	1

Figure 1-1: SALES table

We assume the XML data file does not have any deep nesting structure, in order to simplify our XML parser. Arbitrary XML and XSD files could be supported at an added cost of a more complex parser. This is not pursued in this work.

We will illustrate the problem of lineage tracing in this example of lineage tracing environment and show how explanation could be provided through a collection of lineage tracing functions.

```
.......
<xs:complexType name = "ITEM">
<xs:attribute name="ItemID" type="ID" use= "required"/>
<xs:attribute name="ItemName" type="string"
use="required"/>
<xs:attribute name="Category" type="string"
use="required"/>
</xs:complexType>
........
```

Figure 1-2: Fragments of the XML schema XSD for ITEM

```
.......
<ITEM ItemID="1" ItemName="binder"
Category="stationary"</Item>
<ITEM ItemID ="2" ItemName="pencil"
Category="stationary"</Item>
<ITEM ItemID ="3" ItemName="shirt"
Category="clothing"</Item>
<ITEM ItemID ="4" ItemName="pants"
Category="clothing"</Item>
<ITEM ItemID ="5" ItemName="pot"
Category="kitchenware"</Item>
.......
```

Figure 1-3: Fragments of the XML data files for `ITEM`

We store in a table pool, all the metadata information as well as the data used in answering a query. The *data pool* is essentially a table to which we refer as *Data Reference Table* (*DRT,* for short). Figure 1-4 exhibits an instance of *DRT*, assuming we have a query on SALES and ITEMS.

TableName	***RecordNumber***	***ColumnName***	***Value***
SALES	1	StoreID	2
SALES	1	ItemID	1
SALES	1	NumSold	800
SALES	1	Price	5
SALES	2	StoreID	2
SALES	2	ItemID	21
SALES	2	NumSold	2000
SALES	2	Price	2
ITEM	1	ItemID	1
ITEM	1	ItemName	Binder
ITEM	1	Category	Stationary
ITEM	2	ItemID	21
ITEM	2	ItemName	Pencil
ITEM	2	Category	Stationary

Figure 1-4: Data Reference Table (*DRT*)

As shown in the figure, there is a tuple for each individual data value in the data sources involved in processing the query. For example, consider the first tuple from SALES table in Figure 1-1.

For each component in the table, there is a unique tuple in the *DRT*. Here we have `StoreID` as a column name having value 2 at row 1. We get a unique tuple in *DRT* for each of the four components 2, 1, 800, and 5 in tuple 1 in `SALES` table. This applies to all the tables, `SALES` and `ITEM` involved in answering this query.

As it can be realized, *DRT* is a large table needed to support lineage tracing in our solution by capturing the data items involved in answering a query. Managing *DRT* efficiently to support lineage tracing functions, two modes of batch and interactive, is a main goal in this work.

We have developed a system prototype that implements the proposed algorithms, as a tool for tracing in mediator-based information integration system. Our ideas together with the technical contribution in this research are published in ISSADS 2005 [ST'05].

1.2 Contributions of the Thesis

Before we highlight our contributions in this work, let us recall our motivation and goals in lineage tracing in mediator-based approach of information integration. Lineage tracing has been studied extensively in the context of data warehouse systems [CW'01]. In such a system, data is stored as a central database, after being selected, extracted, cleaned, and transformed from various existing information sources. Because of the nature of a data warehouse being a centralized database, answering queries is similar to answering queries in databases, as all the data and metadata information is available to the query processor component of the database management system.

In a mediator-based system, on the other hand, there is no data stored in the mediator, and hence the data has to be retrieved from the sources during processing of the query. This implies more work and new challenges in query processing in mediator-based systems than data warehouse systems.

There is another requirement here. In addition, the same data that contributed to a query answer must be used for lineage tracing. This requirement is more challenging to satisfy in a mediator-based framework since a data source may no longer be available after query result is returned and a user requests the system for an explanation, or in case of an update of the data in the source or changes in the schema, lineage tracing may not even be possible unless the data or schema used to answer the query is somehow *captured/cached* during processing of the query. This of course should be done with care; otherwise lineage tracing may not be possible.

For instance, a solution might require storing atomic data at the mediator, which is against the philosophy of the mediator approach. Query processing in a mediator-based approach in general requires dealing with several data sources, and hence in order to support lineage tracing, we may need to access data from different sources. All of above could be challenging subjects, which motivated us in the development of the concepts and algorithms in this research, listed as follows:

a) We study lineage tracing problem in the context of mediator-based information integration systems and identify issues to be resolved.

b) We propose a solution technique by extending the corresponding solution in the data warehouse approach, which takes into account the distributed nature of the environment and processes in a mediator-based approach.

As a solution idea, we propose to capture the data used in answering a query in a data pool, called DRT, and to collect metadata information in IMDS. We also propose algorithms to manipulate these data structures to support lineage tracing in batch mode as well as interactive mode.

c) We have successfully developed a prototype, ELIT, to serve as viability of the ideas and techniques proposed in this research.

d) We studies query optimization in the context of ELIT and reported experimental results.

e) We also studied lineage tracing in distributed database systems. Our preliminary result show there are new challenges in this direction, as many query processing and optimization components in a distributed database management system will be affected for supporting lineage tracing.

1.3 Thesis Outline

The rest of this report is organized as follows. In chapter 2 we provide a background and review of related works on lineage tracing. This includes a description of information integration systems and its approaches: data warehouse and mediator-based.

In chapter 3, we illustrate the first step of our solution approach, called *Collecting Schema Information*, to support lineage tracing.

Chapter 4 includes our main contribution in this research. We first introduce the second step, called *Query Formulation and Processing*. We will also introduce the functions and processes required to extract query components to support lineage tracing.

In chapter 5, we introduce our system prototype, ***ELIT*** (**E**xplanation and **LI**neage **T**racing in information integration), and illustrate its features and capabilities through an example application. The tracing algorithms are also illustrated in this chapter.

In chapter 6 we study the lineage tracing in distributed database systems. Unlike what we expect, we will argue that this problem is more difficult to deal with as it requires changes in almost every aspect of query processing and query optimization.

The last part, chapter 7, includes concluding remarks and possible future directions.

Chapter 2

Background and Related Work

Data and data sources are growing rapidly. While they could share a common subject they model, sources could be heterogeneous in many ways, including their types, values, semantics, presentations, etc. In many real life applications, users might need to access several data sources to get answers to their queries. Information integration systems provide users with an interface to view and access different data sources as a unified, single data source. The data sources in an information integration system, in general, are heterogeneous, which are designed, developed, and maintained independently. The system allows users to interact with different data sources without requiring them to know the internal structures and the physical design of the sources.

Large companies and enterprises usually have several data sources that support their day-to-day activities and business processes, in addition to the web technology. Answering queries over such data sources is both desired and complex. Research is underway worldwide to address theoretical and practical issues in this regard.

A number of approaches have been proposed to provide this integrated model including federated databases, data warehouse, mediator-based, and peer-to-peer systems [GMUW'01].

In a federated database, there is a group of databases "agreed" to cooperate in answering queries posed from any user of the federated system. Each database in this system has a "connection" to other databases. There is no global schema accessible in a federated system. In other words, in case these are n databases available in the system, there would be $n \times (n-1)$ interfaces, essentially codes, to support queries between different sources. It is straightforward to build such a system especially when the number of databases is not large.

In addition to scalability concern, another problem here is the amount of work involved in case a new database is added or removed from the system, for the interface to continue to be applicable and to support the querying over the available sources [GMUW'01].

As two main approaches to information integration, in what follows we review the basis of data warehouse and mediator-based systems study the problem of lineage tracing problem in each of these two contents.

2.1 Data Warehouse

A data warehouse is a collection of data stored in a centralized database system containing (possibly heterogeneous) data sources. This collection is integrated and subject oriented, according to William Inmon, considered as the father of modern data warehouse [I'96]. Data from data sources are selected, extracted, transformed, and combined into a global database schema accessible by the users. This global schema looks like an ordinary database. Queries over the warehouse are exactly the same as queries in single conventional databases; however, user is not allowed to update the data warehouse, since the original purpose here has been the ability for query analysis and processing over multiple sources.

To maintain a data warehouse, there are a number of approaches, described as follows:

a) *Periodic reconstruction*: this is the most common way to update the warehouse. Data warehouse is down during the update period and no query is allowed to submit. Availability, long time updating process, and out of date data are disadvantages of this approach.
b) *Incremental Update*: this is the same as the previous approach, but instead of having a fixed period of time to update, the required update time depends on the changes, which have been made to the underlying sources. Short updating time, which is important for large data warehouse is the advantage of this approach compared to the first approach. The disadvantage here is that the process of calculating changes is complex.
c) *Immediate update*: data warehouse is updated immediately after any changes occurred in the underlying data sources. This requires programming skills to implement practical communication and the required processing [GMUW'01].

Data warehouse systems provide system managers and decision makers a uniform access to information quickly to answer queries. User in a data warehouse system has the query result from the integrated, single data source instead of accessing data from multiple heterogeneous data sources. Having such a unified data storage helps to have query result in an easy and more efficient way at the cost of building and maintaining the data warehouse.

For the design of a data warehouse, there are alternative schemas, such as *star* schema and *snowflakes*. In a star schema, there is a fact table, e.g., SALES at the center and a number of dimensions related to the fact, e.g., ITEMS. Figure 2-1 shows a star schema. The snowflake schema is the same as the star schema but with the dimensions normalized in a tree format.

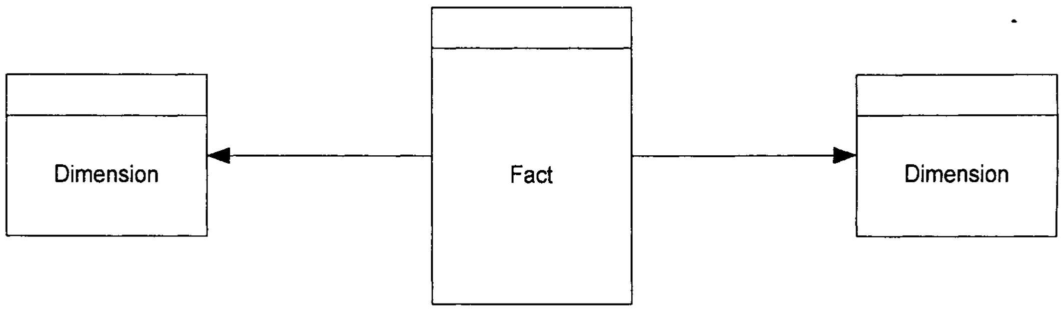


Figure 2-1: Star schema

Figure 2-2 illustrates a typical data warehouse system. As shown in the figure, a user can access the data in the warehouse, which is explicitly stored in the relations and many views. Typically, views in the data warehouse are summarized data, often required to retrieve from the sources. For this, there are three procedures, described as follows:

a) *Extract*: this is a predefined procedure that extracts the data from data sources. As this is a predefined, all necessary data has to be identified when designing the system.

b) *Transform*: This transforms the data from source to the required format in the warehouse

c) *Load*: this loads the data from sources to the appropriate tables in the data warehouse.

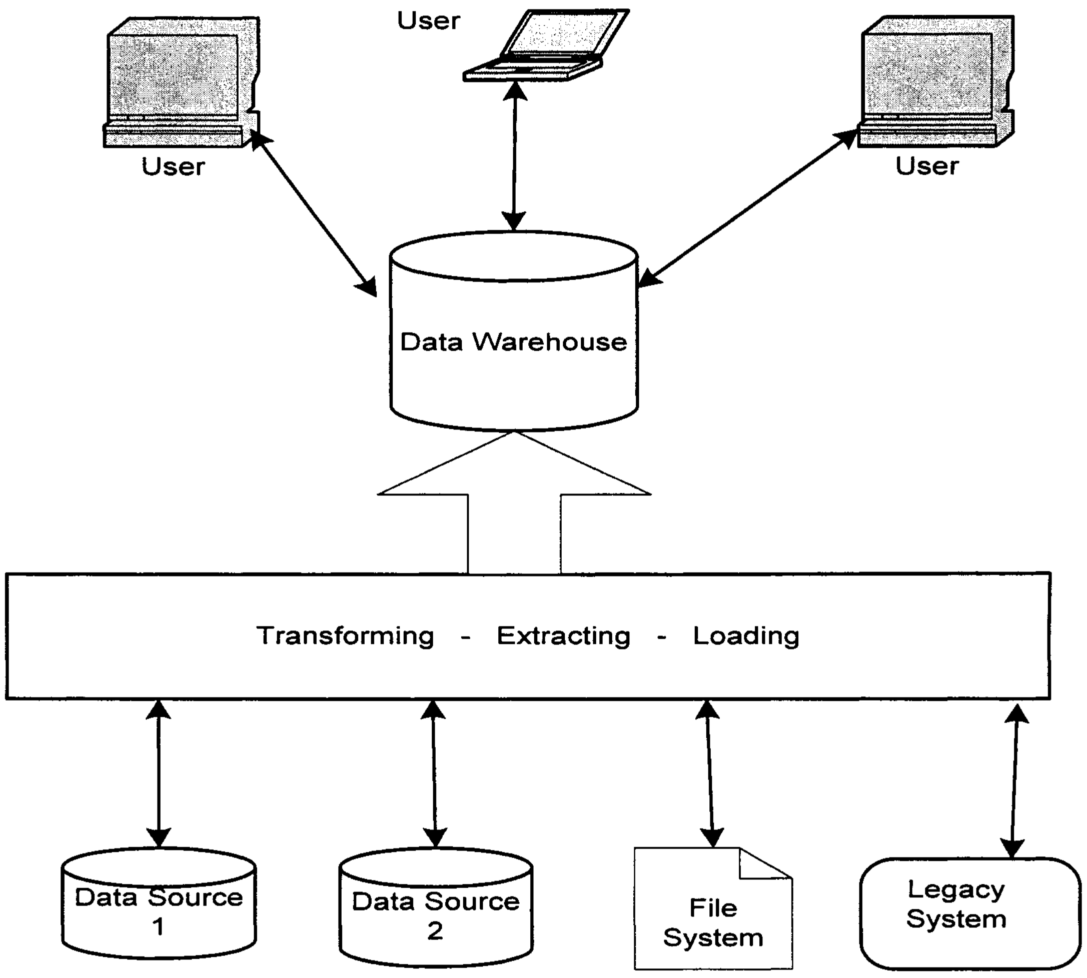


Figure 2-2: Data warehouse system

In a data warehouse, data is updated on some regular basis, which is application dependent. Bringing the data in the warehouse and make it up to date, has been a main issue in data warehouse maintenance, and the solution proposed as similar to view maintenance problem, in databases studied extensively [GMUW'01]. Next, we review mediator-based approach for information integration and study its costs and benefits.

2.2 Mediator Based Systems

A mediator-based system is similar to a warehouse system in the way data is collected data comes from (heterogeneous) data sources, but data is not stored in unified data storage. This can be implemented using a virtual view or collection of views, which interact with the original data sources [GMUW'01].

The information may be different for their types, subjects, models and so on. The system designer provides a global schema based on which user can pose a query. It is the mediator's responsibility to dispatch the query to relevant data sources, process each sub query, combine answers from the sources, and then return the query result back to the user [HIST'03].

Query processing here is different from processing the queries in relational databases. Mediator has access to data sources through wrappers, each of which provide the mediator a view over the available data in the corresponding source. Query processing includes query rewriting and evaluation. The first step for query processing and optimization is done in the mediator and the second step in the wrapper. Unlike the data warehouse, mediators do not hold data. It may have information on the views over the sources provided by the wrapper.

There are many problems in data warehouse systems solved by the mediator-based systems. Data in a warehouse may not be the actual data when accessed because there might be some update transactions running in specific intervals. So at each time, data could be different from what exists in the origin data source. Another problem is to maintain the warehouse. Adding a new data source in a data warehouse system may require reducing of the system including the schemas, and procedures of extract, transform, and load, etc. These two problems are solved in the mediator-based approach. Because of the mediator nature, data is the actual one since all data are stored and retrieved from schema and original sources. On the other hand, adding a new data source requires updating only the global repository. Global repository can be thought of a pool of values, which is used to answer queries.

The focus on designing a mediator- based system is to develop efficient algorithms in order to retrieve the relevant data quickly from the source and reduce/avoid network problems. Data sources are defined independently of each other and can be added or removed from the system with less effort and overhead. Because there is no data stored in a mediator, unlike data warehouse, there is no specific schema definition. Instead, there are some efficient algorithms, which provide quick and reliable access to the data.

To manage all the processes, a mediator needs to have a common ontology as a basis for integration. Data sources can be mapped to same unified concepts via mediator. This allows sources to be changed while system is running and access the latest data as a result. The main task of the system administrator is to keep the metadata repository up-to-date.

Figure 2-3 illustrates a mediator-based system. As shown in the figure, mediator does not need to know exactly about the origin of the data sources. Wrappers act as an interface between the mediator and the data sources. They represent some function to the data transform and some views over the sources. At each source, there is the function, which is needed to transform the input query from mediator to a query to the sources.

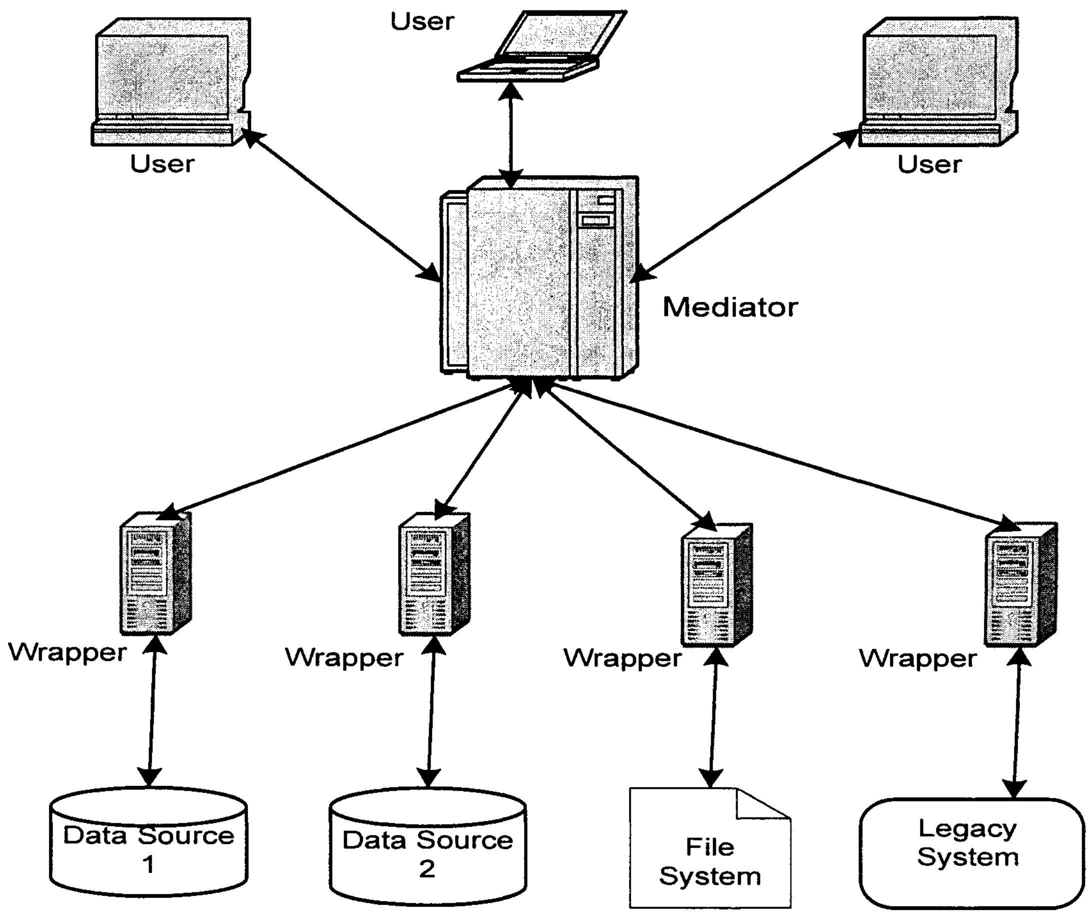


Figure 2-3: Mediator-based system

The mediator dispatches the user query to the appropriate wrappers; it knows which data source has the desired data. The wrappers rewrite the query, which then is executed. The answer is then sent back to the mediator. Mediator is responsible to merge the set of answer returned into one and present it to the user as the query result.

2.3 Related Work

There have been numerous researches on data transformation in general including schema, models, and format transformation [CW'01], which can be divided into main approaches: *coarse-grained* (schema level) and *fine-grained* (instance level). In this context, we study the lineage tracing problem and then discuss works related to ours in this thesis.

2.3.1 Lineage Tracing

An information integration system retrieves data from multiple data sources. The data could be materialized in a unified data store as in data warehouse systems or could be obtained from different sources during query processing, as in mediator-based systems. This data can be analyzed for decision support systems.

For business data, it is useful to provide explanation for a query result, by identifying the data in the sources contributed to the result. This process, tracing data from query answer back to the sources and finding contributing atomic data is known as lineage tracing [CW'01]. It is useful in many areas including: on-line data analysis, processing, and mining (OLAP/OLAM), scientific databases, data cleaning, authorization management, view update problem, and etc.

Let us make it more clearly through the following example about Muhammad Ali, the world's box champion, taken from New York Times. "Ali is said to have Irish roots." Researchers at the Clare Heritage Center claim that Abe Grady came to USA in 1860's and then left for Kentucky and married a black woman. Their son married a black woman and one of the couple's daughters, Odessa Grady, married Cascius Clay, who then had a son named Cascius Clay who changed his name in 1964 to Mohammad Ali. There are enough evidences to this [T'02]. As is clear from this example; it is required to know the origin of the data and the relationship between each transformation.

Some of the works in lineage tracing support schema-level or *coarse-grained* approach, in that, the lineage tracing provides contributed data at the schema level. The other approach is *fine-grained* or instance level, which is more in depth. In general, to support lineage tracing, we need the metadata, original data, and query information itself. Lineage tracing uses a combination of information about atomic data, schema, data transformation, and a number of functions.

Consider again our motivating example of the sales application. We assume that the unified repository for the system has already been created, so we can submit a query using a SQL like language and make a join between two data sources. Consider the following query:

```
SELECT SUM (NumSold) FROM SALES, ITEM
WHERE SALES.ItemID = ITEM.ItemID AND
      ItemName = 'pencil';
```

The query result is 5000. The lineage tracing result is shown as follows.

StoreID	**ItemID**	**NumSold**	**Price**
2	2	2000	2
1	2	3000	1

ItemID	**ItemName**	**Category**
2	Pencil	stationary

Figure 2-4: Lineage tracing result for SALES and ITEM data sources

Identifying the contributing atomic data is desired for a number of users, including information integration designers, administrators, analysts, and system managers.

2.3.2 Lineage Tracing in Data Warehouse Systems

Cui and Widom [CW'01], propose a complete set of techniques and algorithms to retrieve original data contributed to a query answer in data warehouse systems using *fine-grained* approach. Their solution mainly relies on some transformation functions, each of which transforms a set of input data to the output set. Transformation functions satisfy a number of properties listed as follows together with their applications:

a) Dispatcher: Each input data item to *dispatcher* produces zero or more output data items.
b) Filter: Each input data item to *filter* produces a subset of itself.
c) Aggregator: Each input data item is a part of the input partition, including zero or more other input data items. This partition of input data items to *aggregator* produces one output data item.
d) Context-free aggregator: Any two input data items are always in a same input partition or they are never.
e) Key-preserving aggregator: This property is similar to the aggregator, that is, there exists a unique key for each input data item and in an input partition every subset of that partition generates the same output key for the output item.

f) Black-box: This is neither an aggregator nor a dispatcher.

g) Inverse: A transformation T is *invertible* if there exists a transformation T^{-1} such that for each input set I, we have $T^{-1}(T(I)) = I$, and for every output set O, we have $T(T^{-1}(O)) = O$.

There are also some techniques of schema mapping to transfer one input to a new output set. This new output set might be an intermediate set which is supposed to be used by some upcoming transformations.

There are two types of indexing. They propose for optimization of queries, as follows:

a) Conventional: locate data items matching a given search value.

b) Functional: these indexes constructed for a given function F which allow us to quickly locate every data item i such that $F(i) = V$, for a given value V. This type of indexing is used by schema mapping functions.

Based on the input query, transformations can be combined or decomposed. There is a transformation sequence consideration to combine or decompose them. The particular sequence or order of the transformations depends on each transformation itself. Cui and Widom proposed a number of algorithms to classify and combine transformations in an efficient way. Transformations can be done in multiple-input multiple-output way, and they are all *stable* and *deterministic*. A transformation T is *stable* if it never produces spurious output items, i.e., T (null) = null. A transformation is *deterministic* if it always produces the same output, given the same input.

They also developed a prototype in order to demonstrate the feasibility of their solution. There are some techniques employed in the prototype to improve the efficiency, indicating that there is a significant improvement in case of indexing and combining transformations.

2.3.3 Tracing Data Using Schema Transformation Pathways

In this work, we define lineage tracing based on the notions *why-provenance* and *where-provenance* proposed by Fan and Poulovassilis [FP'02]. The former refers to the source data used in processing the query without actually appearing in the query result. On the other hand, *where-provenance* refers to the actual source data that appears in the query result. Corresponding to these two types of data, they introduce two internal data structures, called *affect-pool* and *origin-pool.*

They work on high-level data models, e.g., ER, OO, and relational model, as well as physical data structures. They developed a prototype, called AutoMed, which is based on a lower level data model, HDM, and define high level data models and schema transformation in HDM. They used this common data model to prevent semantic mismatch between models and metadata from different sources. This data model is similar to the internal representation in our work for creating and updating metadata information in ELIT.

The authors also describe how transformation functions affect their two different data pools, called data pool and origin pool. Our *IMDS* and *DRT* tables in ELIT used are similar in sprit to their pools. The difference is in the organization.

In our work, *IMDS* stores only metadata and *DRT* stores data, while this distinction is not there in [8]. In AutoMed, there are some sort of automatically reversible transformation functions to keep track of each change, which might happen to the input data set. AutoMed provides an Intermediate Query Language (IQL) to issue queries over the data sources and models.

Chapter 3

Metadata Collection

In this chapter, we illustrate the schema collection phase for linage tracing in a mediator-based system.

3.1 Collecting Schema Information

We introduce the first phase in our solution approach to support lineage tracing in a mediator-based system. In this phase, we collect information about the schemas of the information sources in the system. This information is independent of any particular query. The assumption, however, is that, the query may require explanation, for which the system has to be prepared, hence justifying this phase.

It is not mandatory to have the original metadata information in a mediator-based system. In such a system there should be a global repository, in which each data source has at least an entry so that the mediator can dispatch the query. On the other hand, for the lineage tracing processes, it is mandatory to have access to metadata while tracing data back to the source.

We assume there is no global repository for the mediator, so we create metadata information if the lineage tracing is in point of interest. There is no need to create such a repository if there is one accessible by mediator.

Collecting schema information requires having some kind of schema mapping functions, since there are heterogeneous data sources, which have to be considered while lineage tracing processes work. This mapping is different from schema mapping for the intermediate tables in a data warehouse system [CW'01].

To support this in our model, we have considered two different schema types: relational database and XML source. For XML sources, schema mapping is done through some functions and a parser that collect the metadata information. Schema mapping in this level contains information about data sources and their individual data items used by the system. Moreover, there are other functions and internal structures which generate and store intermediate data sets produced by each transformation function. Intermediate data sets or results are those created by the transformation functions. It is required to have intermediate data in order to be able to retrieve the next data result in a chain of transformation functions, so they have to be in form of *bags*, since lineage tracing has to provide all the contributed data for a query. For example consider the user query:

```
SELECT STORE.StoreName, sum(SALES.NumSold × SALES.Price)
FROM STORES, SALES, ITEMS
WHERE ITEM.Category  = 'sanitary' AND
      ITEM.ItemID    = SALES.ItemID AND
      STORES.StoreID = SALES.StoreID
      GROUP BY STORE.StoreName;
```

Suppose the user is interested to know the original data including the STORE information. The first step is to produce ItemID's at the contributing tables in the "sanitary" category. This may include redundancy, since there could be several stores having the same product. At this point, the bag semantics should be used in order to include all the relevant data and avoid missing any. The last step is to retrieve the store information from the sources and return it to the user.

Intermediate results are useful to produce the original data if the have been changed. This is, possible by applying some kind of formulas and/or aggregation functions on the intermediate results. They would also be useful to provide user with a "selective" lineage tracing, a feature that allows a user to specify which intermediate data set is he/she interested to be supplied.

Lineage tracing is a backward process from query answer to atomic data contributed to the answer. As a mediator-based system may consist of several heterogeneous data sources [HMNRSW'99], in order to have metadata of these sources and to be able to process a query over them, we need to have an *Integrated Metadata Schema* (*IMDS*) [KR'03, DDL'00]. To create *IMDS*, we need to do some sort of schema mapping [CW'01]. This integrated schema allows us to:

a) Formulate a query referring to both structured and semi structured data in data sources.

b) Support lineage tracing as mentioned before.

The *IMDS* supports the first feature by providing some query components, and supports the second feature by providing specifications of the transformation functions required. The first application of *IMDS* is discussed in section 4.1 to 4.5, while the second one is explained in section 4.6.

In addition to these applications, *IMDS* also reduces the processing overhead. This is because *IMDS* contains "all" metadata information used by current user queries, hence resulting in a significant saving on references to metadata used by the lineage tracing processes. Since *IMDS* is created for lineage tracing purpose, it is not a big repository with all the metadata available in the network. This implementation gives user a GUI to define the metadata repository only for lineage tracing. The system administrator is responsible to do this, to reduce the processing time of lineage tracing.

Query evaluation is done based on the integrated schema model. At this point schema update is a potential problem. Also, we assume the structure of the information does not change meanwhile, which is a reasonable upon noting that schema changes are not frequent. Updating metadata information is the responsibility of the system administrator and can be done based on some specific predefined plan or on demand.

In order to support these functionalities, we naturally assume that information sources are cooperative in the sense that the mediator has access to the actual information sources. The reason for this is that mediator has access to data sources via the wrappers.

There are different ways to define views over the sources in an information integration system, called *global as view - GAV*, and *local as view – LAV*, and *global local as view – GLAV* [L'01]. These approaches provide views for query formulation and processing, and also extracting data from data sources. However, this is not enough to support lineage tracing, and that is why we need to have access to atomic data.

In what follows, we introduce the functions required to support the operations in this phase of collecting schema information. These functions are used to extract metadata of data sources, including table names, column names, data type of data sources, and to store and manage this information as part of the *IMDS*.

These functions are defined as follows, where data source type could be structured or semi-structured. Note that there will be at most one tuple in the *IMDS* of the form ($\mathcal{R}$, $\mathcal{T}$, $\mathcal{S}$), even if various users use or refer to relation (table or XML data source) $\mathcal{R}$ of type $\mathcal{T}$ residing at source $\mathcal{S}$, in many queries.

1. addRelation($\mathcal{R}$, $\mathcal{T}$, $\mathcal{S}$): adds to the *IMDS* a new relation $\mathcal{R}$ with type $\mathcal{T}$.

2. deleteRelation($\mathcal{R}$, $\mathcal{T}$, $\mathcal{S}$): deletes from *IMDS* the definition of relation $\mathcal{R}$ of type $\mathcal{T}$.

3. addColumn($\mathcal{R}$, $\mathcal{C}$): adds to the *IMDS* a new column definition $\mathcal{C}$ in relation $\mathcal{R}$.

4. deleteColumn($\mathcal{R}$, $\mathcal{C}$, $\mathcal{S}$): deletes from the *IMDS* the definition of column $\mathcal{C}$ in relation $\mathcal{R}$.

Consider our Sales application example. Above functions would be as follow:

addRelation (`'SALES', 'Relation','S1'`).

addColumn (`'SALES', 'StoreID'`).

addColumn (`'SALES', 'ItemID'`).

addColumn (`'SALES', 'NumSold'`).

addColumn (`'SALES', 'Price'`).

deleteRelation (`'SALES', 'Relation','S1'`).

deleteColumn (`'SALES', 'StoreID'`).

deleteColumn (`'SALES', 'ItemID'`).

deleteColumn (`'SALES', 'NumSold'`).

deleteColumn (`'SALES', 'Price'`).

3.2 Collecting Schema Information Algorithm

We next introduce an algorithm to collect the schema information. Assume there are k nodes in the system, called $N = (N_1, \ldots, N_k)$, where node N_i includes m data sources $D_1, \ldots, D_m$ such that each D_j is a data source which may be used in a user query. We assume that data sources at each node are unique, which the same data source D_j (with the same schema) may be used in different nodes. So in order to have all the metadata information of all data sources, the input set D has to be changed into D^k, (i.e.) having all data sources from *k* nodes in the system. The output $T = (T_1, \ldots, T_n)$ is a list of tables stored in the internal data structure, *IMDS*, where $n = k \times m$.

```
Input: $D^k$, N     /* The set of data sources involved */
Output: T          /* The IMDS – the metadata of D*/
Algorithm:   Collecting Schema Information – Relational Data Source
    For each $N_i$ in N where i in [1..k]
        For each $D_j$ in $D^i$ where j in [1..m]
            Get the metadata information accessible by mediator.
            Create $T_i$ if there is no such information in IMDS.
    Return.
```

XML metadata collection algorithm retrieves the information from an XSD file. Let $X = (X_1,\ldots, X_n)$ be the input XSD files and $W = (W_1,\ldots, W_m)$ be all the individual words in an XSD files. To collect all the words in all XSD files we use W^k.

```
Input: X, W^k                /* The set of XML sources involved */
Output: T = (T_1,..., T_n)   /* The IMDS – the metadata of T*/
Algorithm:                   Collecting Schema Information – XML Data Source

        For each $X_i$ in X where i in [1..n]
          For each $W_j$ in $W^i$ where j in [1..m]
          Based on each identified word property do
          Create $T_i$ if there is no such information in IMDS.
        Return.
```

Chapter 4

Query Formulation and Processing

In this chapter we introduce the second phase of our solution method to support lineage tracing in mediator-based information integration systems. This phase includes query formulation, processing, and lineage tracing functions.

In a mediator-based environment, each data source can send a query over the network and obtain the results. The mediator is responsible to receive the query, parse it, and dispatch it to each related data source. The mediator then collects and combines the answers, and finally sends it back to the user, which could be a process or a human user.

As mentioned earlier, this process requires that the information about actual sources be available to other sources. In case a local source is a view over the actual data in that source, lineage tracing may not be fully possible as, some actual metadata used in the definition of the view may not be known to the lineage tracing process. Having metadata information allows users to formulate desired queries, and to support lineage tracing.

Having an integrated data model – *IMDS* – helps user to use a generic unified query language to formulate queries over different types of data sources. This query can be mapped to the appropriate query language suitable for each source in the wrapper level, and be executed over the source.

4.1 Visual Query Building

The number of different information systems exists today is tremendous. Query execution over a mediator-based system may require retrieving data from different data sources with different schema types and structures [BDHS'96]. This query might involve structured and semi-structured schemas. For a potential user, who is not an expert in SQL or does not know the data sources structure by hard, it would be very useful to be able to search desired information in these independent systems, without being required to learn the different user interfaces and schemas. In this section, we will introduce a query-building interface, which facilitates formulation of queries over the heterogeneous data sources in our context [TKR'03].

In such a case, the query user developed allows retrieval of data in either of these schemas. We already introduced *IMDS*, an integrated data model for structured and semi-structured schemas, and introduced four mapping functions from one schema to another. The input to a schema mapping function could be structured or semi-structured schemas and the output would be structured, basically a relation with the bag semantics which maintains duplicates. This is important especially when this relation is an intermediate result.

We propose a frame-based query interface, using which a user can formulate queries in a visual environment. Since the unified schema mapping is accessible to the mediator, the user can execute a query assuming the structured format, however both structured and semi-structured schemas will be used to produce the answer and to support lineage tracing. This visual query interface generates a "SQL like" statement, which can then be executed over the mediator-based system. The idea here is to support formulation of queries by non-expert users. Also having such an interface would be useful to have the query components. Query components are used for lineage tracing. More details will be presented in section 4.4

User can easily select tables and related columns from the interface, apply aggregation functions to desired individual columns, build the *where* clause based on the selected tables, and produce a SQL like query as the output.

4.2 Query Type

Different query types can be executed in a mediator-based system. So for lineage tracing, a system should be able to provide query explanation with different query types.

There are different query types, which a user might wish to express and execute. These queries transfer an input data set to the same or different output data set. There would be some different tracing procedures based on the query types [CW'01]. We propose a general lineage tracing function to support different query types. *Query components* are the kind of information required to implement lineage tracing processes. We explain query components in more detail in section 4.4.

A query may have a number of subqueries. Such a query can be rewritten as a simple query without any subqueries producing the same output. Subqueries can be considered as a new data set which one joined with the parent query. In other words, there would be new tables and conditions, which will be added to the selected tables and conditions in the parent query.

This is true also when there is more than one comparison with constant values in the *where* clause of the user query. The following examples illustrate the points.

Suppose, the user query is as follows, which is includes a subquery:

```
SELECT SUM(NumSold) FROM SALES
WHERE ItemID IN (SELECT ItemID  FROM ITEM
                 WHERE ItemName = 'pencil');
```

Then, the rewritten query would be:

```
SELECT SUM(NumSold) FROM SALES,ITEM
WHERE  SALES.ItemID = ITEM.ItemID AND
       ItemName     = 'pencil';
```

As another example, suppose the original user query is:

```
SELECT SUM(NumSold) FROM SALES
WHERE  ItemID IN (1, 2);
```

In this case, the equivalent and rewritten query would be:

```
SELECT SUM(NumSold) FROM SALES,ITEM
WHERE  SALES.ItemID = ITEM.ItemID AND
      (ItemID = 1 OR ItemID = 2);
```

In this work, our main objective is to support lineage tracing for standard SQL queries of *SELECT_PROJECT_JOIN* (SPJ) possibly with aggregation, but without recursion. The current version of our system prototype supports query conditions for 'AND' and 'OR'.

4.3 Query Parsing

Mediator-based systems are responsible for processing queries. This includes retrieving and combining the data using a query language. There are several types of data sources in an information integration system and a query may request data from any of those sources. Therefore, efficient query processing is of primary concern for the users. The first step of query processing is query parsing, followed by query evaluation/execution. We will discuss query evaluation in section 4.6.

Query parsing checks the syntax of the input query. For this, it uses a parser tree. At this point, we need to have the metadata information, which is physically defined in the system. In our implementation, this realized through the *IMDS*, as described before.

Query parsing and analysis is needed before dispatching subqueries over the system. In case of an invalid query, failing to recognize this would result in waste of resources in the system. For query evaluation, we would perhaps need to apply query rewriting and optimization before dispatching the query, in order to reduce network utilization and increase the efficiency of query processing as well as efficiency of lineage tracing processes [PV'99, D'97].

The above steps for lineage tracing are shown in Figure 4-1.

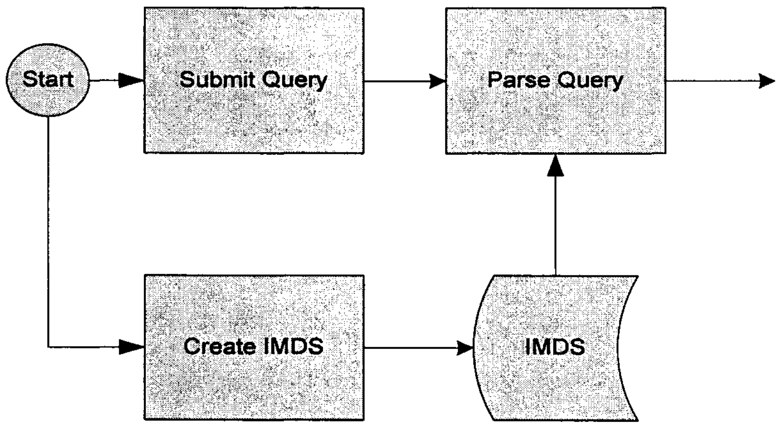


Figure 4-1: Steps in lineage tracing – *IMDS* creation and query parsing

4.4 Query Components

In order to support lineage tracing, we need some transformation functions [CW'01, WQ'97, AKS'96]. The role of these functions is to help users identify the data, which contributed to the query result. Let T be a transformation function with a set S of input data values. The output of T would be R, where R is a subset of S. Note that R could be S, a modified subset of S, or empty. Intuitively, this is used when applying the various conditions in the *WHERE* clause, each of which may result in filtering out tuples which do not contribute to the result. To support lineage tracing, we can view T in the reverse direction as explained before. These types of functions allow us provide query explanation.

To see how transformation functions work, let us consider the functions in Figure 4-2. Transformation function T_1 has the input set S and the output S_1. Transformation function T_2 has the input set S_1 and produces S_2 as its output, and finally T_3 has S_2 as the input and R as its output.

In order to identify the atomic data, it is enough to apply each condition to the data in the *DRT* and obtain the result. But to have the result of each step, the transformation functions must be applied in the *reverses* direction and intermediate results are to be completed. For example, to obtain the atomic data contributed to R after applying T_3, we can obtain S_2 by applying T_3^{-1} to R, i.e. T_3^{-1} (R) = S_2.

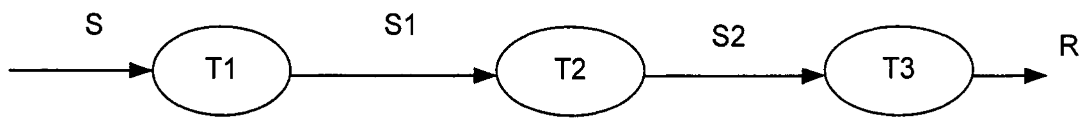


Figure 4-2: Transformation functions execution plan

Transformation functions are created for a query submitted, whenever the query is valid. Each clause in the query transforms the collection of input data from one stage to another. For each query, there are several transformation functions that might be applicable, depending on the query itself. Each *WHERE* clauses, *GROUP BY*, *HAVING*, or any other clauses may transform an input set to a set or bag. In this case, lineage tracing process has to know the transformation functions and their specifications. For each of these functions and the corresponding specifications, it is needed to have all query components to which the appropriate functions are applied. To support linage tracing in an efficient way, transformation functions must be applied in some particular order. Some of these functions reduce the output size, and hence it would be better to apply such functions first.

The following functions are introduced to extract and store data components in *IMDS*:

1. **addQuery** (*Q*, *P*): adds to the *IDMS* the current query *Q* with a unique ID *P* e.g. physical address, process ID, etc.

2. **deleteQuery** (*Q*, *P*): deletes from the *IMDS* query *Q* with the ID *P*.

3. **addTables**(*Q*, *TL*): adds to the *IMDS* the list of tables *TL* contributing to the query *Q*.

4. **deleteTable**(*Q*, *TL*): deletes from *IMDS*, the list of tables *TL* contributing to the query *Q*.

5. **addColumn**(*Q*, *C*): adds to the *IMDS* the list of columns *C* contributing to query *Q*.

6. **deleteColumn**(*Q*, *C*): deletes from *IMDS* the list of columns *C* contributing to query *Q*.

7. **addCondition**(*Q*, *CND*, *CNDT*, *CT*): adds to the *IMDS*, the list of conditions *CND* with type *CNDT* used in query *Q* having the logical operator type *CT*. (i.e., AND, OR, NOT for the *CND* *WHERE*, *GROUP BY*, or *HAVING* clauses).

8. **deleteCondition**(*Q*, *CND*): deletes from the *IMDS*, the list of conditions *CND* used in query *Q*.

The following example illustrates how these functions maybe used when processing a user query. Suppose the following query Q1 is submitted to the system from node P1:

```
SELECT SALES.Price, SALES.NumSold
   FROM ITEM, SALES
   WHERE SALES.ItemID  = ITEM.ItemID AND
         ITEM.Category = 'stationary'
```

Lineage tracing functions below are then applied to *IMDS*, in the order shown.

addQuery (Q1, P1).

addTable (Q1, ''ITEM', 'SALES'').

addColumn (Q1, 'SALES.Price').

addColumn (Q1, 'SALES.NumSold').

addCondition (Q1, 'SALES.ItemID=ITEM.ItemID', 'WHERE', '').

addCondition (Q1, 'SALES.Category='stationary'', 'WHERE', 'AND').

In order to free the resources after executing the lineage tracing processes, we use the following sequence of functions:

deleteQuery (Q1, P1).

deleteTable(Q1, ''ITEM', 'SALES'').

deleteColumn (Q1, 'SALES.Price').

deleteColumn(Q1, 'SALES.NumSold').

deleteCondition (Q1, 'SALES.ItemID=ITEM.ItemID').

deleteCondition(Q1, 'SALES.Category = 'Stationary' ').

4.4.1 Collecting Query Components

Let Q be a query with COL = ((COL_1, F_1), …., (COL_m, F_m)) as selected columns used in Q, and F be the formula applied to those columns. T is a list of data sources referenced/used by Q, where T = (T_1, …, T_n). The query may have some other components such as WHERE, GROUP BY and HAVING clauses. We called them as CON, which may change an input set to another set while processing Q.

They would be in form of CON = ((CON_1, $CONT_1$),…,(CON_k, $CONT_k$)), where $CONT_i$ is the type of condition CON_i. The output would be a set of query components, called COMP, stored in the internal data structure of the system. The steps of query component collecting are formulated in the following algorithm.

```
Input: Q (COL, T, CON)
Output: COMP      /* Query Component */
Algorithm:        Collecting Query Component
 Find all tales $T_j$, where j is in [1..n]
 Find all pair of ($COL_i$, $F_i$), where i is in [1..m]
 Match columns and tables using the information in IMDS.
 Find all conditions $CON_p$, where p is in [1..k].
 Store the information in the internal data structure.
 Return.
```

4.5 Query Evaluation

The SQL-like user query must be mapped to the real data in the source, using the schema information maintained by the mediator system.

Processing a query consists of several basic operations each of which may change the input data to some output. There are several algorithms for implementing each operator. Also there is no predefined algorithm for implementing an operator; this is an optimization issue to decide a particular algorithm at the execution time. Some data factors like relation size; design specifications like existing indexes and even hardware specifications such as memory size are to be considered to implement the operations efficiently.

In a mediator-based system query evaluation has to be done for several heterogeneous data sources. It can be done by the mediator or by the wrapper. Again, in this case, we need to know the actual data schema of the sources and the schema types. It can be done by *IMDS*, however to do it in a more efficient way, more information about physical storage, indexes, partitions, and etc. is needed.

The mediator can have this information in the global schema, and hence it can execute the query and retrieve the result, which are then sent back to the requester, which could be a process or a human user. Figure 4-3 shows the steps of our solution to support lineage tracing.

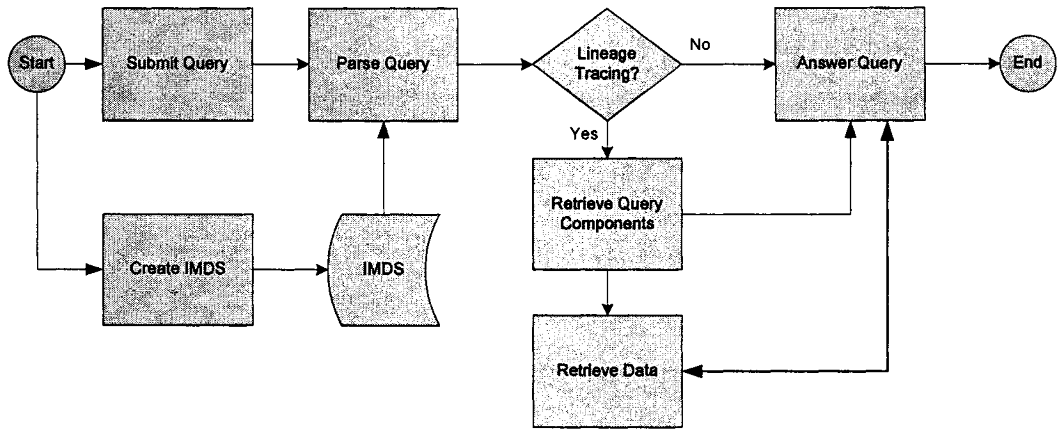


Figure 4-3: Preparation steps for lineage tracing

4.6 Lineage Tracing

Lineage tracing is a backward process, executed to identify atomic data contributed to a query answer. In order to support it, we need to know contributed data sources, and their data, and then apply the transformation function to the input sets in some order.

In section 1.1, we introduced *Data Reference Table* (*DRT*), which is a physical storage to store all data values obtained from data sources while processing a query. Because at the processing time, it is not clear which particular data items will contribute to the answer, we need to collect all data from data sources during this processing. This strategy reduces the risk of reading invalid data contributed to the query answer if the data is updated during query execution.

Having data and metadata in *DRT* enables us to provide intermediate results and also an additional option for the user to do a "selective" lineage tracing. That means a user can select which data item from the involved tables should be considered and traced. The option to have lineage tracing is specified by the user. If a user is not interested in tracing, there would be no over head on the system. [This is similar to the "Explain" feature of *Coral*, by which a user sets the system to provide explanation [ARRSS'93]. For lineage tracing, we have collected enough data of metadata information to which we may just apply the transformation functions defined earlier. For lineage tracing we have the following functions:

1. **addData**(*Q, T, C, V*): adds to the *DRT* the data for query *Q*, table *T*, column *C*, and value *V*.
2. **deleteData**(*Q, T, C, V*): deletes from the *DRT*, the data for query *Q*, table *T*, column *C*, and value *V*.
3. **applyCondition** (*T, C, CND*): applies condition *CND* on column *C* to table *T* and deletes all data which are not satisfies the condition.
4. **explainQuery** (*Q, TL, CL, V*): generates the query explanation for query *Q* based on its table list *TL* and column list *CL* having value *V*.

The application of these functions will appear in the next chapter.

4.6.1 Lineage Tracing Algorithm

The first step in lineage tracing is building the *DRT*. For this purpose, we need the information on the set of data source called T = (T_1,..., T_n) used by the query. T includes data source D_i, and data item COL_j^i , i.e., all necessary columns in each D_i. We also need the information on node N_i. All these information are stores in *IMDS*. *DRT* can be populated, while query is executing.

The inputs to the lineage tracing algorithm are *DRT*, COMP, and the query components. As mentioned above, COMP is of the form $((C_1,T_1), \ldots, (C_h,T_h))$. The output of lineage tracing is a set of tables together with the tuples contributed to the query answer. The output is of the form O (D, COL, V) where D is the data source, COL is the data item in D and V is its value. The conditions mentioned in the following tracing algorithm, OS and TS denote one side and two side conditions.

```
Input: = {$T_1$,..., $T_n$}

Output: DRT

Algorithm: AddData

    Find $T_i$ from IMDS for i in [1..n]

    Fetch data from $D_i$ = {A1,...,Am} in $T_i$.

    For each $COL_i^j$ in $D_i$, create a
      tuple with appropriate tuple id  for that $D_i$ where j in [1..m].

    Return.
```

```
Input: DRT, COMP

Output: O (D, COL,V)

Algorithm: applyCondition

    Find all one side conditions in COMP, named $OS_i$ where i in [1..h]
        Apply $OS_i$ to the DRT and delete those tuples in DRT that do not satisfy the
        condition, $OS_i$.

    Find all two side conditions in COMP named $TS_i$ where i in [1..h]
        Apply $TS_j$ to the DRT and delete those tuples in DRT that do not satisfy the
        condition considering $TS_i$.

    Repeat step 2 until no more tuple is deleted.

    Create the output from DRT from the survived tuples.

    Return.
```

Chapter 5

ELIT: A System Prototype

We have designed and implemented a prototype for lineage tracing in a mediator-based information integration system. We refer to this prototype as *Explanation and LIneage Tracing in Information Integration System* (**ELIT**). We assume the structure of each data source in the system is introduced to ELIT, stored in its internal data structures *IMDS*, for lineage tracing purpose.

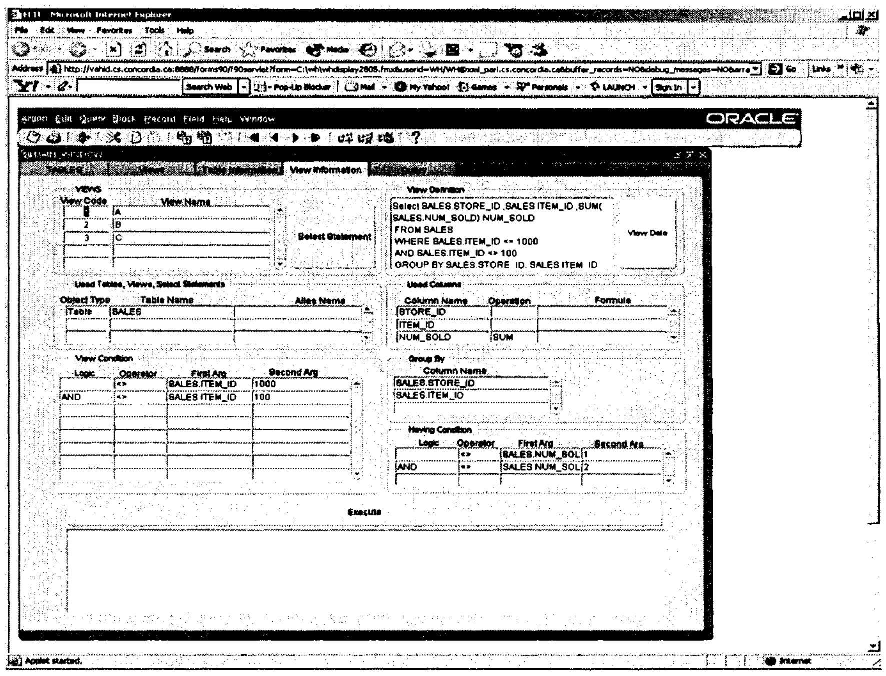
5.1 System Design

ELIT supports both structured and semi structured schemas to support the lineage tracing. For the structured schema, it is needed to map the structures from data sources to the internal data structures. To allow semi-structured data, ELIT includes an XML parser [DHW'01]. Information in the XML file is stored in the internal data structures (*IMDS*), which will be mapped as a relational table in our implementation.

This function retrieves all the information from XML schema file. Having done this step, user has a list of available tables and their columns to formulate valid queries over this schema. In this case, all information sources in the system would be treated as relational sources. So a query in ELIT may involve information from original relational data as well as original XML data, treating them in a uniform way.

In order to retrieve atomic data for a given query, all query components such as column names and query conditions must be retrieved. As shown in Figure 5-1 and 5-2, ELIT has an interface for formulating and submitting queries. The interface displays existing table names, column names, and conditions in "AND" and "OR" format and includes various features to guide users in query formulation.

Once the input is prepared and submitted, ELIT generates a "SQL-like" query corresponding to the input information. This SQL like statement is then parsed for validity of the syntax and the interfaces of metadata information in the query.



VIEWS

View Code	View Name
1	A
2	B
3	C

View Definition

```
Select SALES.STORE_ID ,SALES.ITEM_ID ,SUM(
SALES.NUM_SOLD) NUM_SOLD
FROM SALES
WHERE SALES.ITEM_ID <> 1000
AND SALES.ITEM_ID <> 100
GROUP BY SALES.STORE_ID, SALES.ITEM_ID
```

Used Tables, Views, Select Statements

Object Type	Table Name	Alias Name
Table	SALES	

Used Columns

Column Name	Operation	Formula
STORE_ID		
ITEM_ID		
NUM_SOLD	SUM	

View Condition

Logic	Operator	First Arg	Second Arg
	<>	SALES.ITEM_ID	1000
AND	<>	SALES.ITEM_ID	100

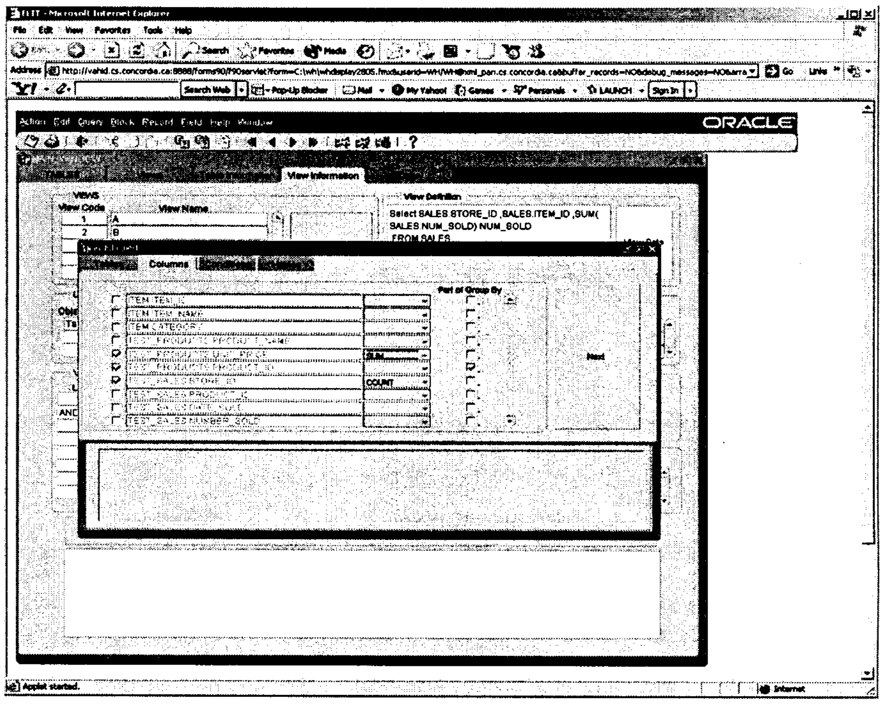
Group By

Column Name
SALES.STORE_ID
SALES.ITEM_ID

Having Condition

Logic	Operator	First Arg	Second Arg
	<>	SALES.NUM_SOL	1
AND	<>	SALES.NUM_SOL	2

Figure 5-1: Storing query information in internal metadata tables


View Definition

```
Select SALES.STORE_ID ,SALES.ITEM_ID ,SUM(
SALES.NUM_SOLD) NUM_SOLD
FROM SALES
```

View Code	View Name
1	A
2	B

	Columns		Part of Group By
☐	ITEM.ITEM_ID		☐
☐	ITEM.ITEM_NAME		☐
☐	ITEM.CATEGORY		☐
☐	TEST_PRODUCTS.PRODUCT_NAME		☐
✓	TEST_PRODUCTS.UNIT_PRICE	SUM	☐
✓	TEST_PRODUCTS.PRODUCT_ID		✓
✓	TEST_SALES.STORE_ID	COUNT	☐
☐	TEST_SALES.PRODUCT_ID		☐
☐	TEST_SALES.DATE_SOLD		☐
☐	TEST_SALES.NUMBER_SOLD		☐

Figure 5-2: User query GUI

ELIT accepts query conditions with "AND" and "OR". Any other types of conditions such as "IN" and "NOT IN" for a predefined set or nested SQL statement is not currently supported in this simple prototype. They can be rewritten as a set of "AND" and "OR" with *inequality* conditions.

5.2 System Illustration

After a query is parsed, the *DRT* is created (Figure 1-3). Regarding our example of `SALES` and `ITEM`, the *DRT* would have *all* the `SALES` and `ITEM` information. It is required to get the `SALES` data from the original relational table and `ITEM` data from XML file, accessible to the mediator, and store them in the *DRT*. This gives a table as shown in Figure 5-3. During data insertion into *DRT*, user can view the result of the query as shown in Figure 5-4.

TableName	***RecordNumber***	***ColumnName***	***Value***
SALES	1	StoreID	2
SALES	1	ItemID	1
SALES	1	NumSold	800
SALES	1	Price	5
….	…	…	…
SALES	10	StoreID	1
SALES	10	ItemID	2
SALES	10	NumSold	3000
SALES	10	Price	1
….	…	…	…
ITEM	1	ItemID	1
ITEM	1	ItemName	binder
ITEM	1	Category	Stationary
…	…	….	….
ITEM	5	ItemID	5
ITEM	5	ItemName	Pot
ITEM	5	Category	Kitchenware

Figure 5-3: *DRT* after inserting metadata and atomic data values

Price	**NumSold**
5	800
2	2000
4	1000
1	3000

Figure 5-4: Query result

All conditions in user query must be applied to the *DRT* and all the records, which do not satisfy the conditions, must be removed from *DRT*. This process can be done in two modes, *batch* and *interactive*. In the batch mode, the system does all the necessary steps for lineage tracing, in a pre-determined sequence of application of lineage tracing functions. In the latter, the user may interact with the system to provide a step-by-step explanation, as desired, through the display interface.

To do this, query conditions should be applied in some order as follows. First, it applies conditions of the form "A θ v", if present in the query, where A is an attribute, v is a value, and θ is a comparison operator. We call this as "one-side-condition." Next, it considers applying conditions of the form "A θ B", which we call as "two-side-condition." Intuitively, this order results in increased efficiency by reducing the number of tuples involved in the join conditions (basically, two-side-conditions).

In our example, *ELIT* looks for a one-side-condition, which is `ITEM.Category`= 'stationary'. It then applies this condition to the *DRT* and finds 2 records with record number 1 and 2 from table `ITEM`, as indicated under the column `TableName` in Figure

5-5. All the records in the *DRT* with record number other than 1 or 2 and TableName 'ITEM' must be marked as deleted.

TableName	***RecordNumber***	***ColumnName***	***Value***
SALES	1	StoreID	2
SALES	1	ItemID	1
SALES	1	NumSold	800
SALES	1	Price	5
....			
SALES	10	StoreID	1
SALES	10	ItemID	2
SALES	10	NumSold	3000
SALES	10	Price	1
....			
ITEM	1	ItemID	1
ITEM	1	ItemName	Binder
ITEM	1	Category	Stationary
ITEM	2	ItemID	2
ITEM	2	ItemName	Pencil
ITEM	2	Category	Stationary

Figure 5-5: *DRT* after applying One-Side-Condition

Next, all two-side-conditions must be applied to the remaining records in *DRT*. Here, we have SALES.ItemID = ITEM.ItemID. Because of the one-side-condition effect, we only have 1 and 2 in *DRT* as ItemID. Therefore, ELIT looks for records with

TableName = 'SALES' and ItemID = 1 or 2. This identifies records 1, 2, 9, and 10, so records with record numbers other than 1,2,9,10 and TableName = 'SALES' must be marked as deleted. Figure 5-6 shows the *DRT* after this step.

TableName	***RecordNumber***	***ColumnName***	***Value***
SALES	1	StoreID	2
SALES	1	ItemID	1
SALES	1	NumSold	800
SALES	1	Price	5
SALES	2	StoreID	2
SALES	2	ItemID	2
SALES	2	NumSold	2000
SALES	2	Price	2
SALES	9	StoreID	1
SALES	9	ItemID	1
SALES	9	NumSold	1000
SALES	9	Price	4
SALES	10	StoreID	1
SALES	10	ItemID	2
SALES	10	NumSold	3000
SALES	10	Price	1
ITEM	1	ItemID	1
ITEM	1	ItemName	Binder
ITEM	1	Category	Stationary
ITEM	2	ItemID	2
ITEM	2	ItemName	Pencil
ITEM	2	Category	Stationary

Figure 5-6: *DRT* after applying Two-Side-Condition

As it can be seen, we have captured the atomic data for the query as well as details of the query answer. This method can be applied to queries with aggregations as well. In this case, we first need to find all the information from the table(s) to which we need to apply the aggregations functions. We can then apply the one-side-condition and two-side-condition processes to identify all the atomic data relevant to the query.

5.3 Architecture of ELIT

ELIT resides in the mediator and has access to the data sources, as well as the mediator. It uses a relational database in order to store metadata and temporary data. This can be done by internal data structures (*IMDS*) in case of having enough resources. *DRT* and metadata information are part of the temporary data which should/could be cleaned up and released after ending lineage tracing processes.

To support multi-user query and lineage tracing in a mediator-based environment, for each query, we consider a unique ID for each query. This ID is based on the node, which issued the query and the query itself. Here having some queries which have been sent before, is an issue for more investigation and one that how this should be treat in order to reduce the time of lineage tracing processes.

ELIT processing model is illustrated in Figure 5-7. It has different modules to support the lineage tracing. "Submit Query" and "Parse Query" are common modules with mediator if tracing is in point of interest or not. "Create IMDS" is the module, which can be used in case of having lineage tracing and no metadata information available in the mediator. After submitting the query, "Retrieve Query Components" and "Create DRT" are those, which have to be done while mediator is running "Retrieve Data". In case of successful ending of all above modules, "Lineage Tracing" returns the atomic data contributed to the query result.

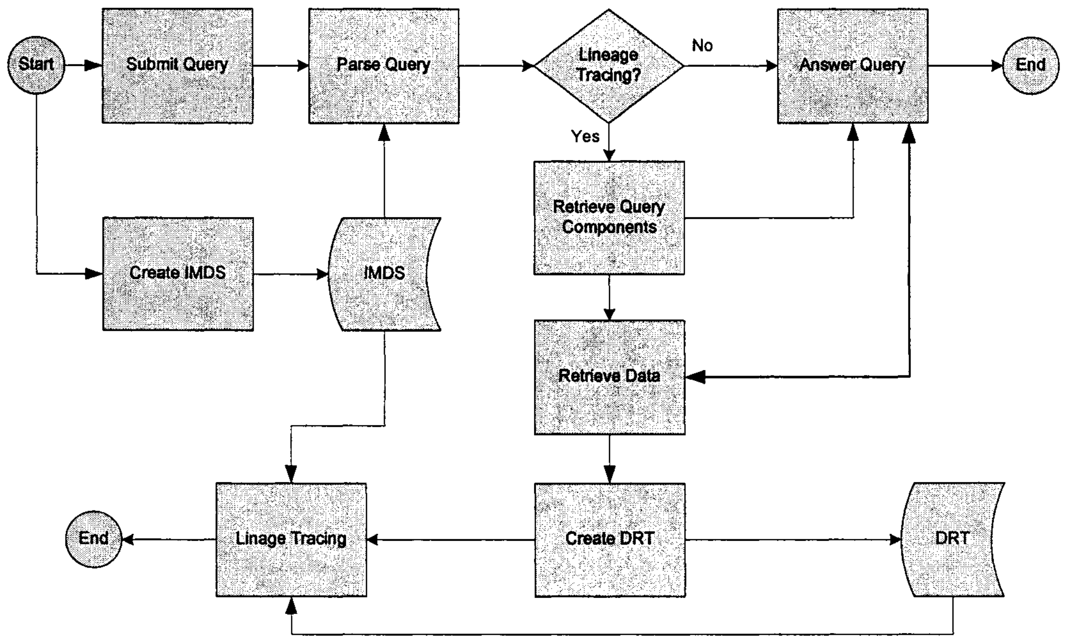


Figure 5-7: ELIT processing model

In terms of development requirements, ELIT has been implemented using Oracle 9i database, Oracle 9i development suite, including Oracle Forms and Oracle Reports builder, Oracle java enabled web server OC4J for development, and windows XP. ELIT also uses a CPU Intel Pentium 4 with a speed of 2.40 GHz and 512 MB for the main memory. Most of the coding is based on PL/SQL language with more than 2000 lines of code. ELIT can run and used in any java enabled web server and on various operating systems such as UNIX, Linux, and any other environment, which has the ability to run java Applets.

5.4 Performance Evaluation

To assess our lineage tracing model and evaluate the performance of our prototype, we generated a range of data sets. ELIT supports two modes of lineage tracing: the *batch mode* and *interactive mode*. In the batch mode, the purpose of lineage tracing is to identify the atomic data contributed to the result. In this mode, the intermediate results are not used in tracing, and hence are not stored during the query processing and lineage tracing. This makes the creation of *DRT* and lineage tracing more efficient. On the other hand, in the interactive mode, we need to keep track of intermediate tables to support requests such as: "which atomic data in the input data set satisfies condition "ITEM.Category = 'stationary' expressed in the user query." This also provides a data analysis mechanism, which is useful and required in some applications.

ELIT has a feature available to users to select atomic data with specific values. For example, in the SALES table with 1,000,000 tuples, there are approximately 100,000 tuples having ItemID = 6. In this case, tuples related to other items are not of interest and hence will be ignored while "preparing" for lineage tracing. Another useful feature in ELIT, called *column selectivity*, allows a user to select which columns he/she would like to be displayed, while tracing.

This is useful in particular when the tables involved have many attributes but only a few of which are of interest to the user. Ideas similar to the value and/or column selection capability can help reduce the time and space for the creation of the *DRT* and its processing. This causes the creation and management of what we call as *partial DRT*, as opposed to the complete *DRT* required in the batch mode of tracing. Figure 5-8, illustrates preliminary result of our performance evaluation of ELIT in supporting batch and interactive modes of lineage tracing.

We have used data sets of 100,000 to 10 million tuples. The size of the partial *DRT* for condition `ItemID` = 6 in the query, the complete *DRT*, as well as the corresponding query processing and lineage tracing times are provided in Figure 5-8. The information in this figure indicates 14 hours to create *DRT* in a non-optimized processing for 1 million records is reduced to 46 minutes for 10 million records, by simply using the above basic ideas of "optimized" processing. For lineage tracing, the processing time reduces from 2.5 minutes to 5 seconds.

There are diagrams from Figure 5-9 to 5-11. They illustrate the ELIT performance improvements from the very basic implementation to enhanced one. We have non-optimized lineage tracing in Figure 5-9. X-axis indicates the number processed records, in the scale of million and Y-axis is the time in the scale of hour. As it is shown, having more records to be processed, make lineage tracing more time consuming.

In the Figure 5-10 and 5-11, Y-axis is the time in the scale of second. As it is clear from figures, there is a big enhancement for the lineage tracing processing time with same number of records,

	Tuples in SALES	Executing User Query	Creating DRT	Linage Tracing	Number of Records in DRT
FULL DRT – Non-Optimized	1,000,000	2”	14 h	-	(3x5) +(4x 1000000) = 4000015
Partial DRT – Non-Optimized	1,000,000	2”	10 ‘ 47”	11’ 47”	(3x2) +(4x 100305) = 401226
Partial DRT - Optimized	1,000,000	2”	2’43”	40”	(3x2) +(4x 100305) = 401226
Partial DRT – Optimized(Using Rename)	100,000	1”	19”	5”	(3x2) +(4x 10019) = 40082
Partial DRT – Optimized(Using Rename)	1,000,000	2”	2’ 43”	19”	(3x2) +(4x 100305) = 401226
Partial DRT – Optimized(Using Rename)	5,000,000	9”	22’ 26”	2’ 28”	(3x2) +(4x 1003050) = 4012206
Partial DRT - Optimized	10,000,000	20”	46’ 15”	3’11”	(3x2) +(4x 501525) = 2006118

Figure 5-8: Lineage tracing evaluation result

A description of the information in Figure 5-8 is as follows:

Tuples in SALES: number of tuples in sales table.

Executing user query: This is the time to run the query. It does not include the time to create the output.

Creating *DRT*: This is the time to create *DRT*.

Lineage Tracing: Time to do the lineage tracing.

Number of Records in *DRT*: In our example we have, (number of ITEM columns × number of ITEM tuples) + (number of SALES columns × number of SALES tuples)

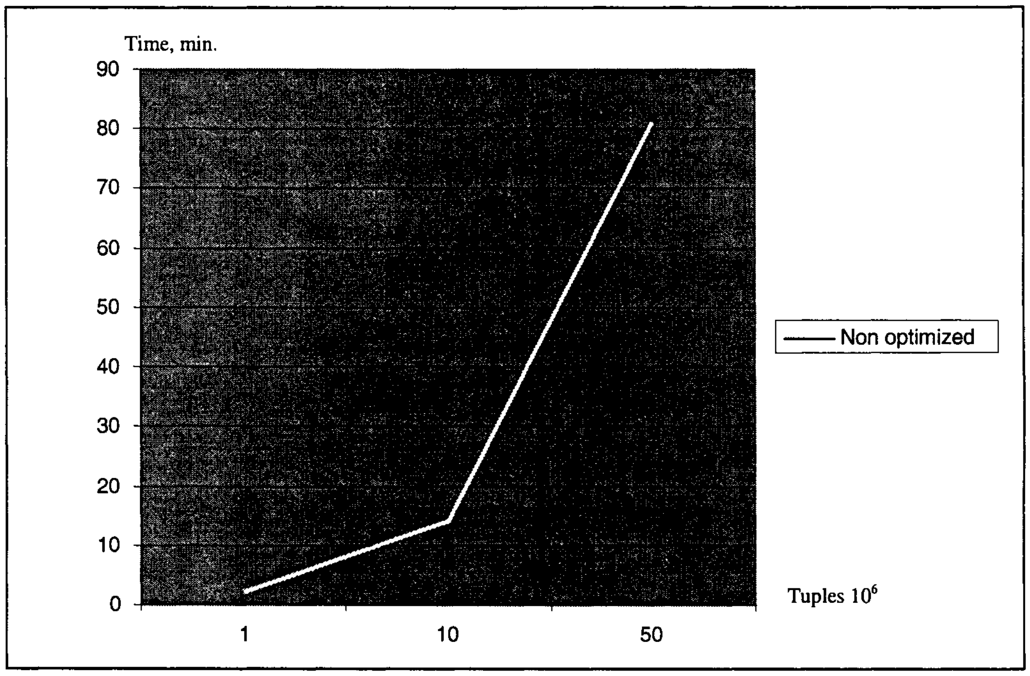


Figure 5-9: Lineage tracing evaluation result – non-optimized

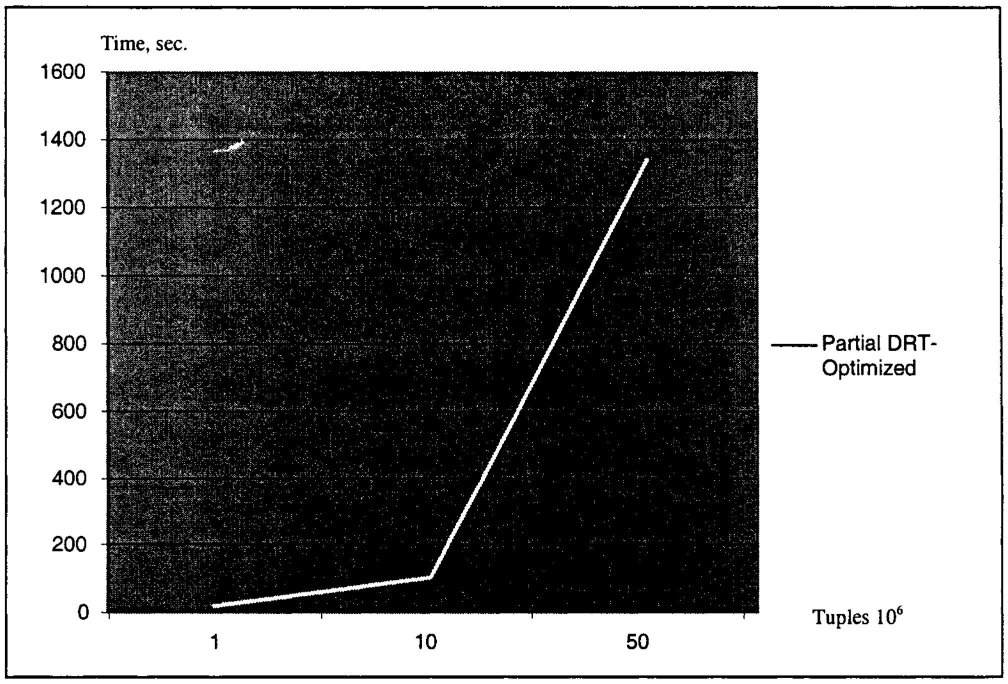


Figure 5-10: Lineage tracing evaluation result – partial *DRT*, optimized

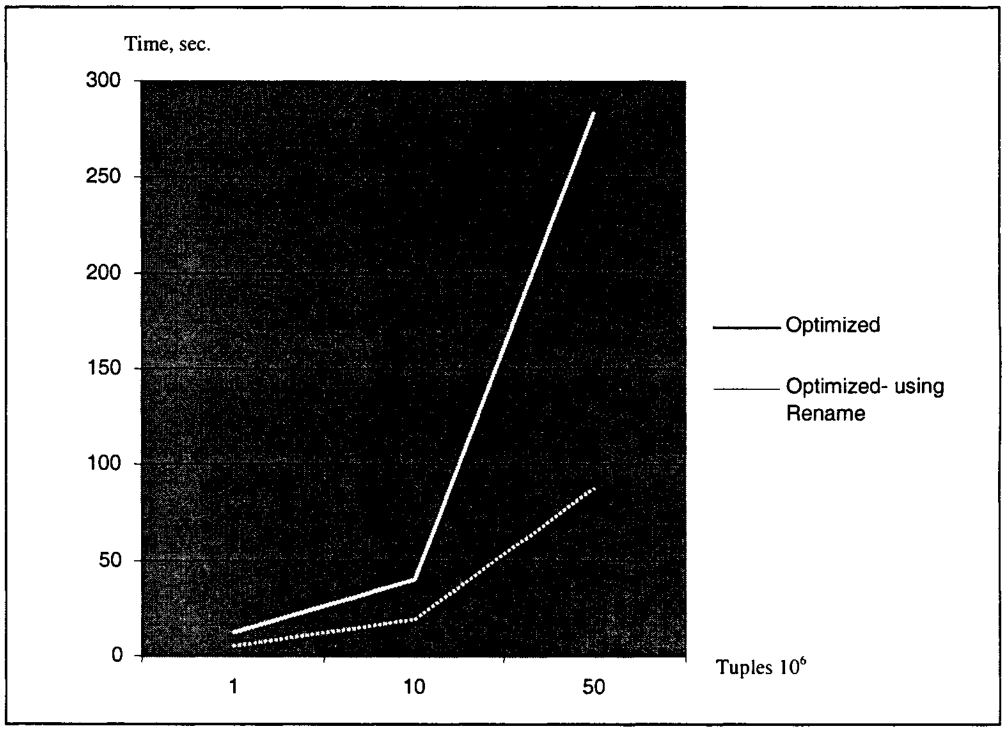


Figure 5-11: Lineage tracing evaluation result – optimized

Chapter 6

Lineage Tracing in Distributed Database Systems

In this chapter, we study the lineage tracing problem in distributed database systems. For this, we first review basic concepts in distributed database systems and then discuss issues and difficulties in supporting lineage tracing in such environments.

6.1 Distributed Database Systems

Information integration systems integrate a uniform access to operational data from multiple (possibly heterogeneous) data sources. Distributed database systems (DDBS) provide such an access to data sources designed at the same time at design stage but implemented to support distributed processing of the data stored at different nodes. A distributed database management system, DDBMS, is software developed to manage DDBS and provides data accessibility to the user in a distributed environment [TV'99].
DDBS take advantage of database and network technologies. Having a central control and management of data is one of the objectives of database systems. On the other hand, the trend in computer network technology is against all centralization efforts. Therefore, in the context of distributed database systems, we need to deploy the two contrasting technologies to provide a uniform accessibility to the data sources over a network. A way

out of this dichotomy is to view database systems as to emphasizing “integration” and not “centralization.” We are not in a position to argue on this issue here but it seems this is exactly what distributed database technology attempts to achieve. In such a DDBS, physical data distribution creates problems, which are not encountered when the database is stored in the same computer. In this context, many components in a centralized DBMS should be revised to support distributed processing of distributed data sources. The revision is in the database design process itself and extends to changes in the DBMS components such as query processing and optimization, metadata management, distributed concurrency control, deadlock management, and reliability. These issues are influenced by the following three factors:

a) The data in a distributed database system may be replicated. In a DDBS, the database design process is the same as a conventional, centralized database. The difference is that a table, or part of it, may be replicated to improve reliability, availability, and/or efficiency of the system. Consequently, for query processing, a DDBMS is responsible for choosing one of the data sources which have the desired data to retrieve data, or in case of transactions, it is responsible for data consistency by ensuring that multiple copies of the same data are identical after update.
b) In case of a communication failure, the system has to make sure that the changes to data will be made to all multiple copies of the same data as soon as the system recovers from the failure.

c) Since there is no global transaction information for a site involved in processing a query or transaction, the synchronization of transactions that access multiple sites is a far complicated task compared to doing this in conventional, centralized database systems.

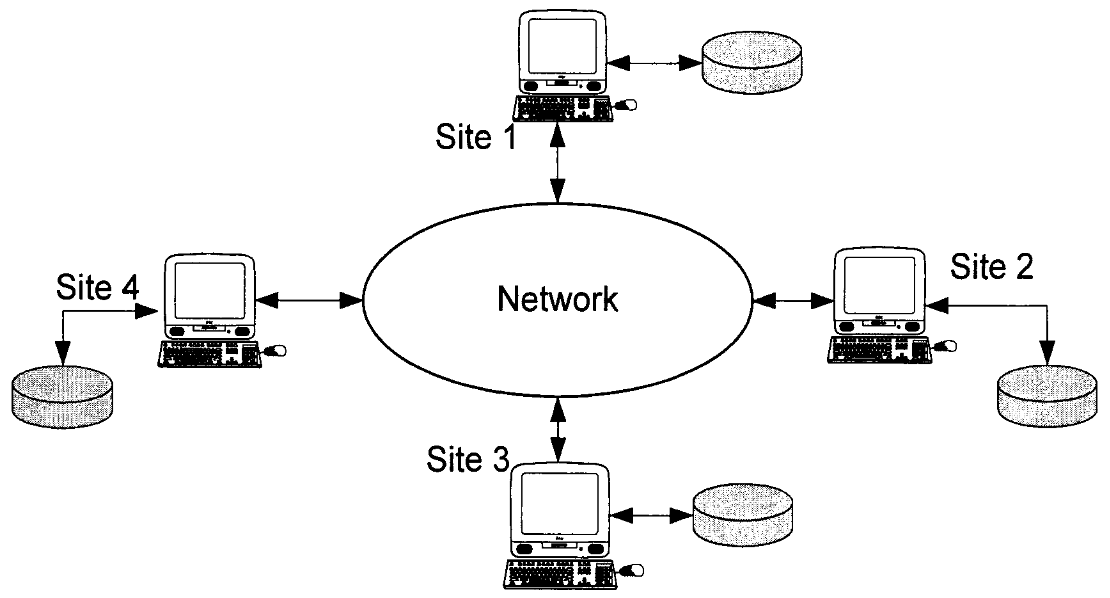


Figure 6-1: Distributed database system architecture

Figure 6-1, [TV'99], shows a distributed database system, which includes four nodes connected through a network. A user query may be (formulated and) issued at any node. The answer to the query is returned to the user by "combining" the results of sub-queries generated by the DBMS from the user query and evaluated over the entire system, in general. Distributed database systems have advantages and disadvantages. The advantages are as follows:

Local Autonomy: Users have their local copy of the shared data, and thus have local control. This allows local policies regarding the use of the data. Such a partitioning and locally authorization is a motivation for distributed database systems.

Improved Performance: Because of the parallelism inherent in distributed systems, performance improvement is possible. Since each site is responsible for only a portion of the data, the CPU and I/O utilization is not as high as in centralized databases. On the other hand, a transaction data may be stored in several sources. Therefore, it is possible to handle transactions in parallel.

Improved reliability/Availability: Since data is replicated, a crash of a node in the system or a failure in the communication link does not necessarily make the data inaccessible. In other words, a node or link crash does not cause total system inoperability. In this case, some data may be inaccessible, but the DDBS can still provide some limited services.

Extensible: In such a system, it is much easier to accommodate increasing database size as well as increasing the processing and storage power to the network.

Data sharing: For many applications, it is more naturally suited to use DDBS, compared to using centralized database systems, to support their daily business rules.

There are also some disadvantages of using distributed database systems, mentioned as follows:

Complexity: Such systems are more complex than centralized databases.

Cost: It is often required to have additional costs for hardware, software, and communication links to support distributed query and transaction processing.

Distribution of control: This was mentioned before as an advantage, however, it results in difficult issues of synchronization and coordination. Adequate policies have to be in place in order to resolve these issues satisfactorily.

Security: It is not difficult to enforce security in the context of one database. However, in a DDBS environment, there is an additional network issue that must be addressed for security.

Fortunately, there are powerful DDBS, which have already provided solution to most of the above issues and problems. For instance, the recent version of IBM WEBSPHERE is an example of such a system [IBM'03]. This has a number of features, which can be used to make a "real" distributed database system. The features include, among others, enterprise search, data federation, data transformation, data placement (caching and replication), data event publishing, augmenting a warehouse with real-time data, building a unified view of customers or products, managing data consistency, distribution, or synchronization across applications, etc [IBM'03].

6.2 Lineage Tracing and Distributed Database Systems

In a DDBS, data sources are connected via a network, in a physical local (LAN) or in a wide-area (WAN) network area. In this context, it is not hard to convince ourselves that providing explanation is useful, or sometime even essential. Some explanations are as follows.

We have already introduced a solution to the problem of lineage tracing in mediator-based systems. Having a global and centralized environment for processing lineage tracing, was a main idea in our solution approach, in which we integrated metadata and the query components information (*IMDS*), together with the data itself to support lineage tracing. Also, in our solution, the lineage tracing processes and routines were conducted on a centralized data store (*DRT*).

The question at this point is how our solution method may be adapted and used in the context of DDBS? In other words, what components in a DDBMS should be changed to support lineage tracing? For each query, for which the user has also indicated that an explanation is desired, the Query Dispatcher has to be changed in order to communicate with other nodes in the system and inform them that, in addition to processing of the query, an explanation is also required. We then need to, create the *IMDS* and *DRT* tables at each node involved in query processing. This means, the Query Processing component has to be changed accordingly. Besides, lineage tracing processes have to be added to the internal DDBMS processes for manipulating data in *DRT* and returning the tracing results. In a way, we can see that many components in a DDBMS need to be revised and reengineered to support the lineage tracing.

One solution to support lineage tracing would be to extend the one discussed for mediator-based, mentioned earlier in this report. In this case, each node in the network can be thought of as a mediator for its data source. For each query, the mediator first collects the information about the metadata and the query, as well as collecting the data itself. The system then provides explanation for the given query, using the lineage tracing functions. It is clear that this solution may require a huge amount of data transferring in the network, which could be prohibitive when the data is very huge. Also centralized storing and processing do not mentioned at the design time. It means that to have such a centralized lineage tracing, it is required to have enough storage media and powerful processor as well as reliable and fast network communication. In addition of this consider any changes, which may be happened to the system while the query processing which causes data inconsistency while creating centralized *IMDS* and *DRT*.

Another solution approach, which is closer to the nature of distributed feature of the systems, is to distribute the lineage tracing processes over the nodes in the network. By doing this, each node, itself, is responsible to do lineage tracing for the sub-query it received, which has been submitted by the requester node. The recipient nodes are also responsible to return the part of the main query explanation, which is related to the sub-query received. At the end, requester collects the answers, merges them, and manipulates the results in order to provide the final query explanation.

As in the first solution, this approach requires having lineage tracing processes at each node, but expectedly in a more efficient away, as there is no need to transfer all the data for building the *DRT* and manipulating it. This is essence is more consistent with the distributed nature of DDBS. This is easier said than done. The problem is efficient implementation of distributed lineage tracing as we need to apply the lineage tracing functions on a collection of distributed (partial) *DRT's*, considering the one-way and two-way conditions discussed earlier, which may not be completely achievable unless the data at multiple sources are gathered in one source in order to finalize the lineage tracing processes. In addition, in a more general setting in which the data may be updated by transactions while lineage tracing is in progress, care must be paid in the construction of the *IMDS* and *DRT* tables. In such a case, an implementation of distributed tracing has to consider all aspects of distributed environment, including distributed transactions, distributed commit, etc.

From the above discussions, it becomes evident that lineage tracing in a distributed database environment is more challenging than a centralized data source, or even in a mediator-based system. Our objective here was more to discuss the problem of lineage tracing in distributed database systems. Providing solution requires further investigations. We just close this chapter by stating that when data is distributed, providing explanation may be more important for data and business analysis.

Chapter 7

Conclusions and Future Work

7.1 Conclusions

Lineage tracing is the problem of recording the history of query processing to provide users with explanation. This problem has been studied extensively in the context of data warehouse systems. In this work, we investigate this problem in mediator-based information integration systems. We identified the difficulties and challenges in this context and made a first attempt to address some of them. The basic idea in our proposed solution is to extract and store "enough" data and metadata information of sources contributed to a query answer. We introduced data structures and algorithms to manage and manipulate the aforementioned information to support lineage tracing.

We have developed a system prototype, ELIT, which includes a number of user interfaces for "building" queries and "displaying" traces. In particular, users can formulate SQL-like queries over structured as well as semi-structured sources through an "input query" interface. ELIT can trace summarized and aggregated data and find atomic data step-by-step in an interactive mode, while query processing is in progress. Our experience in developing ELIT and experimenting with it show viability of the proposed ideas, which can yield useful tools for lineage tracing, by devising more elaborate query optimization techniques.

7.2 Future Work

We are currently working on improving the efficiency of the ELIT, which would be an issue when dealing with large data sets. *DRT* is a large table in which we store all the necessary information to support lineage tracing. More efficient storage management with suitable indexes is needed to manage the size of *DRT*. Another issue that worth investigation, is to compare the proposed framework when the mediator includes just one source which is a relational data source with the lineage tracing method proposed for data warehouse systems to study cost-benefit of ELIT.

The computation model of ELIT can be adapted and used in a peer-to-peer system. To this end, each peer would be an ELIT system having connection to other peers in the system. This of course would be a challenge to process user queries and to support lineage tracing in a large peer-to-peer system with numerous heterogeneous data sources that deserve further research.

When data is distributed, providing explanation may be more important for data and business analysis. This requires an independent study. However, as described, lineage tracing in a distributed database system would be more challenging than a mediator-based system.

References

[AKS'96] Y. Arens, C.A. Knoblock, and W. Shen: *Query reformulation for dynamic information integration*. Journal of Intelligent Information Systems, Volume 6, Number 2/3, Pages 99--130, May 1996.

[ARRSS'93] T. Arora, R. Ramakrishnan, W. G. Roth, P. Seshadri, and D. Srivastava: *The CORAL deductive database system*. In Proc. of 3rd International Conference on Deductive and Object-Oriented Databases, 1993.

[BDHS'96] P. Buneman, S. Davidson, G. Hillebrand, and D. Suciu: *A query language and optimization techniques for unstructured data*. In Proc. of ACM SIGMOD Int'l Conference on Management of Data. Pages 505--516, Montreal, Canada, June 1996.

[CW'01] Y. Cui and Widom: *Lineage tracing for general data warehouse transformations*. In Proc. of VLDB, 2001.

[D'97] O. Duschka: *Query planning and optimization in information Integration*. Stanford University, ISBN: 0-591-90831-X, 1998.

[DDL'00] A. Doan, P. Domings, and A.Y. Levy: *Learning source descriptions for data integration*. In Proc. of the International Workshop on the Web and Databases (WebDB), 2000.

[DHW'01] D. Draper, A.Y. Halevy, and D.S. Weld: *The nimble XML data* *integration system.* In ICDE, pages 155--160, 2001.

[FP'02] H. Fan and A. Poulovassilis: *Tracing data lineage using schema transformation pathways.* In Proc. of Workshop on Knowledge Transformation for the Semantic Web (with ECAI'02), 2002.

[GMUW'01] H. Garcia-Molina, J.D. Ullman, and J. Widom: *Database systems: The complete book.* Prentice Hall, ISBN: 0130319953, 2001.

[HC'03] A. Halevy and C. Li: *Information integration research: Summary of NSF IDM workshop breakout session.* In Proc. of IDM 2003 workshop, 2003.

[HIST'03] A.Y. Halevy, Z.G. Ives, D. Suciu, and I. Tatarinov: *Schema mediation in peer data management.* In Proc. of ICDE 2003.

[HMNRSW'99] L.M. Haas, R.J. Miller, B. Niswonger, M.T. Roth, P.M. Schwarz, and E.L. Wimmers: *Transforming heterogeneous data with database middleware: Beyond integration.* IEEE Data Engineering Bulletin, pages 31--36, 1999.

[I'96] W.H. Inmon: *Building the data warehouse.* John Wiley & Sons, Inc., ISBN: 0-471-14161-5, 1996.

[IBM'03] IBM Corporation: *Business activity management: Your window of opportunity for better business operations*, IBM paper in: http://www.elink.ibmlink.ibm.com/public/applications/publications/cgibin/pbi.cgi?CTY=US&FNC=SRX&PBL=G325-2306, 2003.

[KR'03] A. Kementsietsidis M.A. Ren´ee, and J. Miller: *Mapping data in peer-to-peer systems: Semantics and Algorithmic Issues.* SIGMOD

Conference, Pages 325-336, 2003.

[L'01] M. Lenzerini: *Data integration is harder than you thought.* In Proc. of CoopIS, September 2001.

[WQ'97] J. Widom and D. Quass: *On-line warehouse view maintenance.* In Proc. of International Conference on Management of Data, 1997.

[PV'99] Y. Papakonstantinou and V. Vassalos: *Query rewriting for semi-structured data.* In Proc. of ACM SIGMOD Int. Conf. on Management of Data, pages 455-466, 1999.

[ST'05] N. Shiri and A.T. Azari: *Lineage tracing in mediator-based information integration systems.* In Proc. of 5th IEEE International Symposium and School on Advance Distributed Systems (ISSADS), Mexico, Guadalajara, January 2005.

[T'02] T. Duster: *Tracing Lineage: A Social Project and a Genetic Stamp of Approval.* In Proc. of African Genealogy and Genetics: Looking Back to Move Forward, 2002.

[TKR'03] T. Katchaounov and T. Risch: *Interface capabilities for query processing in peer mediator system.* Technical report 2003-048, Department of Information Technology, Uppsala University, 2003.

[TV'99] M. Tamer Ozsu and P. Valduriez: *Principles of Distributed Database Systems,* Second edition, Prentice-Hall, ISBN: 0-13-659707-6, 1999.

Appendices

In this section, we describe technical details of the design and implementation of ELIT, including hardware and software specifications, internal data model, including ERD and list of all relations, their attributes and constraints. We also provide details of main functions developed in ELIT, and some screen shots of the user interfaces.

Appendix A: System Requirements

To implement ELIT, we have considered the following requirements:

a) Hardware and software availability: ELIT has been implemented as three-tier network architecture. It can be used in an intranet or Internet. For this purpose we used different clients, a web server, and a data server. Each client has a role as data source. The web server role is to simulate the mediator and wrapper in connection with data sources. Having a data server helps us manage the internal ELIT data structures.

b) Appropriate Internet-connection: Reasonable bandwidth in a secure environment is needed for ELIT.

c) Data Communication: Metadata and data stored in different clients are accessible from web sever or mediator.

d) Development requirements: ELIT has been implemented using Oracle 9i database, Oracle 9i DS, Oracle java enabled web server OC4J for development, and windows XP. ELIT can run and used in any java enabled web server and on various operating systems such as UNIX, Linux, and any other environment, which has the ability to run java Applets.

Appendix B: User Interface Specifications

The following lists all forms and their corresponding internal data structures. Data structures are related to the tables and views, which we already defined. For each form, a screen shot will be provided.

Form Name	*Data Block Name*	*Related Table Name*
QUERY	WH_TABLES	WH_TABLES
	CONDITION	N/A
	WH_TABLE_COLUMNS_V	WH_TABLE_COLUMNS_V
	HAVING	N/A
	USER_BLOCK	N/A
WH_DISPLAY	WH_TABLES	WH_TABLES
	WH_TABLE_COLUMNS	WH_TABLE_COLUMNS
	WH_VIEWS	WH_VIEWS
	WH_VIEW_TABLE	WH_VIEW_TABLE
	WH_VIEW_COLUMN_V	WH_VIEW_COLUMN_V
	WH_VIEW_CONDITION_V	WH_VIEW_CONDITION_V
	WH_VIEW_GROUPBY_V	WH_VIEW_GROUPBY_V
	USER_BLOCK	N/A
	XML_BLOCK	N/A
	XML_SOURCE_BLOCK	N/A

Table 1: List of forms and related data blocks and their tables

Appendix C: List of Tables

ELIT internal data structure consists of several tables. These tables support *IMDS* and *DRT* concepts, we introduced in this work. The following table contains table names, column names, and a short description of each column.

Table Name	***Column Name***	***Description***
DATA_REFERENCE_TABLE	VIEW_CODE	Unique view code.
	TABLE_NAME	Table name used by input query.
	RECORD_NUMBER	Record number to make a relation between different DRT Rows.
	COLUMN_NAME	Table name, which input query table, contains.
	VALUE	Column value.
WH_TABLES	TABLE_CODE	Unique table code.
	TABLE_NAME	Table name, which is supposed to be used in a mediator.
	XML_SOURCE	Indicates if the source is XML.
WH_TABLE_COLUMNS	COLUMN_CODE	Unique column code.
	TABLE_CODE	Unique table code.
	COLUMN_NAME	Column Name.
	COLUMN_TYPE	Column type.
WH_VIEWS	VIEW_CODE	Unique view code.
	VIEW_NAME	View name.
WH_VIEW_COLUMNS	VIEW_CODE	Unique view code.
	COLUMN_CODE	Unique column code.
	COLUMN_NAME	Column Name.
	OPERATION	Aggregation operation for the virtual column.
	FORMULA	Having a formula instead of having a table column.

Table Name	*Column Name*	*Description*
WH_VIEW_CONDITION	VIEW_CODE	Unique view code.
	SEQ_NUM	Unique sequence number.
	PARENT_CODE	In case of having a sub query in a query, this column indicates the parent query, which has been already defined.
	KEYWORD_TYPE	Condition operator like 'AND' and 'OR'.
	OPERATOR	Comparison operator.
	FIRST_ARG_TEXT	Text argument in case of having text instead of table column.
	FIRST_ARG_CODE	Table column_code.
	SECOND_ARG_TEXT	Text argument in case of having text instead of table column.
	SECOND_ARG_CODE	Table column_code.
WH_VIEW_GROUPBY	SEQ_NUM	Unique sequence number.
	VIEW_CODE	Unique view code.
	COLUMN_CODE	Column code.
WH_VIEW_HAVING	VIEW_CODE	Unique view Code.
	SEQ_NUM	Unique sequence number.
	PARENT_CODE	In case of having a sub query in a query, this column indicates the parent query, which has been already defined.
	KEYWORD_TYPE	Comparison operator.
	OPERATOR	Text argument in case of having text instead of table column.
	FIRST_ARG_TEXT	Table column_code.

Table Name	*Column Name*	*Description*
	FIRST_ARG_CODE	Text argument in case of having text instead of table column.
	SECOND_ARG_TEXT	Table column_code.
	SECOND_ARG_CODE	Unique sequence number.

Table 2: List of internal tables

Appendix D: ERD Diagram of the Main Tables

In this section, we introduce the entity relationship diagram ERD) of the entity sets and relationships in ELIT. The main relations used in ELIT are *WH_TABLES* and *WH_TABLE_COLUMNS*, which hold the metadata information. If a mediator has its own repository and metadata information, these tables are not required.

Table *WH_VIEWS* refers to the entity set, which stores query information. For the implementation purposes and working environment, we treat each submitted query as a view. We remark that it is not a real view of the system but only a concept that we can refer to later.

For storing information about query components, we used two main entity sets: *WH_VIEW_CONDITION* and *WH_VIEW_HAVING*. These entity sets are used by lineage tracing transformation functions. As explained before, for each individual condition in a WHERE clause or HAVING clause, there is a tuple in the tables corresponding to these sets. As discussed earlier in chapter 5, these tables have to be cleaned up after lineage tracing.

DATA_REFERENCE_TABLE or DRT is the table that holds the metadata and data. This helps to have intermediate results based on the user interest.

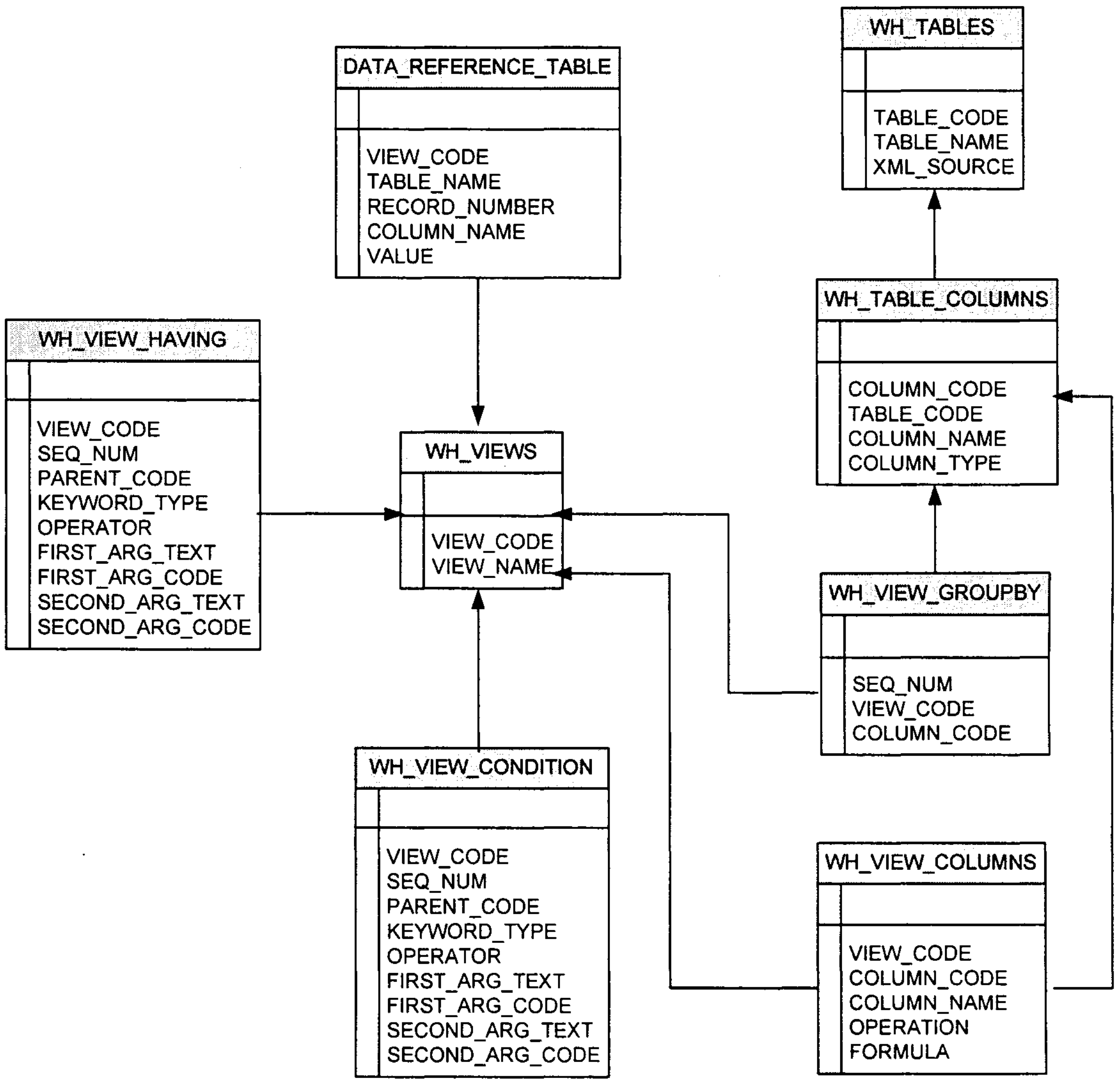


Figure A-1: Entity relationship diagram

Appendix E: List of Views

The following table lists system internal views and their definitions.

View Name	*Definition*
WH_TABLE_COLUMNS_V	V_TABLE_COLUMNS (TABLE_NAME, TABLE_CODE, COLUMN_NAME, COLUMN_CODE, TABLE_COLUMN) AS select a.table_name, a.table_code, b.column_name, b.column_code , a.table_name||'.'||b.column_name table_column from wh_tables a , wh_table_columns b where a.table_code = b.table_code
WH_VIEW_COLUMN_V	WH_VIEW_COLUMN_V (COLUMN_CODE,VIEW_CODE, VTV_NUM, COLUMN_NAME, OPERATION, FORMULA, TABLE_CODE, COLUMN_TYPE) AS select a.VC_NUM column_code ,a.VIEW_CODE view_code ,a.VTV_NUM vtv_num ,b.COLUMN_NAME column_name ,a.OPERATION operation ,a.FORMULA formula ,b.table_code table_code ,b.column_type column_type from wh_view_columns a, wh_table_columns b wherea.column_code = b.column_code
WH_VIEW_CONDITION_V	WH_VIEW_CONDITION_V (VIEW_CODE, SEQ_NUM, PARENT_CODE, KEYWORD_TYPE, OPERATOR, FIRST_ARG, FIRST_ARG_CODE, SECOND_ARG, SECOND_ARG_CODE, F_TABLE_NAME, F_COLUMN_NAME, S_TABLE_NAME, S_COLUMN_NAME) AS select a.view_code view_code, a.seq_num seq_num, a.parent_code parent_code, a.keyword_type keyword_type, a.operator operator, decode(a.first_arg_text,null, (f1.table_name ||'.'||b.column_name), a.first_arg_text) first_arg, a.first_arg_code first_arg_code, decode (a.second_arg_text,null, (f2.table_name ||'.'||c.column_name), a.second_arg_text) second_arg, a.second_arg_code second_arg_code, f1.table_name f_table_name , b.column_name f_column_name,

View Name	***Definition***
	f2.table_name s_table_name ,
	c.column_name s_column_name
	from
	wh_view_condition a,
	wh_table_columns b,
	wh_table_columns c,
	wh_tables f1,
	wh_tables f2
	where
	a.first_arg_code = b.column_code(+)
	and a.second_arg_code = c.column_code(+)
	and b.table_code = f1.table_code(+)
	and c.table_code = f2.table_code(+)
	WH_VIEW_GROUPBY_V (SEQ_NUM, VIEW_CODE, COLUMN_NAME)
	AS select
	a.seq_num seq_num,
	a.view_code view_code,
	c.table_name ||'.'||b.column_name column_name
	from
WH_VIEW_GROUPBY_V	wh_view_groupby a,
	wh_table_columns b,
	wh_tables c
	where
	a.column_code = b.column_code
	and b.table_code = c.table_code
	WH_VIEW_HAVING_V (VIEW_CODE, SEQ_NUM, PARENT_CODE,
	KEYWORD_TYPE, OPERATOR, FIRST_ARG, FIRST_ARG_CODE,
	SECOND_ARG, SECOND_ARG_CODE, F_TABLE_NAME,
	F_COLUMN_NAME, S_TABLE_NAME, S_COLUMN_NAME
	) AS select
	a.view_code view_code,
	a.seq_num seq_num,
	a.parent_code parent_code,
	a.keyword_type keyword_type,
	a.operator operator,
	decode(a.first_arg_text,null,
	(f1.table_name ||'.'||b.column_name),
	a.first_arg_text) first_arg,
	a.first_arg_code first_arg_code,
WH_VIEW_HAVING_V	decode (a.second_arg_text,null,
	(f2.table_name ||'.'||c.column_name),
	a.second_arg_text) second_arg,
	a.second_arg_code second_arg_code,
	f1.table_name f_table_name ,
	b.column_name f_column_name,
	f2.table_name s_table_name ,
	c.column_name s_column_name
	from
	wh_view_having a,
	wh_table_columns b,
	wh_table_columns c,
	wh_tables f1,
	wh_tables f2
	where
	a.first_arg_code = b.column_code(+)
	and a.second_arg_code = c.column_code(+)

View Name	*Definition*

	`and b.table_code       = f1.table_code(+)`
	`and c.table_code       = f2.table_code(+)`

WH_VIEW_TABLE	`WH_VIEW_TABLE ( VTV_NUM, PARENT_TYPE, VIEW_NAME,`
	`VIEW_CODE, OBJECT_TYPE, TABLE_NAME, TABLE_CODE,`
	`ALIAS_NAME ) AS select`
	`a.vtv_num                      vtv_num,`
	`'View'                         parent_type,`
	`c.view_name                    view_name ,`
	`c.view_code                    view_code,`
	`Decode(tvs_type, 'T','Table') object_type,`
	`b.table_name                   table_name,`
	`b.table_code                   table_code,`
	`a.alias_name                   alias_name`
	`from wh_view_table_view a,`
	`     wh_tables          b,`
	`     wh_views           c`
	`where parent_type = 'V'`
	and a.parent_code = c.view_code
	and a.tv_code = b.table_code

Table 3: List of internal views and their definition

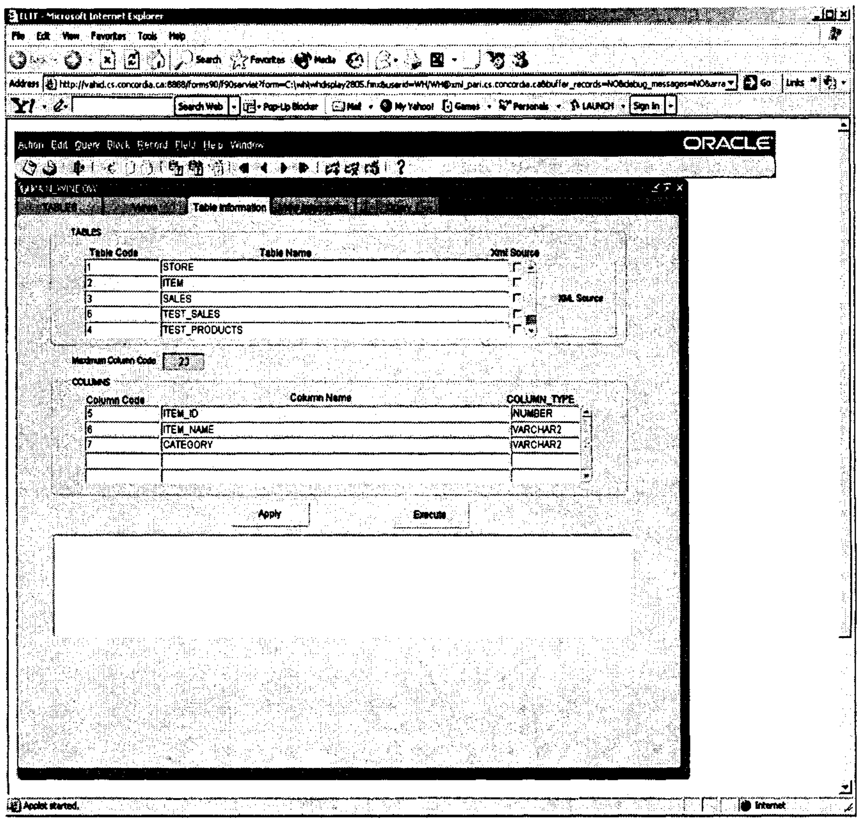
Appendix F: List of User Interfaces in ELIT

This section provides screen shots and main functions for each individual user interfaces in ELIT. The order of presentation of these interfaces in this section is based on their functionality as query formulation, processing, and tracing proceeds. We start with the interface that manages the metadata information. It is used to accept, store, and update the metadata information.

F-1: Table Definition

This GUI has been created to define new tables in a mediator-based system. Data come from different data sources. Their sources could be structured or semi-structured models. We consider relational and XML data and develop appropriate mapping functions. For the XML files, we have introduced additional functions to read and map the XML structure information to relational model.

We next introduce the query component interface. Using this one, a user creates a query and submits it to the system for validation and processing. Query components can be stored in the internal structures as soon as the query is parsed and evaluated correctly. While system processes the query, the required data is stored in *DRT.* This technique may prevent data to become inconsistent, when lineage tracing process is activated by the user.



TABLES

Table Code	Table Name	Xml Source
1	STORE	☐
2	ITEM	☐
3	SALES	☐
6	TEST_SALES	☐
4	TEST_PRODUCTS	☐

XML Source

Maximum Column Code 23

COLUMNS

Column Code	Column Name	COLUMN_TYPE
5	ITEM_ID	NUMBER
6	ITEM_NAME	VARCHAR2
7	CATEGORY	VARCHAR2

Figure A-2: Table definition GUI

F-2: XML Source Selection

The purpose of this canvas is having a graphical user interface to the network in order to select the XML and/or XSD file.

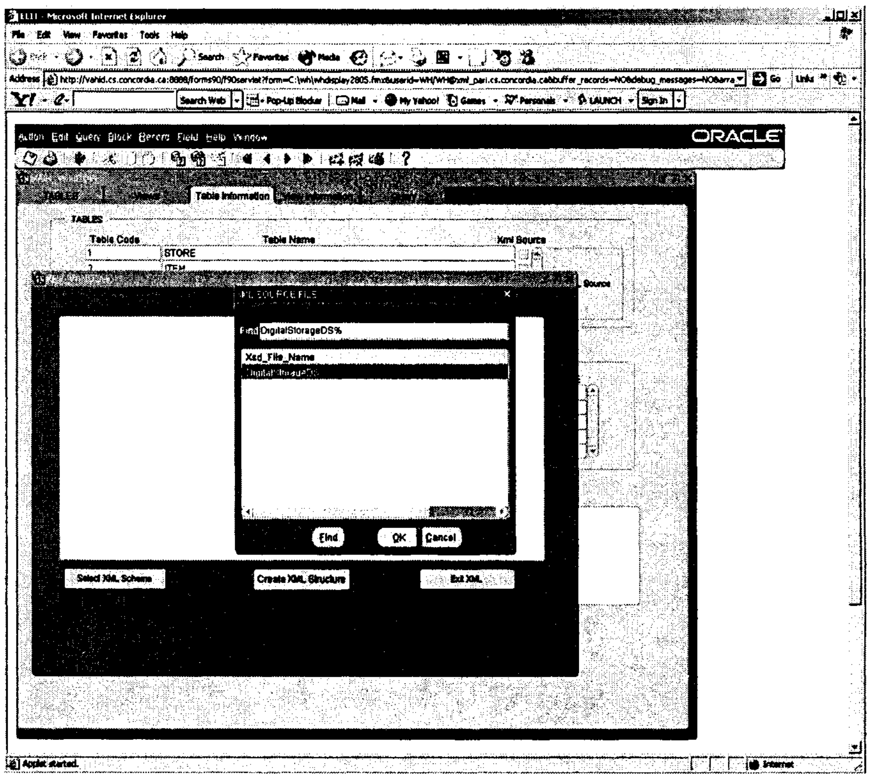


Figure A-3: XML source selection GUI

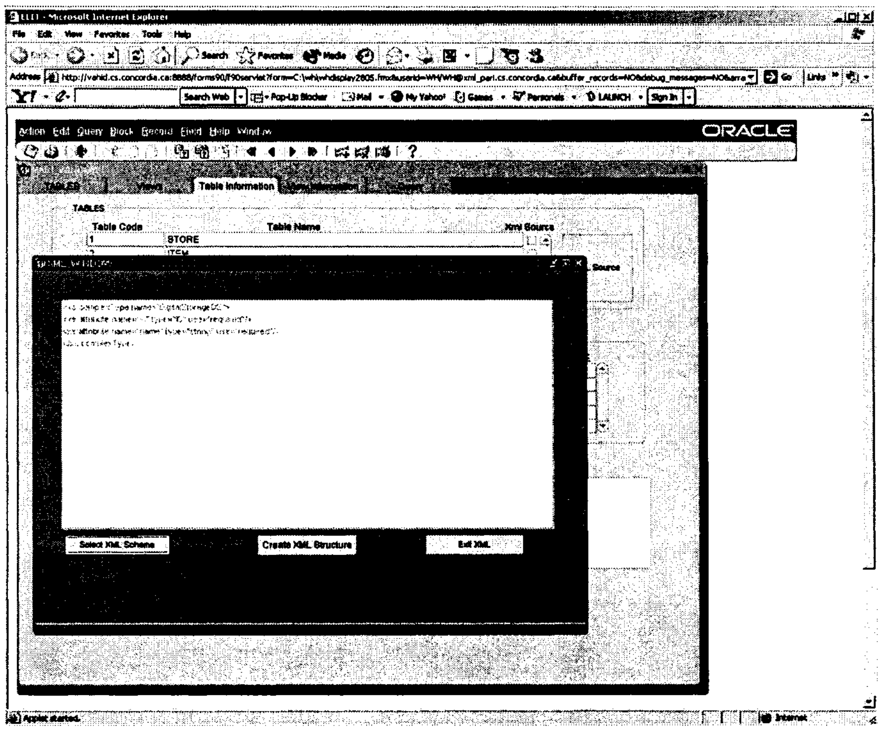
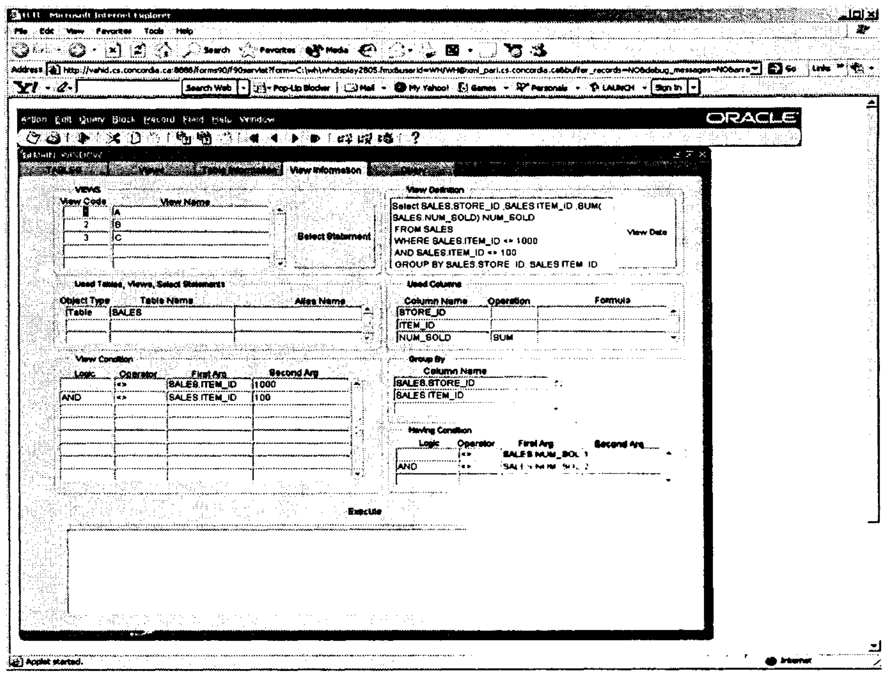


Figure A-4: XML structure view GUI

F-3: Query Component Creation

This GUI provides user a visual environment over the Web to define queries. Using this, the user can write a unified query over heterogeneous data sources and enables the system to store the query components.



VIEWS

View Code	View Name
1	A
2	B
3	C

View Definition

```
Select SALES.STORE_ID ,SALES.ITEM_ID ,SUM(
SALES.NUM_SOLD) NUM_SOLD
FROM SALES
WHERE SALES.ITEM_ID <> 1000
AND SALES.ITEM_ID <> 100
GROUP BY SALES.STORE_ID, SALES.ITEM_ID
```

Used Tables, Views, Select Statements

Object Type	Table Name	Alias Name
Table	SALES	

Used Columns

Column Name	Operation	Formula
STORE_ID		
ITEM_ID		
NUM_SOLD	SUM	

View Condition

Logic	Operator	First Arg	Second Arg
	<>	SALES.ITEM_ID	1000
AND	<>	SALES.ITEM_ID	100

Group By

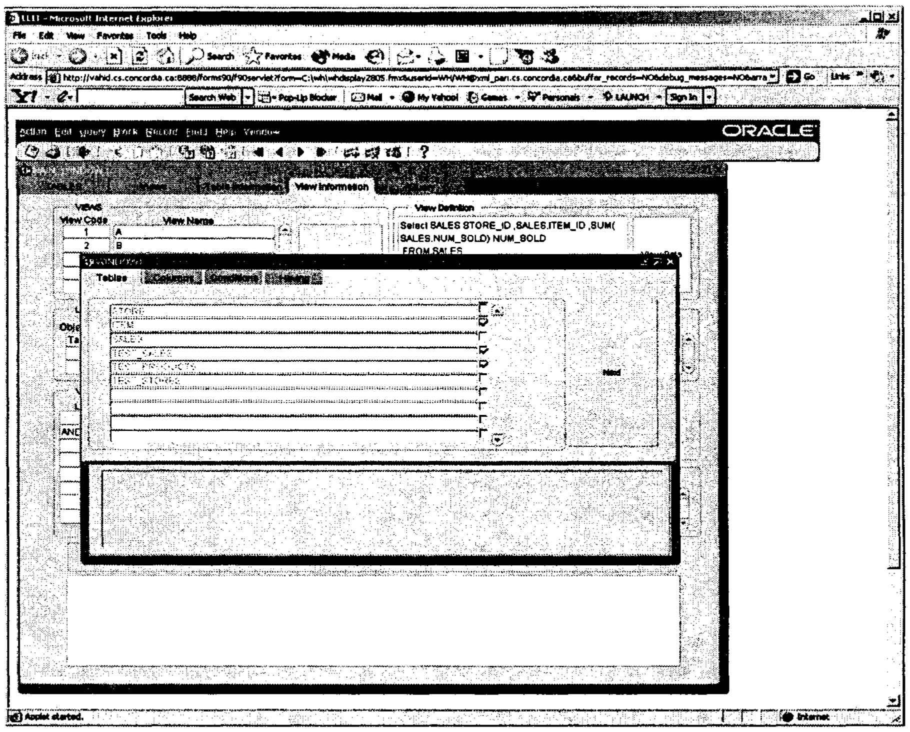
Column Name
SALES.STORE_ID
SALES.ITEM_ID

Having Condition

Logic	Operator	First Arg	Second Arg
	<>	SALES.NUM_SOL 1	
AND	<>	[illegible] 2	

Figure.A-5: View definition GUI

This interface contains several screens to select desired sources and their data items. The sources are already defined in the *IMDS* table. This prevents any syntax or semantic errors. User has the ability to create any SQL query, even those with aggregation functions.

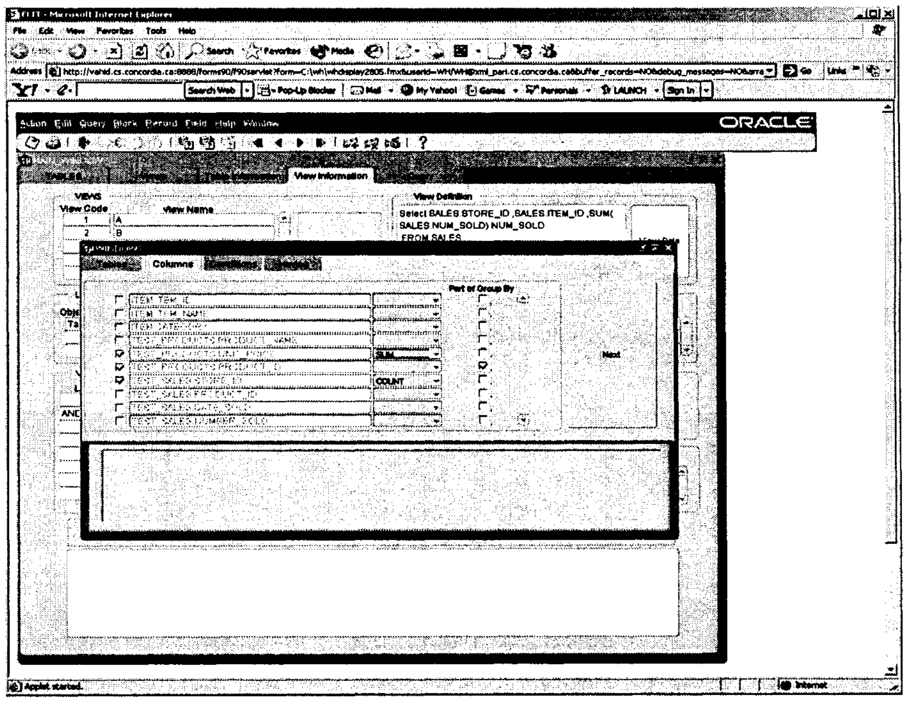


View Code	View Name
1	A
2	B

```
Select SALES STORE_ID ,SALES.ITEM_ID ,SUM(
SALES.NUM_SOLD) NUM_SOLD
FROM SALES
```

STORE	☐
ITEM	✓
SALES	☐
TEST_SALES	✓
TEST_PRODUCTS	✓
TEST_STORES	☐

Figure A-6: Table selection GUI

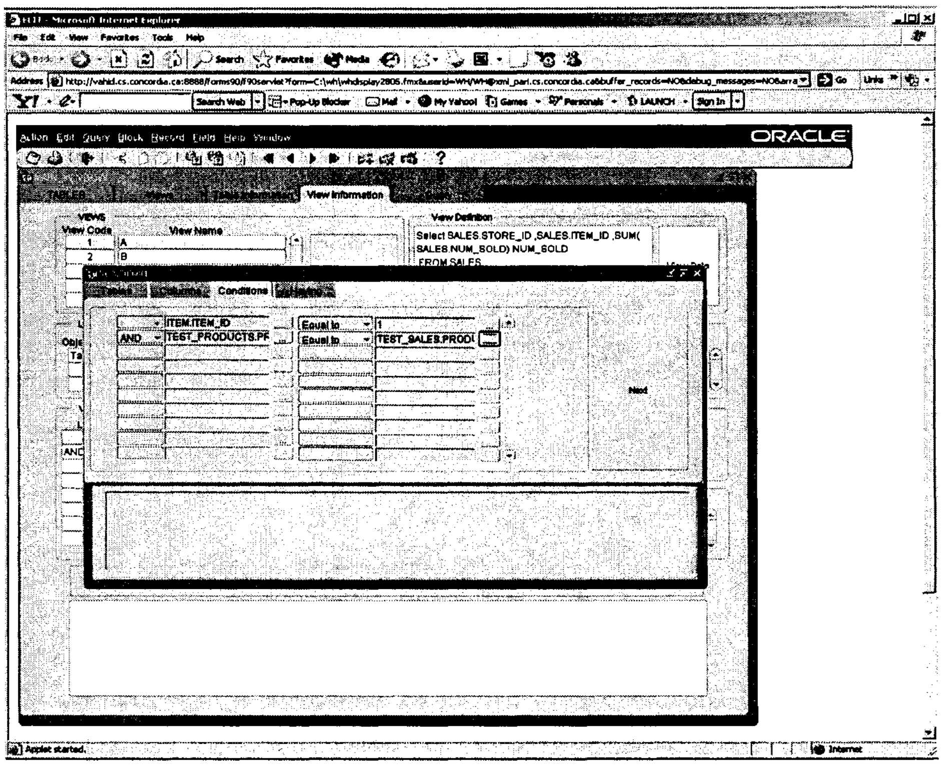


View Code	View Name
1	A
2	B

```
Select SALES.STORE_ID ,SALES.ITEM_ID ,SUM(
SALES.NUM_SOLD) NUM_SOLD
FROM SALES
```

			Part of Group By
☐	ITEM.ITEM_ID		☐
☐	ITEM.ITEM_NAME		☐
☐	ITEM.CATEGORY		☐
☐	TEST_PRODUCTS.PRODUCT_NAME		☐
✓	TEST_PRODUCTS.UNIT_PRICE	SUM	☐
✓	TEST_PRODUCTS.PRODUCT_ID		✓
✓	TEST_SALES.STORE_ID	COUNT	☐
☐	TEST_SALES.PRODUCT_ID		☐
☐	TEST_SALES.DATE_SOLD		☐
☐	TEST_SALES.NUMBER_SOLD		☐

Figure A-7: Column selection GUI

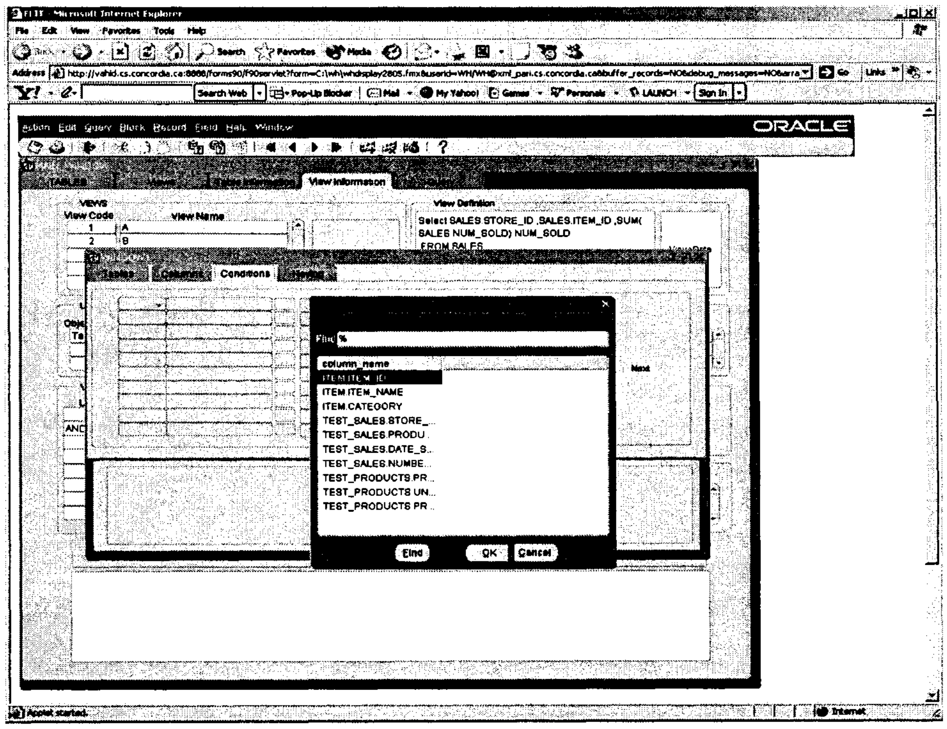


View Code	View Name
1	A
2	B

Select SALES.STORE_ID ,SALES.ITEM_ID ,SUM(SALES.NUM_SOLD) NUM_SOLD
FROM SALES

	ITEM.ITEM_ID	Equal to	1
AND	TEST_PRODUCTS.PF	Equal to	TEST_SALES.PRODI

Figure A-8: Where condition GUI



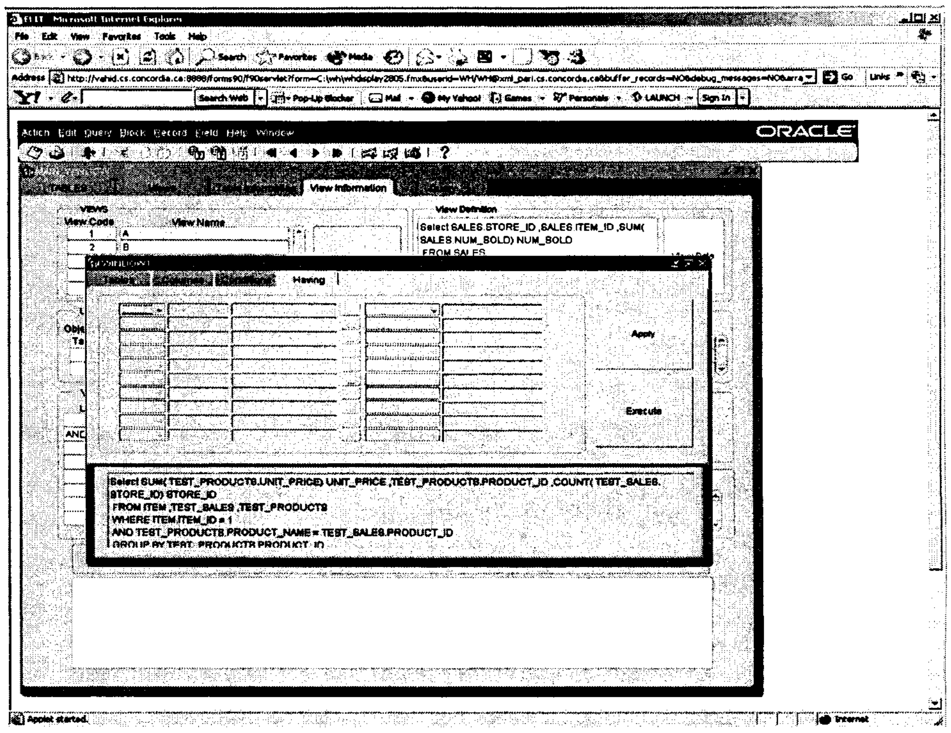
View Code	View Name
1	A
2	B

Select SALES.STORE_ID ,SALES.ITEM_ID ,SUM(SALES.NUM_SOLD) NUM_SOLD
FROM SALES

Find %

column_name
ITEM.ITEM_ID
ITEM.ITEM_NAME
ITEM.CATEGORY
TEST_SALES.STORE_...
TEST_SALES.PRODU...
TEST_SALES.DATE_S...
TEST_SALES.NUMBE...
TEST_PRODUCTS.PR...
TEST_PRODUCTS.UN...
TEST_PRODUCTS.PR...

Figure A-9: Where condition, column selection GUI



```
Select SALES.STORE_ID ,SALES.ITEM_ID ,SUM(
SALES.NUM_BOLD) NUM_BOLD
FROM SALES
```

```
Select SUM( TEST_PRODUCTS.UNIT_PRICE) UNIT_PRICE ,TEST_PRODUCTS.PRODUCT_ID ,COUNT( TEST_SALES.
STORE_ID) STORE_ID
FROM ITEM ,TEST_SALES ,TEST_PRODUCTS
WHERE ITEM.ITEM_ID = 1
AND TEST_PRODUCTS.PRODUCT_NAME = TEST_SALES.PRODUCT_ID
GROUP BY TEST_PRODUCTS.PRODUCT_ID
```

Figure A-10: Final select statement GUI

F-4: Lineage Tracing Process

This interface provides user choices to guide system to support lineage tracing. The answer can be illustrated in an interactive mode, after applying the first side condition, two side conditions, or complete lineage tracing. Each level generates a report containing the tables, columns, and atomic data involved in providing query answers.

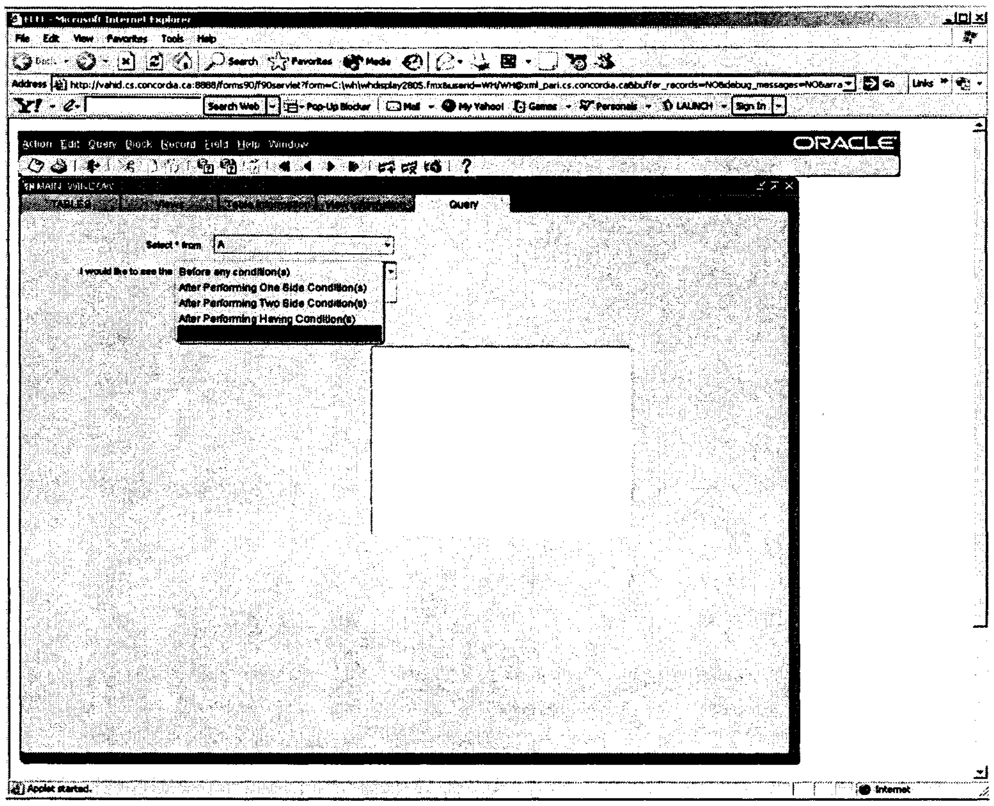

Action Edit Query Block Record Field Help Window
ORACLE
MAIN WINDOW
TABLES
Views
Query
Select * from
A
I would like to see the
Before any condition(s)
After Performing One Side Condition(s)
After Performing Two Side Condition(s)
After Performing Having Condition(s)
Applet started.
Internet

Figure A-11: Lineage tracing GUI