Dynamic Integration of Evolving Distributed Databases using Services

WENG, BIN

Dynamic Integration of Evolving Distributed Databases using Services

Name: Bin Weng

Supervisor: Prof. Keith Bennett

Ph.D. Thesis

School of Engineering
University of Durham

2009

Abstract

This thesis investigates the integration of many separate existing heterogeneous and distributed databases which, due to organizational changes, must be merged and appear as one database. A solution to some database evolution problems is presented. It presents an *Evolution Adaptive Service-Oriented Data Integration Architecture (EA-SODIA)* to dynamically integrate heterogeneous and distributed source databases, aiming to minimize the cost of the maintenance caused by database evolution.

An algorithm, named *Relational Schema Mapping by Views (RSMV)*, is designed to integrate source databases that are exposed as services into a pre-designed global schema that is in a *data integrator service*. Instead of producing hard-coded programs, views are built using *relational algebra operations* to eliminate the heterogeneities among the source databases. More importantly, the definitions of those views are represented and stored in the *meta-database* with some constraints to test their validity. Consequently, the method, called *Evolution Detection*, is then able to identify in the meta-database the views affected by evolutions and then modify them automatically.

An evaluation is presented using *case study.* Firstly, it is shown that most types of heterogeneity defined in this thesis can be eliminated by RSMV, except semantic conflict. Secondly, it presents that few manual modification on the system is required as long as the evolutions follow the rules. For only three types of database evolutions, human intervention is required and some existing views are discarded. Thirdly, the computational cost of the automatic modification shows a slow linear growth in the number of source database. Other characteristics addressed include EA-SODIA' scalability, domain independence, autonomy of source databases, and potential of involving other data sources (e.g.XML). Finally, the descriptive comparison with other data integration approaches is presented. It shows that although other approaches may provide better performance of query processing in some circumstances, the service-oriented architecture provide better autonomy, flexibility and capability of evolution.

Acknowledgements

I would like to thank everyone who has helped me with my research. In particular, I am very grateful to my supervisor, Professor Keith Bennett, whose advice, insight, and encouragement have been invaluable throughout. I would like to thank Professor Roger Crouch whose advice and encouragement during the final stage of my research have been extremely helpful. My thanks also go to all the other people in the department who have discussed my work with me, Mr. Chong Wang in particular.

I have thoroughly enjoyed my time as a Ph.D. student and much of this is due to the people I have shared an office with over the years.

I would like to thank my parents, HanSheng Weng and JingQi Huang, and the rest of my family who have provided encouragement, love, and support throughout. They have listened to many explanations of my ideas and have always been happy to read my work.

Finally, my greatest thanks go to my wife Jun Yang. Without Her unfailing love, company, encouragement and confidence in me, the ups and downs of research would have been a harder ride. Life wouldn’t be as much fun without her and the support she has given me during this work has been immeasurable. This thesis is dedicated to her with all my love.

Statement of Copyright

Declaration

The material presented in this thesis is the sole work of the author and has not been previously submitted for a degree at this or any other university.

Contents

Chapter 1 Introduction

1.1 Context

Data integration aims to combine existing databases that are distributed, heterogeneous and autonomously managed, providing the user with a unified view of these data. Most of the time, each of the sources is independently designed for autonomous operation long before the data integration system. Traditional data integration approaches such as *federated database systems* [36] and *data warehousing* [40,80] focus on resolving the heterogeneities of the source databases and building a global view on which the user can raise queries.

However, the IBHIS project (Integration Broker for Heterogeneous Information Sources) [06] shows that the source databases may evolve constantly to reflect the business drivers. Traditional approaches fail to meet the requirements of the evolving environment, as they require tremendous modification work on the data integration system. Consequently, architectures and algorithms are required to provide solution to evolution problems while resolving the heterogeneity issues.

1.2 Area of Interest

1.2.1 Terms and Notations

In order to avoid ambiguity, the terms and concepts used in this thesis need to be precisely defined. As the primary focus is on integrating relational databases, the terms and their notations (e.g. Semantic Modelling (Entity-Relationship (ER) Model) and Relational Data Model) in this thesis follow those used by C.J Date [91], unless explicitly indicated. In relational models, a *relation instance* can be denoted as *r*, which consists of a *schema* and a *body*. The schema of r is defined as a set of attributes, which are defined as ordered pairs *<Ai, Ti> (i = 1, 2, ..., n)*, denoted as *R{A1 T1, A2 T2, ..., Ai Ti},* where *Ai* is an attributed name and *Ti* is a type name. The value n is the degree or arity of r. For simplicity, we will use *Ai* to mean the attribute

whose name is *Ai*. The body of r is a set of tuples (denoted as t), all having the same schema as r. Note that in this thesis by relation R we mean relation instance with a schema R unless explicitly indicated.

A *database schema* is the set of all relation schemas that are involved in the database.

A *local schema* is the schema of a source database in the integration system.

In semantic modelling, ER model in this thesis, an entity type is denoted by *E*, which represents a set of similar entities. A relationship, which is denoted as *RS*, is an association among entities. Both relationships and entities have properties, denoted as *Pi*. Entity type *E1* is a subtype of entity *E2* if and only if every entity of *E1* is necessarily an entity of *E2*.

In addition, distributed and autonomously managed databases are denoted as D0, D1,…,Di located at sites S0, S1,…, Si, respectively.

Metadata is the auxiliary data describing the main data - is maintained in the integrated systems to deal with the problems caused by the heterogeneity. It can contain both technical information about the sources (such as query capabilities and access methods), and also semantic information (such as the semantic connections between the relations, the domain dictionary specification).

Middleware [91] is not a precisely defined term. Generally, it describes a piece of software that connects two or more software applications, allowing them to exchange data. This thesis is concerned with *data access middleware* (also known as *mediators*).

1.2.2 Major Characteristics of Distributed Databases

The data sources considered in this thesis are mostly relational databases. As the source databases are often designed for different purposes and owned by different organizations, some of their characteristics make the design and modelling and operation of a data integration system very difficult. The major characteristics analyzed in most current researches [36] are distribution, heterogeneity and autonomy of source databases, while in this thesis database evolution is the major focus.

1.2.2.1 Distribution of Data Sources

The individual source databases in an application domain are distributed across different organizations and sites rather than situated on the same host. To illustrate this precisely, assume we have databases D1, D2 … Dn which locate at geographically different sites S1, S2 … Sn, respectively. A data integration system may need to access some of the them (e.g. D2, D3 and D5) to answer a single user query.

1.2.2.2 Autonomy of Data Sources

Usually data sources are created in advance of the integrated system and do not know that they will be a part of the integrated system. They can make decisions independently and they can not be forced to act in certain ways. In addition, their own systems are running independently of the integrated system. As a consequence of this, they can also change their data or functionality without any announcement to the outside world. A database, which has all the above features, is called a *fully-autonomous* database in this thesis.

1.2.2.3 Heterogeneity

In [36], the types of heterogeneities in the databases systems can be divided into those due to the differences in Database Management Systems (DBMSs) and those due to the differences in the semantics of data. The former includes differences in data models (e.g. relational model or object-oriented model) and in query languages (e.g.

QUEL and SQL) and in system level support (e.g. concurrency control, commit and recovery), while the latter one occurs when there is a disagreement about the meaning, interpretation, or intended use of the same of related data. More importantly, the data may be represented in different structures even if they are in the same data model (e.g. relational model), as they are designed by individual organisations in their own ways.

This research is mostly focusing on the semantic heterogeneity of the databases, although the architecture proposed in this thesis is capable of integrating data from different DBMSs hiding low-level heterogeneities (e.g. hardware platforms, operating systems, and networking protocols). As the data sources are autonomously managed, they may represent the information about the same entity or relationship type in various schemas due to the fact that the database designers may model the real-world concepts in different ways even if they all use relational models. In [92], the ER model is most relevant to the first three steps: requirements analysis, conceptual database design, and logical database design, in sequence.

1) The requirements analysis process is concerned with understanding what data is to be stored in the database; namely to find out what the users want from the database.
2) The second step is to develop the semantic model of the data to be stored in the database based on the information gathered in the requirements analysis step.
3) The logical database design is to implement the database design by converting the conceptual database design into a database schema in the relational data model.

Although the differences may emerge at any step, they do not have much impact on the heterogeneities in the final result. In order to describe those heterogeneities precisely, we assume that they mostly arise at logical database design step, namely converting the same ER model into the relational data model. We adopt the following taxonomy of heterogeneities from [86] with some modification and describe them more precisely as follows.

Naming Conflicts

Different schemas may use the same term to describe different concepts (*homonyms*) or two different terms to describe the same concept (*synonyms*). Let E (or RS) be either a set of entities (or a set of relationships) with a set of properties {P1, P2,…, Pi}. Assume that E (or RS) is mapped into relation R0 defined over the set of attributes A = {A0 T0, A1 T1,…, Ai Ti} in database D0, and to relation R1 defined over the set of attributes B = {B0 T0, B1 T1,…, Bi Ti} in database D1. D0 and D1 are said to have naming conflicts if one or both of the following conditions are true:

1) R0 and R1 have different names, denoted as $Rname(R0) \neq Rname(R1)$.
2) There exists an attribute Aj $\in$ A and an attribute Bk $\in$ B such that Aj and Bk are mapped to the same property of E (or RS) and have different names, denoted as $Aj \neq Bk$.

Semantic Conflicts

Different schemas use different levels of abstraction to model the same entity. Let *E* be an entity type which has two subtypes *E1* and *E2*. Assume that we have two databases *D0* and *D1*. *D0* and *D1* are said to have semantic conflicts if E is mapped into relations in *D0* (or *D1*) with *E1* and *E2* indicated, while E is mapped into relations in *D1* (or *D0*) without *E1* and *E2*. For example, one database might distinguish between "cars" and "trucks", whereas another database in the same integrated system might simply model "automobiles" and fail to store the car/truck distinction.

Structural Conflicts

Different schemas may represent the same information in different ways. Let *RS* be a relationship type which includes a set of participant entity types *E {E0, E1,…,Ei}(i = 0,1,…,n*). Assume that we have two databases *D0* and *D1*. *RS* and entity types in E are mapped into a set of relations *R0 {R01, R01,…, R0j}(j = 0,1,…,n)* in *D0* and a set of relations *R1 (R10, R11,…, R1k) (k = 0,1,…,n)* in *D1*. *D0* and *D1* are said to have

structural conflicts if any of the following conditions take place:

Condition (1) *RS* is mapped into a subset *R2* of *R0* in *D0* and another subset *R3* of *R1* in *D1* (namely *RS* can be mapped into a derived relation from a join operation among all the relations either of *R2* or of *R3*). Let x be the cardinality of *R2 (x>0)* and y be the cardinality of *R3 (y>0)*, then $x \neq y$.

Condition (2) There is a subset *E1* of *E* such that *E1* is mapped into a subset *R2* of *R0* in *D0* and another subset *R3* of *R1* in *D1* (namely the *E1* can be mapped into a derived relation from a join operation among all the relations either of *R2* or of *R3*). Let x be the cardinality of *R2 (x>0)* and y be the cardinality of *R3 (y>0)*, then $x \neq y$.

Condition (3) There is a property *P* of *Ei* such that $Ei \in E$, and P is mapped into a single attribute *A* of a relation in one of *D0* and *D1*, and is mapped into a composite attribute *A (A1, A2,…, An)* in another database (e.g. Name (firstname, middlename, lastname)).

Condition (4) There is a subset *E1* of E such that *E1* is mapped into a subset *R2* of *R0* in *D0* and another subset *R3* of *R1* in *D1* (namely the *E1* can be mapped into a derived relation from a join operation among either the relations in *R2* or the relations in *R3*). Let *x* be the cardinality of *R2* and *y* be the cardinality of *R3*, then $x = 0$ when $y > 0$ or $y = 0$ when $x > 0$.

Condition (5) There is a set of properties P of Ei, such that $Ei \in E$ and the cardinality of P is greater than 0, which are mapped into a set of attributes of a set of relations in some databases and are not mapped into any attributes in other databases.

Metadata Conflicts

A concept can be represented with the schema in one data source, but as regular (non-schema) data in another data source. Let *E* be an entity type which has two subtypes *E1* and *E2*. Assume that we have two databases *D0* and *D1*. *D0* and *D1* are said to have metadata conflicts if:

1) In one database *D0* (or *D1*), E is mapped into a single relation *R* and one of

the attributes *Ai* of *R* indicates whether a tuple in *R0* represents an entity of subtype *E1* or *E2*.

2) And in anther database *D1* (or *D0*), E is mapped into two relations *R1* and *R2* which represent *E1* and *E2* separately. Tuples in *R1* represent the entities of subtype *E1*, while tuples in *R2* represent the entities of subtype *E2*.

For example, one data source may distinguish between cars and trucks by maintaining two separate relations; one for cars and one for vans. Which relation a tuple appears in specifies whether the vehicle is a car or a truck. Another data source may use a single relation, but have an attribute in that relation that indicates whether or not a tuple in the relation represents a car or a van.

Domain Conflicts

Different schemas use different simple values to represent data. Let *E* be an entity type with a set of properties *P {P0, P1,..., Pn}*. Assume that we have two databases *D0* and *D1*. E is mapped into a relation *R0* in *D0* and mapped into a relation *R1* in *D1*. *D0* and *D1* are said to have domain conflicts if there exists attributes *<A0 T0>* in *D0* and *<A1 T1>* in *D1* which represent the same property in *P*, and $T0 \neq T1$. For example, one relation might store car price as an integer number, while another might store a textual-rendition of the car's price in a text string.

1.2.2.4 Related Work

Most traditional approaches [63] to database integration combine data residing at different sources databases that have the above characteristics, and provide the user with a unified view of these data. Such a unified view is represented by the global schema, and provides a reconciled view of all data, which can be queried by the user. Generally, the current approaches can be categorized as follows [80]:

- *Virtual View Approach*: In this approach data is accessed from the sources on-demand when a user submits a query to the integrated system. This is also called a *lazy* approach.

- *Materialized View (or Data Warehousing) approach*: Some filtered information from data sources is pre-stored (materialized) in a repository and can be queried later by users. This method is also called an *eager* approach.

Table 1-1 illustrates four major approaches that are either a virtual view or materialized view approach.

Table 1-1 Major approaches to data integration

Data Integration Approach	**Categorization**	**Explanation**
Federated Database Systems (FDBS)	Virtual View Approach	These comprise an integrated collection of distributed databases, in which the component administrators maintain control over their local system [36].
Mediated System	Virtual View Approach	These integrate heterogeneous data sources by providing a global schema (virtual view) of all this data. Users ask queries based on the global schema without the need to know about the data source location, schemas or access methods.
Open Grid Service Architecture-Distributed Query Processing (OGSA-DQP)	Virtual View Approach	This is an approach to service-based distributed query processing on the Grid [45]. It exposes the schemas of source databases exposed by grid data services, and allow users to build queries on those schemas as if they are in a single database. However, the users have to figure out all the heterogeneity problems.
Data Warehousing	Materialized View Approach	A warehouse is a centralized repository of information extracted from multiple data sources.

As all the approaches aim to provide a global schema (or view) for the users to raise queries, one of the main tasks in a data integration approach is to establish the mapping between the source databases and the global schema [63]. In this task, all the heterogeneities must be resolved. In this research, this task is called *Schema Reconciliation*. Three major approaches to integrating source databases involve schema integration [36], Local as Views (LAV) and Global as Views (GAV) [63]. A federated database system and a data warehousing often use the schema integration,

while a mediated system may use either LAV or GAV. Each approach to data integration is further described in Chapter 2 with relevant approaches to schema reconciliation.

In order to complete the schema reconciliation task, huge amounts of hard-coded programs have to be produced. For example, in a federated system or data warehouse or OGSA-DQP system, hard-coded programs are produced to hide the technical details of connection to the source databases. Also, existing queries must explicitly specify the source databases from which they intend to obtain results. Moreover, the queries are also built in terms of the schemas of the source databases and require hard-coded programs to tackle the heterogeneity among them. As for a mediated system based approach, although the users are built in terms of global schemas, it requires hard-coded programs for each source database to eliminate the heterogeneities. These require that the participating source databases have to be known in advance. These approaches are referred as *static binding* in the IBHIS project [06]. Consequently, it is very hard to maintain those systems when database evolutions occur constantly.

1.2.2.5 Database Evolution Problems

The IBHIS identifies two classifications of evolutions as following [06]:

- Internal changes (within the organisation) refer to the changes that occur within organisations. Those changes are reflected in data resources, and include: data, data structure, constraints, permissions or rules, data model and semantics.
- External changes (outside the organisation) refer to changes made by third party service providers.

We focus on the evolutions on the source databases that involve both internal changes and external changes, and further classify them into three types: schema evolution, system evolution and organization evolution. Table 2 shows these evolutions with

explanations and the difficulty caused by them in an integration system.

Table 1-2 Database Evolutions

Evolution	Explanation	Successive Difficulty
Schema Evolution	This refers to the evolution in local schemas of the source databases. This involves: ➢ Adding or removing attributes into or from a relation schema. ➢ Changing of the name or domain of an attribute. ➢ Decomposing a single attribute into more attributes. ➢ Adding or removing relation schemas into or from a local schema. ➢ Changing of the name of a relation schema. ➢ Decomposing a single relation schema into more relation schemas. ➢ New source databases coming in or existing source databases become available.	In the traditional approaches introduced above, all the hard-coded programs that specify the evolved schemas have to be identified and modified. These programs involve the programs for dealing with schema reconciliation, existing user queries and the programs to update the information in warehouses (for data warehousing systems). The system cannot work properly until all the modifications are complete.
System Evolution	This refers to the evolution in the descriptive information of the source databases. For example, the name and URL of a source database, which is used for programs to access the source database, may change.	This may result in modifications on the hard-coded programs dealing with the connection to the evolved source databases.
Organization Evolution	This refers to the evolution in the organizational hierarchical structure of the source databases. For example, the definition of the region changes.	The existing queries that analyse or summarize data based on the old organisation structure cannot reflect the latest situation.

Organization evolution is a slightly complicated evolution which may cause two problems. Firstly, some existing queries, which intend to provide the latest analysis

and summarization of the data based on the organization structure, may no longer provide the proper results. Therefore, the queries must be modified. Secondly, some queries may need to compare and analyse the data over several years based on the organization structure. However, during these years the organization structure may have been changed several times. Consequently, both the old organization structures and the new one have to be retained. Also, changes to the queries to involve the new organization structure require huge work. The detailed description and the formal representations of those database evolutions will be presented in Chapter 5, 6, and 7.

1.2.2.6 Service-based Architecture and Dynamic Binding

The concept "Software as a Service" (SaaS) was proposed by the Pennine Research Group, in which *services* [07] are composed out of smaller ones (and so on recursively), procured and (possibly) paid for on demand. This solution provides a possible approach for organisations to share resources in a constantly changing environment. The central technical issue for this solution is very *late binding*, at the point of the execution of a system.

Based on the idea of SaaS and late (dynamic) binding [01,02], the IBHIS project proposed a service-oriented data integration architecture (SODIA) to provide a dynamically unified view of data on demand from various autonomous, heterogeneous and distributed data sources [06]. It indicates that source databases are published as services that are dynamically determined and bound at the time of execution.

Our research aims to provide an *Evolution Adaptive Service-Oriented Data Integration Architecture (EA-SODIA)* to dynamically integrate existing, heterogeneous, autonomous databases in a constantly evolving environment. Each source database is published as a *data service* into a *data integrator service* that is responsible for dynamically identifying and combining relevant source databases. The *data integrator service* is published into a *registry service* for the user to find. LAV is

used in this architecture to establish mapping between local schemas and the global schema, as it provides a possible solution of dynamic binding of source databases.

1.3 Discussion of Issues

1.3.1 Research Issues

The integrated system based on our architecture is referred to as a virtual view approach and needs to:

- provide an integrated view of data from autonomous heterogeneous data sources.
- allow data sources to evolve independently.

Therefore, the major issues in this research are described as following:

- **Schema Reconciliation**: is to provide a global schema for users to raise queries on it, and to integrate each source database to the global schema using LAV. It is also to eliminate the heterogeneities between the local schema and the global one, using *relational algebra operations* (it is referred to as *source description*). The mappings and source description are stored in a meta-database, ensuring that any evolutions from the source databases will be managed within the meta-database.
- **Query Process**: is to dynamically determine the source databases to be accessed when a user query is posed on the global schema. It is also to decompose the user query into queries that are in terms of each local schema based on the mapping in the meta-database.
- **Schema Evolution Detection**: is to identify the impact of a schema evolution on the mappings in the meta-database and automatically modify them, in order to further reduce the manual maintenance.

This approach is able to cover most heterogeneities defined previously, except the semantic conflict. It is because LAV and the relational language are unable to further

classify an entity if there is no extra attribute to indicate. For example, a query using relational language cannot find an automobile that is a van if there is not an attribute to represent this property. Also, the nature of dynamic binding is to find the latest source databases and produce latest results by combining them. Therefore, the second organization evolution problem in which both the old organization structure and the new one have to be retained cannot be resolved by the dynamic binding directly. However, future research may involve this problem based on our architecture using the method such as storing different versions or domain ontology in the meta-database. The complete discussion of this architecture is presented in Chapter 8 (Evaluation).

1.3.2 Problem Boundaries

The architecture and approaches in this thesis have been presented with the assumption of certain problem boundaries.

1) In general, the data sources involved in the architecture are currently assumed to be relational databases due to their dominating position in the industry, although they can be managed by various DBMSs. Each database provider has responsibility for publishing its database as a service, and for building the mapping between the local schema and the global schema.
2) It is also assumed that every relation schema in the system has to be in at least first normal form by which we mean that the attributes of relations need to be atomic. For the time being, although the schema of source databases can be defined using different terms, ontology problems on the contents of the relations and security problems are beyond the scope of this thesis. They require further research based on our approach because they are also inevitable.
3) In addition, redundancy is not a problem considered in our research. As our objective is to provide an architecture and approach to building an evolvable data integration system, query optimization is not a key problem and is simply described as a small part of query decomposition.
4) Finally, the changes mentioned in this thesis are those occurring in local schemas and their organizational hierarchy. We assume that the global schema on which

user queries are raised basically remain unchanged. As this approach is currently referred as a virtual view approach, the solution to maintaining views in a materialized approach when database evolutions occur is left for further research.

In summary, the thesis aims to provide a virtual view approach to solve some database evolution problems while combining heterogeneous, autonomous and distributed databases. The heterogeneities considered in this research only exist among the schemas of source databases, rather than among the contents. The evolutions that are in question are primarily the evolutions of source databases (their schema and connection information) and of the organization structure. Moreover, our approach intends to make the system work properly on the current source databases (after the evolution), and cannot reflect the data before the evolution. However, our architecture has the potential to incorporate these problems if further research is carried out.

1.4 Research Aims and Criteria for Success

Many aspects of both the problem and solution are covered by the aims of this research and the criteria for success. A case study [93,94,95] is required to examine whether the approach fulfills the aims of this research. The case study method is chosen for the following reasons:

- The evolutions in the system cannot be fully controlled and be predicted. Some evolutions may occur constantly, while others may never occur in a real project. Therefore, a formal experiment [93] is not possible to conduct as it requires full control on the evaluation variables. A case study is more suitable to test over some typical evolutions.
- Formal experiment and survey requires multiple projects, while a case study can investigate a single large-scale project. Although, formal experiments are easier to be generalized to every possible situation, this research is focusing on a typical situation (changing environment).

A case study is further described and discussed in chapter 8.

The heterogeneity and database evolution problems introduced previously have to be solved and some experimental implementations are constructed to demonstrate the feasibility of the architecture. The criteria for success are formally defined as follows.

1) The heterogeneities defined in Chapter 1 can be eliminated using the RSMV approach and the query processor.
2) The RSMV approach and meta-database can reduce the cost of modification work caused by schema evolutions, and the query processor can reduce the number of the queries which require modification when any organizational evolution occurs.
3) If any schema evolution occurs in one source database, the views of other source databases do not require modification so that the system can still work properly.
4) The SED can reduce the cost of modification work caused by schema evolutions.
5) SOA and web services can help reduce the cost caused by the database evolutions and system evolutions because they provide high reusability, autonomy and discoverability.

As a consequence of all above criteria, the cost of maintenance resulting from database evolution can be largely reduced. In Chapter 7 and 8, the success of the architecture and the approaches are discussed with reference to the above criteria through a case study. The case study methodology itself will be introduced and further discussed as well. We will discuss the performance of the approach in different situations. However, the security and the data transportation efficiency of the system based on our approach are not considered as criteria of success, because they are not the focus of this research.

1.5 Evaluation Criteria

In addition to reaching the criteria of success listed in the previous section, the architecture should be capable of handling real-world applications. Again, the architecture is extensively evaluated in chapter 8; where the extent to which it can help in a real application (and its strengths and weakness) is explored. The evaluation is based on following criteria:

- Cost of modifying views
- Scalability
- Domain independence
- Language independence
- DBMS independence
- Expandability

These criteria cover various aspects and can help in figuring out where the architecture can be applied.

1.6 Contribution

This research proposes a service-oriented architecture, based on the concept of *SaaS* [07] and *late binding* [07], for dynamic integration of existing, distributed, heterogeneous and autonomous databases in an evolving environment. The primary contributions are the schema reconciliation algorithm using both LAV and extended relational algebra operations, and a meta-database to store all the data resulted from the schema reconciliation algorithm. Also, the data service is different from that of other service-based architectures. A data service explores only the reconciled schema of the underlying database, rather than the local schema. It receives the whole query as a parameter and then translates it into the query in terms of the local schema, based on the data in the meta-database. Finally, the schema evolution detection algorithm is provided to conduct automatic modification of the data in the meta-database when evolutions occur. The following research issues within data dynamic data integration

systems are addressed:

- **Schema Reconciliation**: A combined and extended approach of **LAV** and relational algebra is used to achieve this goal. Also, the meta-database ensures that any database evolutions from the source databases will be managed within the meta-database.
- **Query Process**: relevant source databases (services) are determined and combined, making sure that no hard-coded queries specific to individual databases exist until run-time.
- **Evolution Detection**: The ruled-based algorithm is used to reason about what views are affected by the changes and how they can be modified.

The approach provided in this research is suitable for information integration systems which are in a changing environment. They produce constraints and rules for databases to be integrated into the system and their schemas accommodated into a global schema. They also define rules for evolution detection programs to figure out what views are affected and to modify the views automatically. The approach is compared to other data integration methods throughout this thesis and extensive evaluation of it is presented.

1.7 Thesis Structure

This thesis is divided into ten chapters

Chapter 1 introduces the motivation and context for the research, discusses the problem to be solved, and sets out the research aims and criteria for success.

Chapter 2 introduces the main approaches to data integration with the support of database evolution problems. The Service-Oriented concept is introduced with the main techniques.

Chapter 3 presents an overview of the service-oriented architecture in this research with a brief introduction to its components. Each process is generally introduced.

Chapter 4 presents the approach to building global schema and establishing mapping between the local schema and the global schema. The approach to representing mappings in a meta-database is also depicted. Rules and constraints based on set theory and logics are defined.

Chapter 5 describes evolution detection methods based on rules. Various types of database evolutions are described and represented in the meta-database. The processes of identifying the affected mappings and automatically modifying them are presented.

Chapter 6 introduces the algorithm of decomposing queries over the global schema into subqueries that refer to source databases and translating subqueries into queries that are directly over the schema of the source databases.

Chapter 7 presents the design of the services in the architecture. A case study including an experimental implementation using web services is also presented.

Chapter 8 discusses the results of case study evaluation of the architecture and approaches proposed in this research with reference to the criteria of success and evaluation presented in sections 1.4 and 1.5. The methodology, called case study, is used to evaluate the approach.

Chapter 9 concludes the research by giving a general discussion and summary of the work accomplished. The success of the research is considered in terms of the criteria presented in section 1.4. The ideas for further work are also suggested.

1.8 Summary

This chapter has given an introduction to the work presented in this thesis. Some basic terms that are used throughout the thesis are explained. The motivation and context of the research have been explained with reference to other research achievements in the field. The two main characteristics of distributed databases, making data integration difficult, have been introduced: heterogeneity and data evolution. The major research issues have also been identified: schema reconciliation, query processing, building meta-database, and evolution detection. Evaluation criteria have been presented and the structure of the thesis is explained.

Chapter 2 Background

2.1 Introduction

Chapter 1 introduced the context and motivation of the work in this thesis. Various characteristics of heterogeneous and distributed databases were presented and the research problem was defined with several issues. Criteria for success and for evaluating both the approach and the research were presented.

This chapter further examines some current approaches to data integration and discusses the basic process of them. Major approaches to mapping the data sources to global schema are described and compared. Their support for database evolution problems is then discussed. An introduction to the concepts of Software as a Service (SaaS) and *late binding* [01,02] is then presented. Finally, service-oriented architecture (SOA) and Web services are presented with relevant techniques and standards.

2.2 Current Approaches to Data Integration

As mentioned in the previous section, two common approaches to data integration are the Virtual View Approach and the Materialized View Approach [80]. Each of them includes one or more architectures. The Virtual View Approach is also called a lazy approach to data integration. This approach is based on the following very general two-step process [80]:

1. In this case the data is accessed from the sources on-demand when a user submits a query to the information system. That is why it is also referred to as a lazy approach. Three architectures involved in a virtual view approach are described later in this section: *federated database systems*, *mediated systems* and *distributed query processing (DQP).*
2. The Materialized View Approach is also referred to as data warehousing or an eager approach to data integration. Information from each source that may be of

interest is extracted in advance, translated and filtered as appropriate, merged with relevant information from other sources, and stored in a (logically) centralized repository. When a query is posed, the query is evaluated directly at the repository, without accessing the original information sources.

2.2.1 Federated Database Systems

A *Federated Database System (FDBS)* consists of semi-autonomous components (database systems) that participate in a federation to partially share data with each other [36]. The databases cannot be called fully-autonomous because each database is modified by adding an interface that allows communication with all other databases in the federation. In a federated architecture, a *federated DBMS* serves as a middleware providing transparent access to a number of heterogeneous, distributed data sources. Each source in the federation can also operate independently from the others and the federation. FDBSs [36] can be categorized as loosely coupled or tightly coupled based on who manages the federation and how the components are integrated. A tightly coupled federation has one or unified schemas which are built by federation DBAs, while a loosely coupled federation has no unified schema and it is the end user's responsibility to create and maintain the federation.

A tightly coupled federation is static and usually difficult to evolve, because creating a federated schema is like database schema integration which does not allow adding or removing components easily. The key limitation of this approach is that applications have to explicitly specify the data sources in a federated query [47]. This means that the applications must be changed when new data sources are added. Each data source must also be explicitly registered to the federated DBMS. Also, it is very costly if organizations change the data sources.

Loosely coupled FDBSs are dynamic, as their federated schema may be managed on the fly (created, changed, dropped easily) by a user. Requested data comes from the exporter of this data itself and each component can decide how it will view all the

accessible data in the federation. However, humans must still resolve all semantic heterogeneities. The most naive way to achieve interoperability in the loosely coupled federation is to map each source's schema to all others' schemas, so-called pair-wise mapping. However, it requires $n*(n-1)$ schema translation and becomes too tedious when the number of components becomes very large.

2.2.2 Mediated systems

A mediated system integrates heterogeneous data sources, which can be databases, legacy systems, filed systems, web sources, etc, by providing virtual views of all this data. End users who raise queries on the mediated system do not have to know anything about data source location, schemas or access methods, because such a system presents one global schema to the users so that they ask queries in terms of it.

Although a mediation architecture is, to some extent, likely to be similar to a tightly federated system, it is different in the following ways [36]:

- A mediated architecture may have non-database components.
- The query capabilities of sources in a mediator-based system can be restricted and the sources do not have to support SQL-querying at all.
- Access to the sources in a mediator-based system is usually read-only as opposed to read-write access in a FDBS (due to the fact that the sources in the mediator-based system are more autonomous).
- Sources in a mediator-based approach have complete autonomy which means it is easy to add or remove new data sources.

The main components of a mediated system are the mediator and one wrapper per data source. The mediator receives user queries on the global schema, and decomposes them into subqueries to local individual sources based on source descriptions, and then sends them to the wrappers of individual sources. A wrapper executes the subquery, hiding technical and data model detail of the data source from the mediator. Usually, there are some specific programs dealing with the

transformation from a global schema to a local schema. We will talk through this problem in more detail later in the thesis.

Two basic approaches have been used to specify the mapping between the sources and the global schema [86].

- **Global-as-view (GAV)**: It requires that the global schema is expressed in terms of the data sources. In turn, the global schema are defined as views over the global schema
- **Local-as-view (LAV)**: It requires that the global schema is specified independently from the sources. In turn, the sources are defined as views over the global schema.

Two typical example systems implementing mediator-based architecture are: TSIMMIS (The Stanford-IBM Manager of Multiple Information Sources) [96] that is based on GAV, and Information Manifold [97] that is based on LAV. IM makes it quite convenient to add new sources. One must write a wrapper for the sources and define its views and constraints in terms of the global schema. However, no change to the query processing algorithm is needed. The new views will be used whenever they are appropriate for the query. In contrast, new Tsimmis sources not only must be wrapped, but the mediators that use them have to be redefined and their definitions recompiled. The administrator of the system must figure out whether and how to use the new sources. A complete comparison of the approaches is reported in [98].

2.2.3 Data Warehousing

Data warehousing [37] (materialized views) offers higher availability and better query performance as all information can be retrieved from a single location, and thus is a suitable choice when high-performance query processing and data analysis is critical. In a data warehousing approach, data from various sources is integrated by providing a unified view (or unified schema) of this data, like in a virtual view approach, but here this filtered data is actually stored in a single repository (called a *data warehouse*).

The wrapper component is responsible for translating information from the native format of the source into the format and data model used by the data warehousing system, while the monitor component is responsible for automatically detecting changes of interest in the source data and reporting them to the integrator.

Another component in the architecture, the integrator, is responsible for installing the information in the warehouse, which may include filtering the information, summarizing it, or merging it with information from other sources. In order to properly integrate new information into the warehouse, it may be necessary for the integrator to obtain further information from the same or different information sources. The warehouse may be also implemented as a distributed database system. In the data warehousing system, the sources that are integrated always retain their autonomy.

2.2.4 Open Grid Services Architecture-Distributed Query Processor (OGSA-DQP)

Distributed query processing (DQP) has been widely used in data intensive applications where data of relevance to users is stored at multiple locations [49, 44]. DQP is found in several contexts such as distributed database systems, federated database systems and query-based middlewares.

OGSA-DQP [45] is an approach to service-based distributed query processing on the Grid. It is an example of a high level data integration framework. In the framework, each source database is exposed as a grid data service (GDS) which accepts and executes queries hiding technical details such as the type of DBMSs from the external users or applications. OGSA-DQP exposes the schemas of source databases exposed by GDSs, and allow users to build queries on those schemas exposed as if in a single database.

Although OGSA-DQP provides an approach to integrating existing distributed and

autonomous databases on the Grid, it does not directly address some of the problems mentioned previously. For example, there are no algorithms for schema integration or unified schema that are requested for dealing with heterogeneity. Therefore, users need to raise queries on the local schema directly so that they have to understand all the individual schemas requested very well and have to explicitly specify the data sources in a federated query. It means that the system can not provide a unified view for users and needs expertise to build queries. In addition, the system based OGSA-DQP may have difficulties when adding or removing data sources, because the queries are written in hard-code based on their local conceptual schemas. This requires that some middlewares are placed over DQP to more complete functions to above problems. The Service-Oriented Architecture and relevant techniques such as web services and grid services are discussed in further detail in the next chapter.

2.2.5 Comparison of the architectures

In general, the virtual view approach to data integration is preferable for information that changes rapidly, for clients with unpredictable needs, and for queries that operate over vast amounts of data from very large numbers of information sources. However, the virtual view approach may incur inefficiency and delay in query processing, especially when queries are raised multiple times, when information sources are slow, expensive to access, or periodically unavailable, and when significant processing is required for the translation, filtering, and merging steps.

If, however, sources are permanent, and do not get upgraded too often and the designers of the integrated system know what kind of queries are going to be asked most often, answers to these queries can be materialized. A data warehousing approach might be chosen to improve the performance if some sources are physically located far away from the mediator leading to delay in response time. However, a Data Warehousing system does not provide very up to date information and is not appropriate in above mentioned circumstances where the virtual approach is preferable.

Among the architectures based on the virtual view approach (primarily federation and mediation), the mediated approach is chosen more often. As for the federation, the systems with this architecture are not very common nowadays due to the large number of interfaces that need to be written for each source to communicate with all the others.

2.2.6 Support for Evolution Problems

Generally, the currently approaches introduced above fail to meet the requirements of constant database evolutions.

- The tightly coupled federation is mostly based on manual and static schema integration at the design time. The federated DBMS has to contain all the technical details of the source databases and their wrappers. Applications on the federated DBMS have to explicitly specify the data sources in a federated query. It is not appropriate in a dynamic environment where the source databases are constantly evolving, because huge amount of changes on the system are required.
- The loosely coupled federation requires a huge amount of work on translating queries-based schema matching, which is expected to be done by users. Consequently, when a source database evolves, all the components accessing the evolved source database require modification. It is also not realistic that all the relevant components can be notified when an evolution occurs.
- As the Data Warehouse is centralized and static, it cannot meet the requirements of the changing data resource such as data structure changes and emerging new data resources.
- The mediation based on GAV is hard to evolve, because new sources not only must be wrapped, but the mediators that use them have to be redefined and their definitions recompiled. The primary reason for the redefinition of the mediators is that the global views of the mediators specify explicitly the relationships between source databases. Consequently, all the relevant global views must be changed when a source database evolves.

- The mediation based on LAV, to the contrary, provides a unified virtual view to users and allows adding and removing more easily. However, evolutions in the underlying sources may cause changes to the wrappers and the global view. Adding a data source also requires that a new wrapper for the new data be defined.
- DQP is most likely to be a query process tool on which a data integration system can be built. The integrated system based on DQP requires much modification following the database evolutions too.

To conclude, none of above current approaches provides a solution to schema evolution problems which are the focus of this thesis. All the above approaches are further compared with the architecture proposed by this thesis in Chapter 3

2.3 Service-Oriented Concept and Techniques

Evolutions are inevitable, expensive and very hard to undertake. Both Bennett [06,02] and Ghezzi [99] suggest that traditional static bound (early bound) supply-side systems cannot meet the needs of continually changing environments. Therefore, the concept "Software as a Service" (SaaS) was proposed by the Pennine Research Group, in which *services* [06] are composed out of smaller ones (and so on recursively), procured and (possibly) paid for on demand. This solution provides a possible approach for organisations to share resources in a constantly changing environment. The central technical issue for this solution is very *late binding*, at the point of the execution of a system.

Based on the idea of SaaS and late (dynamic) binding [01,02], the IBHIS project proposed a service-oriented data integration architecture (SODIA) to provide a dynamically unified view of data on demand from various autonomous, heterogeneous and distributed data sources [06]. It indicates that data sources are published as services that are dynamically determined and bound at the time of

execution. Aligned with Web services, Service-Oriented Architecture (SOA) provides some technical support for the above concept and architecture.

2.3.1 Service-Oriented Architecture (SOA)

A SOA (Service-Oriented Architecture) is a component model that inter-relates the different functional units of an application, called *services*, through well-defined interfaces and contracts between these services [19]. The interface is defined in a neutral manner that should be independent of the hardware platform, the operating system, and the programming language the service is implemented in. It is typically characterized by the following properties [19]:

- Logical view: The service is an abstracted, *logical* view of actual programs, databases, business processes, etc., defined in terms of what it does, typically carrying out a business-level operation.
- Message orientation: The service is formally defined in terms of the messages exchanged between provider agents and requester agents, and not the properties of the agents themselves. The internal structure of an agent, including features such as its implementation language, process structure and even database structure, are deliberately abstracted away in the SOA: using the SOA discipline one does not and should not need to know how an agent implementing a service is constructed. A key benefit of this concerns so-called legacy systems. By avoiding any knowledge of the internal structure of an agent, one can incorporate any software component or application that can be "wrapped" in message handling code that allows it to adhere to the formal service definition.
- Description orientation: A service is described by machine-processable meta data. The description supports the public nature of the SOA: only those details that are exposed to the public and important for the use of the service should be included in the description. The semantics of a service should be documented, either directly or indirectly, by its description.

- Granularity: Services tend to use a small number of operations with relatively large and complex messages.
- Network orientation: Services tend to be oriented toward use over a network, though this is not an absolute requirement.
- Platform neutral: Messages are sent in a platform-neutral, standardized format delivered through the interfaces. XML [14] is the most obvious format that meets this constraint.

Although SOA is a conceptual model independent of technologies, it is often coupled with *Web services* technology due to the fact that no one technology advancement has been so suitable and successful in manifesting SOA than Web services [90]. Web services specifications define the details needed to implement services and interact with them.

2.3.2 Web Services

The Web Services architecture is based upon the interactions between three roles: service provider, service registry and service requestor. The interactions involve publish, find and bind operations. Typically, a service provider hosts an implementation of the web service, and defines a *service description* for the web service and publishes it to a service requestor or *service registry*. The service requestor can then discover the service description locally or from the service registry and uses the service description to bind with the service provider and invoke the Web service implementation. The information exchange between services is based on *messaging* [90]. Figure 2-1 [13] shows the architecture of web services.

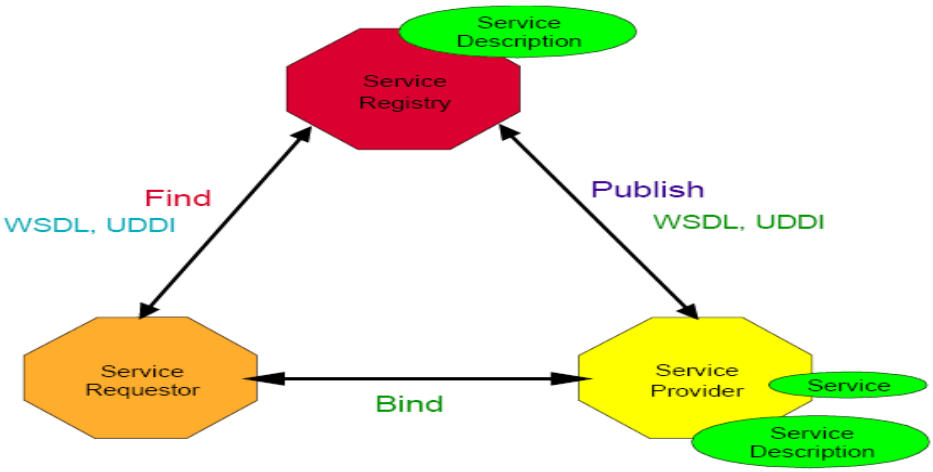

Figure 2-1. [13] Web Services roles, operations and artifacts

The roles and operations in the above architecture are listed in Table 2-1 and Table 2-2 respectively.

Table 2-1 Roles in a Web Services Architecture [13]

Role	Description
Service provider	From a business perspective, this is the owner of the service. From an architectural perspective, this is the platform that hosts access to the service.
Service requestor	From a business perspective, this is the business that requires certain functions to be satisfied. From an architectural perspective, this is the application that is looking for and invoking or initiating an interaction with a service.
Service registry	This is a searchable registry of service descriptions where service providers publish their service descriptions.

Table 2-2 Operations in a Web Service Architecture [13]

Operation	Description
Publish	To be accessible, a service description needs to be published so that the service requestor can find it.
Find or Discover (discussed later)	In the find operation, the service requestor retrieves a service description directly or queries the service registry for

	the type of service required
Bind	Eventually, a service needs to be invoked. In the bind operation the service requestor invokes or initiates an interaction with the service at runtime using the binding details in the service description to locate, contact and invoke the service.

Three basic web services standards fulfilled the model shown in Figure 2-1 as follows:

- ***Simple Object Access Protocol (SOAP)***: This is a lightweight protocol intended for exchanging structured information in a decentralized, distributed environment. SOAP uses XML technologies to define an extensible messaging framework, which provides a message construct that can be exchanged over a variety of underlying protocols. The framework has been designed to be independent of any particular programming model and other implementation specific semantics. [20]
- ***Web Services Description Language (WSDL)***: This is the de facto standard for XML-based service description. WSDL is an XML-based language for describing Web services and how to access them [13]. This is the minimum standard service description necessary to support interoperable Web Services. WSDL defines the interface and mechanics of service interaction.
- ***Universal Description, Discovery and Integration (UDDI)***: This is the central technique in Web Service architecture, which defines a standard method for publishing and discovering the network-based software components of a service-oriented architecture (SOA) [32].

The service interface definition together with the service implementation definition composes a complete WSDL definition of the service. They contain sufficient information to describe to the service requestor how to invoke and interact with the Web service. However, WSDL does not describe some high level and semantic information of services such as context of services (e.g. what business is hosting this

service?), metadata of services (e.g. what business is it? what products or services is it providing? and what is the key words to find this service?) and Quality of Services. These higher level aspects of services, which are especially important in data-intensive services, require additional service description documents, which complement the WSDL documents.

UDDI lacks support to metadata of the service such as the detailed description of the service (e.g. what products and services does the service provider produce?). The metadata of service is especially required in data-intensive service to describe the heterogeneous databases autonomously managed and provided by the owner, as not only does the application need to discover the data service, but composes the query on databases at run-time according to the metadata. Thus, we propose a meta-database that complements the UDDI and WSDL with the meaningful description of the database structure. The meta-database will be described formally in Chapter 4.

2.4 Summary

This chapter discussed some current approaches to integration of existing autonomous, distributed, and heterogeneous databases. It concluded that none of the current approaches are appropriate in a dynamic environment where the source databases are constantly evolving. The concepts of SaaS and late binding provide a possible approach for organisations to share resources in a constantly changing environment. The IBHIS project also suggested a Service-Oriented Data Integration Architecture in which data sources are published as services that will be dynamically bound on demand.

Chapter 3 Evolution Adaptive Service-Oriented Data Integration Architecture

3.1 Introduction

Chapter 2 introduced, in detail, some current approaches to integrating databases, and discussed service-oriented architecture and relevant techniques. This chapter outlines the service-oriented data integration architecture which is easier to maintain when any changes of databases occur. It is termed EA-SODIA. The data integration approach used in this architecture is similar to mediation.

The main processes of integrating database schemas and of maintaining the system in response to changes of databases are briefly introduced. It is also described how service-oriented architecture can help in these processes. The characteristics of the architecture are then explained in comparison to other approaches to data integration.

3.2 Overview of Evolution Adaptive Service-Oriented Data Integration Architecture

The approach proposed in this thesis is a service-oriented architecture in which both the data integrator component and source databases are deployed and published as services. As mentioned in chapter 2, this architecture is based on the concept of Software as a Service (*SaaS*) [01,02] and *late binding* [06] aligned with service-oriented architecture and web services technologies. The terms used to describe the architecture follow [98].

The architecture is designed with two primary aspects:

- The ability to integrate distributed databases, dealing with the heterogeneities among them.
- The ability to evolve easily without modification of hard-code programs, in response to the evolution of the underlying databases.

The heterogeneities and the database evolution mentioned above are those defined in chapter 1. Although another aspect, autonomy, is not the focus of this research, it can

be addressed within this architecture and will be discussed in this chapter.

Figure 3-1 shows the general architecture of EA-SODIA with its basic services. Three types of basic service are involved as follows.

- Each source database, in the architecture, is exposed as a *Data Service (DS)* by its provider which receives queries and returns results. A DS exposes its reconciled schema (*exporting views*) and receives a standardized query which is on the reconciled schema, and then converts it into queries which can be executed upon the schema of its underlying database. The conversion of queries is based on the specification of the mapping from the global schema to the local schema. This specification, called meta-data, is stored and maintained in the meta-database (MDB) at data source site by each data provider.
- Over those DSs is *Data Integrator Service (DIS)* which:
  - receives a query from a user, and dynamically finds the source databases that can provide data for this query.
  - decomposes the query into subqueries referencing to each source database, and then delivers the subqueries to corresponding DSs.

  Again, the query decomposition is based on the metadata stored in the meta-database (MDB) of DIS. The results produced by DSs are sent back to the DIS which subsequently composes those results into a final one and sends it to the user. Above the DIS are various client applications enabling end users to send queries to DIS. The queries are those supported by the DIS. Client applications are not parts of the architecture and will not be further described in this thesis.
- *Registry Service*: Both DIS and DSs are published into a *registry service* for other software components or services to discover which services to access and how to access. A registry service is based on UDDI that contains only the information such as the location and the methods of a service.

Note that although there is only one DIS shown on the graph, more DISs can be added performing the same function in order to enhance the performance of the architecture. They all need to be registered in the registry. This is discussed further in Chapter 7.

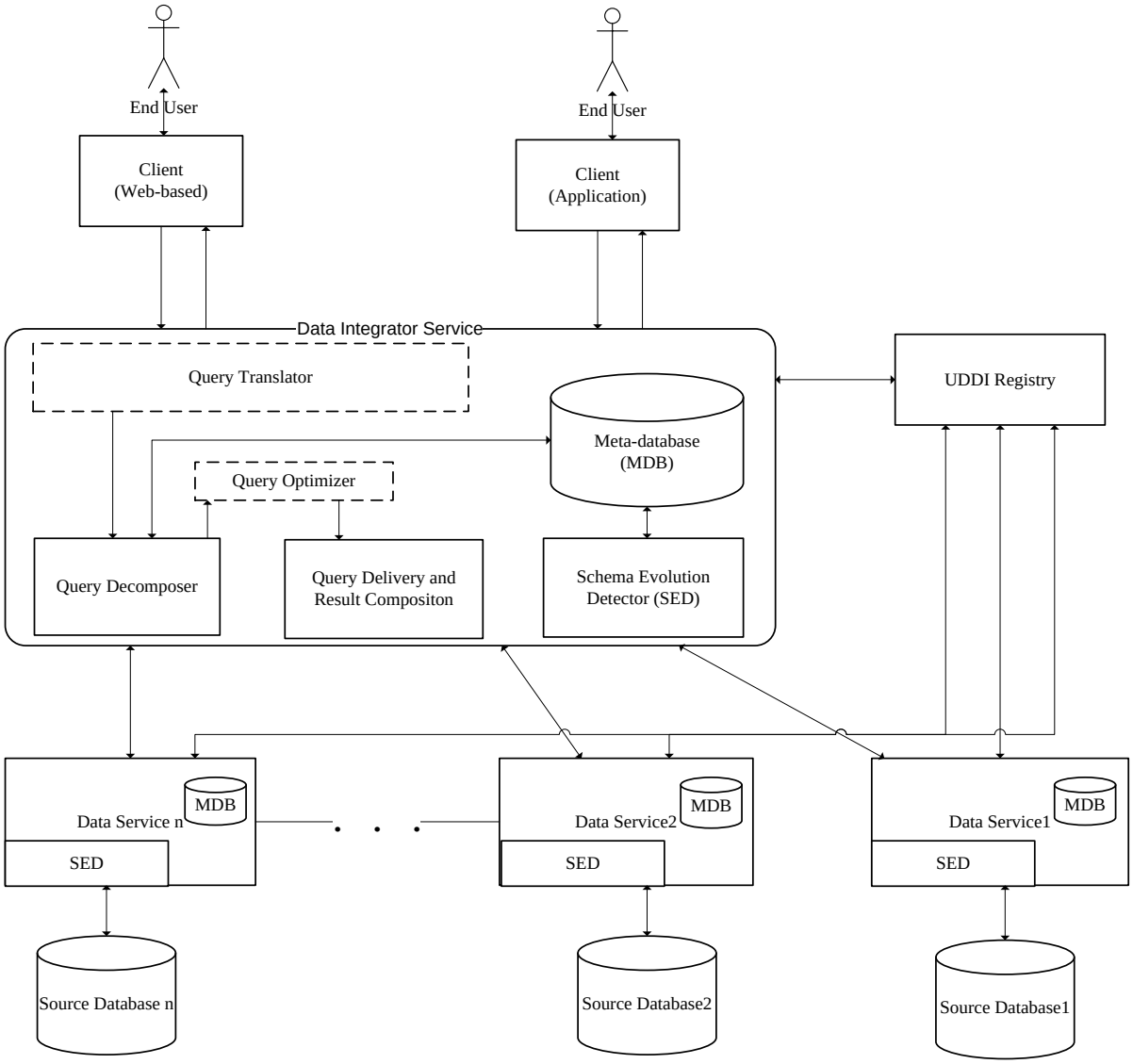

Figure 3-1 The general architecture of EA-SODIA

This architecture, to some extent, can be considered as a mediated system based on a service-oriented architecture where mediators are performing the same functions as those of DIS and wrappers are working in a similar way to DSs. However, it is more flexible than a traditional mediation approach due to the reusability and the accessibility provided by SOA. More importantly, a system based on EA-SODIA is expected to be relatively more maintainable as the creation of the schema integration in EA-SODIA needs no hard-coded programs. Therefore, no programs need to be modified when the schema of source databases change. In addition, there is no need to provide a hard-coded wrapper for a newly incoming source database. These will be evaluated by the case study in Chapter 7 and 8.

3.3 Processes of Data Integration and Evolution

The following processes are involved in this research:

- Schema reconciliation: reconciles the schemas and the representation of heterogeneous source databases, establishing mapping between the local schemas and the global schema in the *meta-database*. This is accomplished by two stages at DIS and DS sites. An algorithm for schema reconciliation, named Relational Schema Mapping by Views (RSMV), is provided.
- Query Process: decomposes the queries built on the global schema into subqueries that refer to each individual source database. These subqueries, which are still in the global format, are then delivered to the relevant DS where they are converted into queries which can be executed on the local schema.
- Schema Evolution Detection: records the data evolution mentioned in chapter1 conducted by individual database providers. It provides an automatic check on which data (mappings) have been affected and provides a semi-automatic tool to help service providers and global schema administrator to modify the definition of the views in response to the schema evolutions.

In general, two principles which must be followed when building a data integration system based on EV-SODIA are that:

- No hard-coded programs are required for establishing mappings between the local schemas and the global schema.
- Queries from end users are all built upon global schema, and therefore no hard-coded queries refer directly to the local schemas.

3.3.1 Schema reconciliation

As introduced in chapter 1, various local schemas need to be reconciled to follow a global schema in order to provide end users with a global schema on which users can raise queries. Fundamentally, the following targets need to be achieved for reconciling database schema.

- Modeling Global Schema: builds a global schema which models real-world concepts. It is the schema on which end users raise queries. The global

schema can be built because we assume that all the databases involved represent the concepts in the same application area, and therefore the concepts being modeled present similar attributes. It is supposed to be accomplished at the DIS site.

- Building Exporting Views: gets rid of the heterogeneities between the global schema and each local schema by building views, termed *Exporting View*, on the local schema. The exporting views are in the format of the global schema and represent the information that a source database is going to present to external applications. Each source database needs to provide exporting views and maintain the definitions of those views in a meta-database of its own site. Exporting views are defined with a set of extended relational algebra operations.
- Building Importing Views: maps the exporting views of a source database (DS) to the global schema. LAV is applied to identify the relationship between the global schema and exporting views representing the source databases. Exporting views of each source database are imported as importing views. Subsequently, the importing views are defined as views over the global schema.
- Representing Data in the Meta-database: creates a repository to store the local schema, global schema, and the definitions of the views. The Meta-database is a conceptual single database that in practice can be distributed databases. In this architecture, both DIS and DS have their meta-databases. The DIS creates and maintains a meta-database where the description of the source databases and definition of importing views are stored, while each DS creates and maintains a meta-database where the definition of the exporting views are stored.

Schema reconciliation is further described with its algorithm, language and functions in later chapter 4.

3.3.2 Query Process

Once the schema reconciliation is complete, a user is able to raise a query on the global schema. In theory, the user queries can be in any query language which is a higher level language. As higher level query language is not the focus of this research,

however, we currently use an extended version of the Datalog query language as user query language. Figure 3-2 shows how the queries raised by end users are processed.

- The user query needs to be decomposed into subqueries which are in terms of the importing views representing source databases. In LAV, the importing views are defined on the global schema using *Datalog* [96]. As the user query is also built in terms of the global schema, the *query containment test* [96] is adopted to dynamically find the importing views that can produce data for the user query. Consequently, the source databases providing the importing views are also determined. A DIS conducts this process and sends the subqueries to the relevant source databases (DSs).
- As the schema of an importing view is the same as its corresponding exporting view, those subqueries that are in terms of importing views are also in terms of exporting views. At each DS, the subqueries must be rewritten into queries that are directly in terms of the local schema, based on the definitions of the exporting views in the meta-database. In this thesis, the relational algebra operations are assumed to be the local queries as they can easily be translated into any SQL supported by various dominating DBMSs.

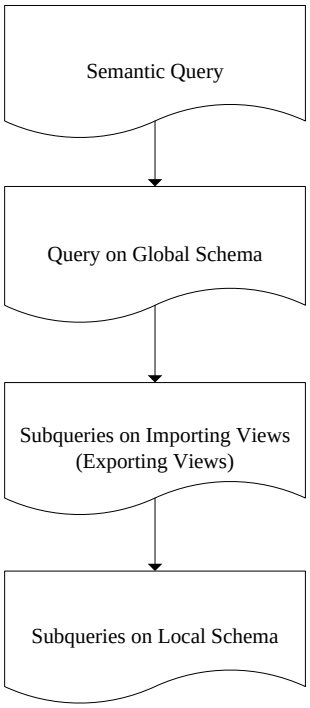

Figure 3-2 The general workflow of query process

Query decomposition is further described with relevant algorithms in chapter 6.

3.3.3 Schema Evolution Detection

The EA-SODIA is designed to build a data integration system which is easier to maintain when any database evolutions occur. As no hard-coded queries exist in the system, the maintenance brought by the database evolution is primarily on the view definitions in the meta-database. This thesis presents a method, called *Schema Evolution Detection*, which automatically modifies the views in the meta-database when a schema evolution occurs. The method works closely together with the approach RSMV as the latter is the prerequisite of the former. Automatic tools for modifying view definitions can be provided based on this method.

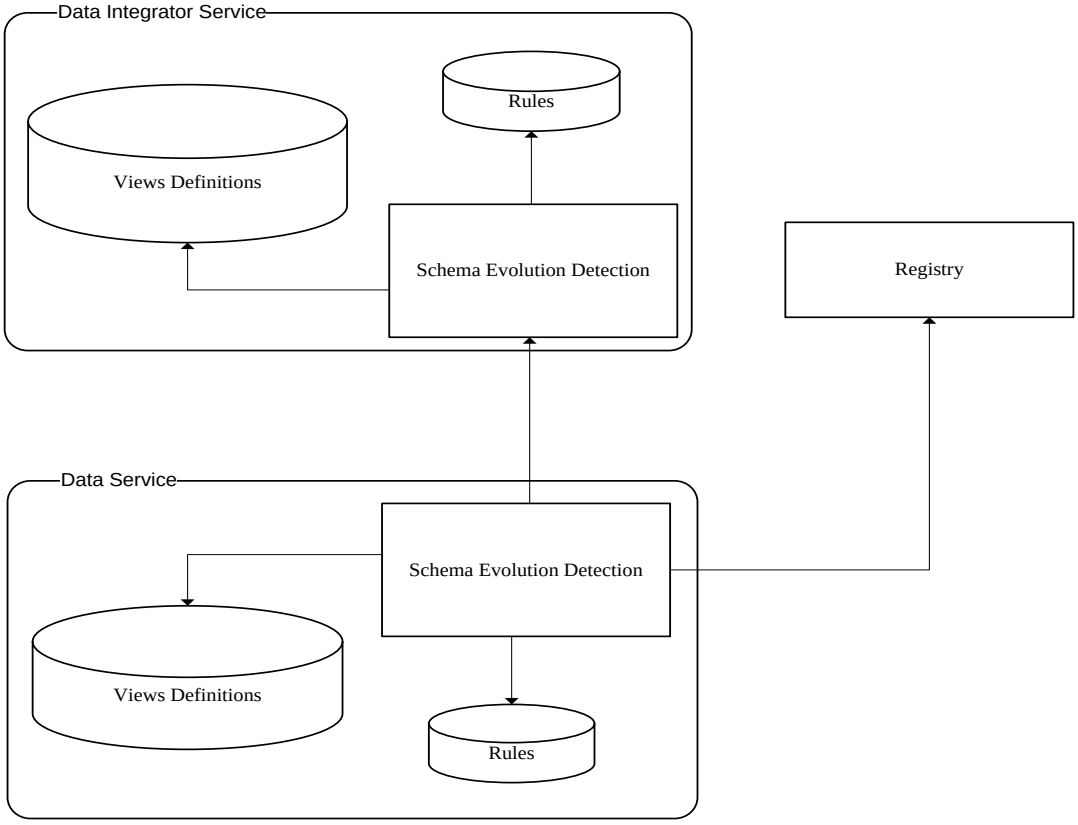

Figure 3-3 Schema Evolution Detection in EA-SODIA

Figure 3-3 illustrates how the relevant components of evolution detection are deployed in the EA-SODIA. The schema evolution detection function exists in both DIS and DS. The schema evolution detection in the DS modifies the exporting views in DS's meta-database, while the schema evolution detection in DIS modifies the corresponding importing views of those exporting views in DIS's meta-database. They work together to detect which view definitions at DIS and DS are affected by the data evolved, and to modify the affected views based on pre-defined roles. For example, when a schema evolution occurs in a DS, the data provider triggers the

schema evolution detection of the DS. The schema evolution detection checks and modifies the exporting view definitions in the meta-database of the DS corresponding to the schema evolution. The schema evolution detection then checks whether the importing view definitions in the DIS require modification. If it does, the schema evolution detection of the DS calls the schema evolution detection of the DIS which checks and modifies the importing views relevant to the changed DS. A registry service is designed to record basic information about all the DISs, such as the location and the name of a service. The schema evolution detection of a DS can find all the DISs in the registry service. This may considerably reduce the cost of maintenance brought by the database evolution. The algorithms and the roles of evolution detection are further described in chapter 6.

3.4 Data Integrator Service

Data Integrator Service (DIS) is the service that exposes the global schema to users and can receive a query from a user. It involves the following components to undertake different processes:

- The *Query Translator* receives queries from users, and translates those queries to Datalog queries in terms of global schema. The queries sent by end users may be higher level queries which are not Datalog queries required by the DIS to undertake the containment test. However, translating higher level queries to Datalog queries is not our focus in this research. For simplicity, we assume that the end users send the queries which are in an extended version of Datalog language.
- The *Query Decomposer* decomposes the query into subqueries that are in terms of importing views which represent source databases based on view definitions stored in the meta-database.
- The *Query Optimizer* is a common component of a data integration system or a distributed databases systems, which optimizes queries and then improves the performance of the system. However, it will not be described in this thesis since Query Optimization is beyond the scope of this research.
- The *Query Delivery and Result Composition:* further divides the subqueries into smaller pieces each of which is relevant to a single source database. It then sends those queries to corresponding DSs which will return the results back to DIS. The

DIS acts the same way as the mediator in a mediated data integration approach.

- *Meta-database (MDB)* is a database where metadata of both the global schema and the importing views of each source database. This is the most important component in this architecture as most processes involved in this architecture are based on the information in the meta-database. The schema evolution detection particularly relies on the view definitions in the meta-database.
- *Schema Evolution Detection* is another important component which is invoked by the DS where changes in the local database schema occur. It aims at checking which importing views of that DS are affected by the changes. It then modifies the affected views probably with human intervention.

The dashed rectangles in the DIS shown in Figure 3-1 represent the components and processes which are not involved in this research and are therefore not described in further detail. In general, the function which is usually accessed by users is the Query Translator, while Evolution Detection is mostly invoked by DSs.

Although there is only one DIS in the architecture illustrated in Figure 3-1, more DISs that are replications of each other can be used in order to improve the performance. Every DIS is registered into the registry so that users and DSs and other applications are able to find the correct DIS. For example, the name of a relation schema in one of the source databases changed. In order for user queries to run properly, the exporting views involving that relation schema have to be modified correspondingly. Consequently, the corresponding importing views at DIS site may need to be modified in order to keep the consistency with relevant exporting views. Therefore, the DS needs to search the registry to find all relevant DISs and invoke the evolution detections of the DISs to keep the importing views up to date.

3.5 Data Services

DSs are the services which accept the queries sent by a DIS and then translate those queries which are the exporting views into queries over the local schema. A DS exposes the exporting views as its schema for external applications (e.g. integrator services). Each database provider needs to expose their database as a DS which can be accessed by DIS and other external applications via a SOAP message which involves

the name of the method and the query which can be processed by that method. The queries transferred within SOAP messages are in a standard query language agreed by all the DISs and DSs involved in the system, conjunctive query is used in this research.

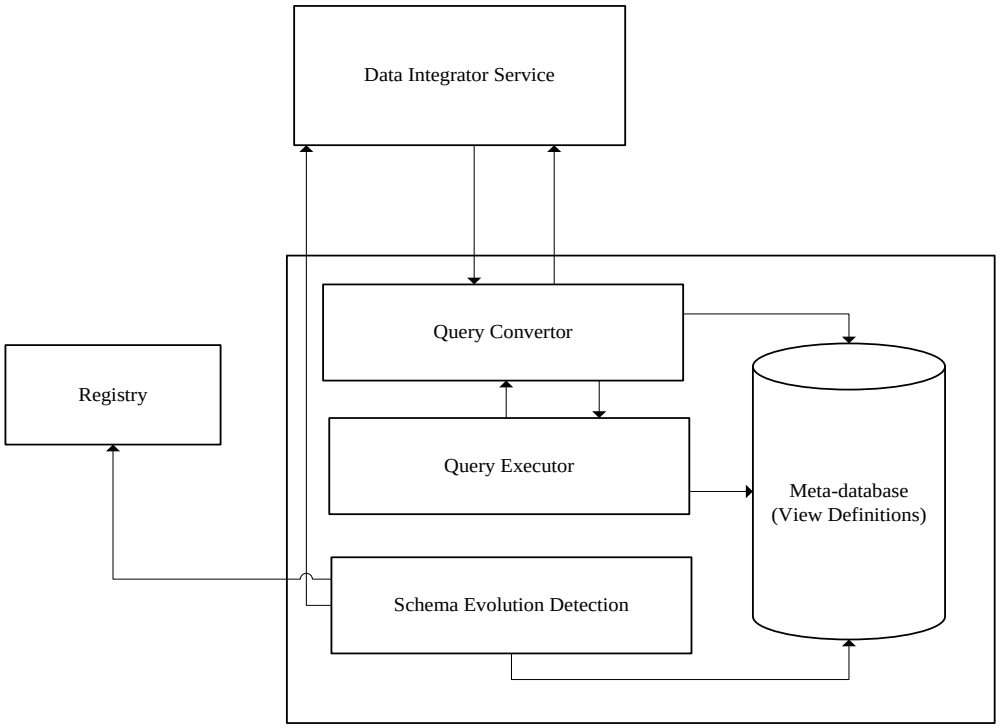

Figure 3-4 The Structure of Data Service

Figure 3-4 shows the components of a DS and how they work cooperatively. Four components are involved in a DS as follows:

- Query Converter: accepts the Datalog queries sent from the DIS and then convert those queries into the queries in the query language which can be understood by this DS and is used to build the exporting views on the underlying source database. In this research, every DS builds its exporting views using relational algebra. Therefore, the translated query is still on the exporting views, but expressed in terms of relational algebra operators.
- Query Executor: decomposes the translated query based on the definitions of the exporting views and produce subqueries on the underlying local database schema. It then executes these subqueries on the local database and composes the results which will be returned to the query converter. As the decomposed subqueries are still in relational algebra, the query executor is also responsible for converting those subqueries into queries in a query language (e.g. SQL) which can be

executed in the local database management system.

- Schema Evolution Detection: is responsible for detecting any changes in the schema of the local database and examining whether the changes have affected the definitions of the exporting views. If this is the case, it then applies modifications automatically to the views definitions. This component works together with the evolution detector of the DIS to keep the consistency between the exporting views at the DS site and the importing views at the DIS site.
- Meta-database: is the database where the local schema and the exporting views of the source database (DS) are stored. It also enables the evolution detector to detect changes on local schema and modify the affected exporting views simply by retrieving and updating the meta-database.

Each database provider needs to publish its database as a DS. Therefore, the DS and the source database have a one-to-one relationship, meaning that a DS uniquely represents a source database in this research. The database exposed by a single DS is usually a database which is managed by a single DBMS (e.g. MySQL, Oracle or DB2), although in practice it can be another distributed database which has a global schema and is able to accept and execute queries on it. In the latter case, the query executor needs to invoke an extra component which can convert the queries in relational algebra into queries in the query language which can be understood by the distributed database management system. Even if the exposed database is a single database, the query executor has to convert the subqueries into queries in the language which is supported by the local DBMS. For the time being, we focus on the case where the underlying databases are all single databases and the query language supported by all of the databases is standard SQL, as the conversion between the relational algebra and other languages is beyond the scope of this research. In addition, how to involve distributed databases as local databases is another problem which is left for future work and is further discussed in a later chapter.

3.6 Registry Service

In SOA and web service architecture, registry service is a central component to enable dynamic service discovery. However, the current UDDI specifications fail to provide such information as quality of service and semantic information for an external

application to not only understand how to access the service but also examine which one is providing better service suitable for the external application. As such, in this research, it was initially designed that the registry should store the metadata of all the databases involved so that DISs can compose queries on the fly based on information in the registry. However, a replication of the meta-database in each DIS is eventually proposed for the following reasons:

- Current specifications of the registry in both SOA and web service architecture fail to provide the metadata of the source database of a DS.
- Although the registry can be extended to describe the semantic information of databases, the access to the registry can be extremely intensive when there are a large amount of DISs. Building a meta-database in each integrator service may avoid more interaction across the network and increase the performance of the integrator services.

Therefore, the registry in this architecture is rather a simple service which aims at enabling DSs to find how to access all the DISs when the schema of the database in a DS has changed. A DS needs to access the registry to obtain information of the DISs only when the importing views stored in the DISs require modification. In this architecture, a DIS does not access the registry to gain the information of a DS when a user raises a query.

Consequently, the registry service records only some basic information of DISs shown as follows:

- Names and URIs of all the DISs involved in the integrated system.
- The name of the schema evolution detection function of every DIS which can be accessed by a DS.

As each DS must record the location and the access method of the registry, every DS requires modification if the registry changes. However, it is assumed in this research that the registry is managed by the administrators of the integrated system and therefore does not change frequently. Although the access to the registry may increase when DSs change frequent, we assume that the user queries come more frequently than the evolution of the DSs.

3.7 Characteristics of Architecture

This section discusses various characteristics of EA-SODIA by comparing it with current approaches such as data warehousing, federated databases and data mediation which have been described in chapter 1. The characteristics examined are the flexibility and the scalability of the approaches, the complexity of creation and maintenance of the system based on those approaches and the performance and the completeness of the result produced. By complete result we mean that a result of a query is referred to as a complete result if the result has all the tuples which can result from all the databases in the integrated system when applying the query to each of the databases. Namely, the integrated system can find an answer to a query as long as such an answer exists.

Table 3-1 shows some simple results of comparison on the above characteristics between EA-SODIA and current major architectures.

	Flexibility	**Scalability**	**Complexity of Creation**	**Complexity of Maintenance**	**Performance**	**Complete-ness of result**
Data Warehousing	Low	Low	High	High	High	Complete
Tightly Coupled Data Federation	Low	Low	High	High	Low	Complete
Loosely Coupled Data Federation	Medium	Medium	High	High	Low	Complete
Mediation	High	High	Medium	Medium	Medium	Incomplete
EA-SODIA	High	High	Low	Low	Medium	Incomplete

Table 3-1 Comparison of characteristics among major approaches

Data warehousing aims at integrating data from various sources by providing a unified view of them. It also aims at providing summarized and analytical information requested by users in relatively short response-time. This goal is archived by pre-storing the filtered data of the unified view in a single repository so that end users can query on this repository instead of accessing a large set of databases. Therefore, it may produce the best performance when end users ask queries on the unified view.

Because each involved database is integrated and accessed to produce the virtual view as long as it has the data requested, a data warehousing system can produce a complete result to the end user. However, data warehousing may provide lower flexibility and scalability in some environments. It becomes difficult to build a data warehousing system when the number of data sources becomes very large. It is not flexible enough because it can not ask the queries which were not expected when building the system.

In addition, the cost of creation and maintenance can be extremely high when the number of databases is very large and/or the sources are likely to be updated frequently. This is due to the fact that each database requires a wrapper to load and pre-process its data and an efficient approach to refreshing the data warehouse when the data in data sources has been updated. Data evolution, the major focus of our research, is currently an unsolved problem in maintenance of the system. As the data of the unified view is pre-stored and refreshed periodically, data warehousing does not allow changes in data sources and in their schema.

Data federation provides semi-autonomy for each component (database system) involved. The source database can not be called “fully-autonomous” because each database is modified by adding an interface that allows communication with other databases in the federation. A tightly coupled federated database system is static and inflexible, as it does not allow adding or removing databases easily. It produces relatively lower scalability due to the large number of interfaces that need to be written for each source database to communicate with all the others. A loosely coupled federated database system has similar problems. Although it makes the source databases more autonomous, it requires a huge amount of work for the schema translations and becomes too tedious with a large number of databases in a federation. Consequently, creation of a data federation is a complex and time-consuming work. Similar to data warehousing, a federated database system is hard to maintain and evolve. When compared with data warehousing, it provides lower performance, but provides updated and complete results each time.

Comparing with the above architectures, a mediated system is more flexible and more scalable. The involved source databases have complete autonomy which means it is

easier to add or remove source databases. Instead of writing interfaces for each database to communicate with all the others, wrappers need to be built for each database to make its own data accessible to the mediator. Consequently, less work is required to create and maintain a mediated system. A mediated system, which adopts LAV as the approach to integrating source databases and creating the global view, is able to remove or add a source database more easily. LAV is also able to produce a better performance than a federated database system does when the number of source databases is very large. However, as the query process of LAV is based on query containment test, a mediated system is more likely to provide incomplete result. The discussion of finding complete result for a query using LAV can be found in [68]. Again, data evolution is still a problem.

EA-SODIA is to some degree similar to the mediation approach, providing better flexibility and scalability. It is more flexible and scalable than a mediated system because service-oriented architecture is adopted which exposes source databases as DSs instead of building wrappers for each database. Moreover, RSMV makes an EA-SODIA system more flexible and feasible, as it tackles schema reconciliation at each source database by creating views in the meta-database instead of writing hard-coded programs. It also reduces the cost of creation and maintenance of the system. The most important feature, the focus in this research, is that data evolution is handled much easier in an EA-SODIA because there are no hard-coded queries and programs for schema reconciliation. As LAV is used to populate the global view, the results may be incomplete.

To sum up, data warehousing is preferable when source databases are permanent, do not get upgraded too often and it is easy to predict what kind of queries the users will ask. It provides better performance than the others. However, data evolution may be a nightmare for the maintainers of a data warehouse system. Among the two traditional architectures based on the virtual view approach, the mediated approach is chosen more often, as creating and maintaining federated systems are costly and time-consuming. In the environment where source databases are evolving frequently, however, EA-SODIA may be preferable to the others above.

3.8 Summary

This chapter presents an overview of the Evolution Adaptive Service-Oriented Data Integration Architecture (EA-SODIA). Each service in the architecture is introduced with its functionality. The DIS dynamically binds the DSs based on the algorithm, RSMV. The RSMV also eliminates the heterogeneities between the local schemas and the global schema by building views. Both the DIS and the DSs maintain a meta-database that store the metadata of the local schemas and the global schema and mappings (importing views and exporting views). The function, schema evolution detection, is provided at both DIS and DS to work together to tackle the schema evolution problem. The algorithms, RSMV and schema evolution detection and query decompositions, are introduced in detail in chapter 4, 5 and 6, respectively.

Chapter 4 Schema Reconciliation and Meta-database

4.1 Introduction

Chapter 3 presents an overview of the Evolution Adaptive Service-Oriented Data Integration Architecture (EA-SODIA) which is easier to maintain when the databases integrated evolve. Each component in the architecture is introduced. This chapter describes, in detail, how the schemas of various databases to be integrated are reconciled by constructing mappings from each local database schema to the global schema using the algorithm Relational Schema Mapping by Views (RSMV) in order for the architecture to be maintained more easily when changes of the databases occur. More importantly, representing view definitions resulting from RSMV in the meta-database is a crucial part of the research which prepares the integrated database system for evolution detection and automatic view modification.

An overview of schema reconciliation with its three basic steps to integrate various source databases is presented, followed by a detailed and formal description of each step. The three steps are Design of Global Schema, Integrating Source Databases and Conversion from Local Schema to Global Schema, respectively. The latter two comprise RSMV. The rules and functions that are required to complete those steps are defined throughout each section. The results of RSMV, view definitions (mappings), are defined by set theory instead of hard-coded programs and stored in a meta-database. It is explained in detail how relations and views and relational algebra operators are represented as set. The case study produced in chapter 7 is used to demonstrate how schema reconciliation can be achieved.

4.2 Overview of Schema Reconciliation and RSMV

One of the goals of the architecture in this research is to provide end users with a unified view over the data in various underlying databases as if they were a single database. In this work, a global schema which comprises of virtual views is designed on which users can raise queries as if the global database is materialized (data are physically stored). However, the real data are actually stored in underling source databases which are in fact the ones users intend to access. Therefore, one of the roles of the integrated system is to accept a query over global schema from a user and access the corresponding source databases individually for data and then combine those data to produce a final answer to the user. The schema of a source database is called *Local Schema* in this research. Obviously, one of the main tasks in the design of a data integration system is to establish the mapping between global schema and the source databases, in order for the system to understand which source databases to access and how to access them. In this work, the task is called Schema Reconciliation.

We propose an approach to achieving Schema Reconciliation by establishing mappings between local schemas to the global schema. The approach is called Relational Schema Mapping by Views (RSMV) which adopts some concepts from both Schema Mapping and Local as View (LAV). It builds a mapping between each local schema and the global schema individually. Figure 4-1 shows the general process of Schema Reconciliation using RSMV.

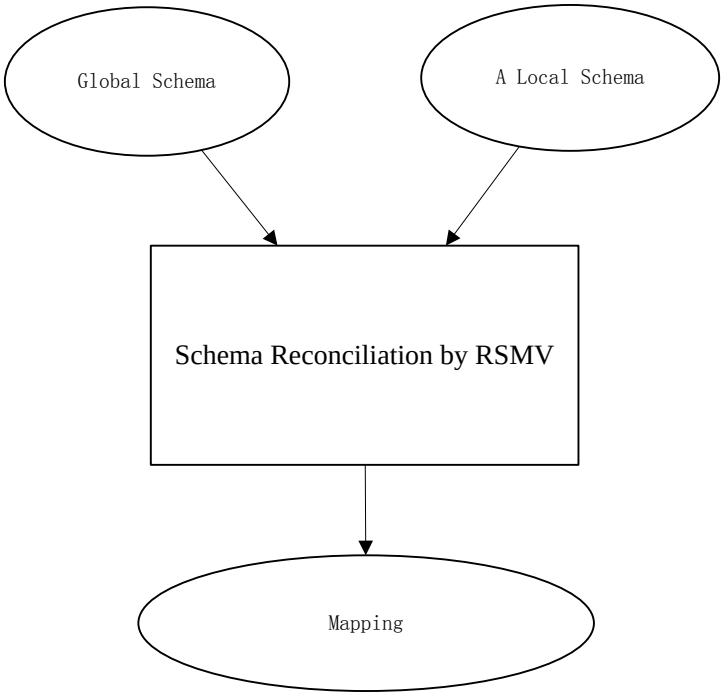

Figure 4-1 Process of Schema Reconciliation by RSMV

It can be seen from Figure 4-1 that the input of RSMV are the global schema and a local schema and the output is the mapping between them. Each source database, which is to be involved in the integrated system, needs to go through the above process using RSMV.

In order to describe the approach formally, let *{R1, R2,…,Rn}* be the set of relation schemas in the global schema, we denote the global schema as ***GS{Ri} (i = N)*** which is the set of all relation schemas in the global schema. Let *{S1, S2,…,Sn}* be the set of relation schemas in a source database *D*, a local schema of a source database is defined as a set $\boldsymbol{LS^D(Si)\ (i=N)}$ which is the set of all relation schemas in *D*. The mapping between the global schema *GS* and a local schema LS^D is denoted as ***M***. Therefore, the process Schema Reconciliation by RSMV is defined as a function:

$$\boldsymbol{RSMV: (GS, LS^D) \rightarrow M}$$

The function means that given a pair of a local schema and the global schema, the

RSMV produces a mapping between them. Namely, for each source database participating in the integrated system, there is a mapping between the global schema and its local schema. Therefore, let the set {D1, D2,…Dn} be the set of all source databases participating in the integrated system, the integration of those databases can be defined as:

$$\boldsymbol{I\ \{(GS, LS^{Di}, Mi)\}\ (i=n)}$$

It is a set of triples each of which denotes the integration from a local schema to the global schema. We will define the mapping *M* in more detail later.

Designing the global schema may be the first step of constructing the integrated system, as all the local schemas are to be integrated into the system by establishing mappings between the global schema and them. Therefore, the schema reconciliation can be conducted on the premise that the global schema already exists. Schema reconciliation by RSMV involves two basic steps to integrate various source databases, which are described as follows:

- ***Eliminating Heterogeneities between Local Schema and Global Schema***: this step is to map each local database schema to the global schema by defining exporting views over the local database, in order to eliminate heterogeneity. Most types of heterogeneity will be tackled through this step. Therefore, the exporting views built on local database schema are homogeneous to the global schema (It will be formally defined how the views can be homogeneous to the global schema in a later section). Relational algebra operations are used to define the exporting views, because in this research the source databases are all relational databases and the relational algebra is easily translated into other languages. The exporting views are then ready to use in the next step.

- ***Integrating Source Databases*****:** once each database schema has been converted to

comply with the global schema by building exporting views at its own site, the exporting views are then ready to be integrated with the global schema. Integrating with the global schema aims to build a relationship between the global schema and the local schema, indicating what information a source database is providing. Therefore, the Query Composer can find which source databases contain the information required by an end user when the end user raises a query on the global schema. LAV is used in this step to define exporting views representing a source database as importing views over the global schema.

The results of the above two steps are exporting views over the local schema and importing views over the global schema, respectively. The importing views and the exporting views are associated with each other to constitute the mapping between the local schema and the global schema. The process of RSMV can be further illustrated in Figure 4-2.

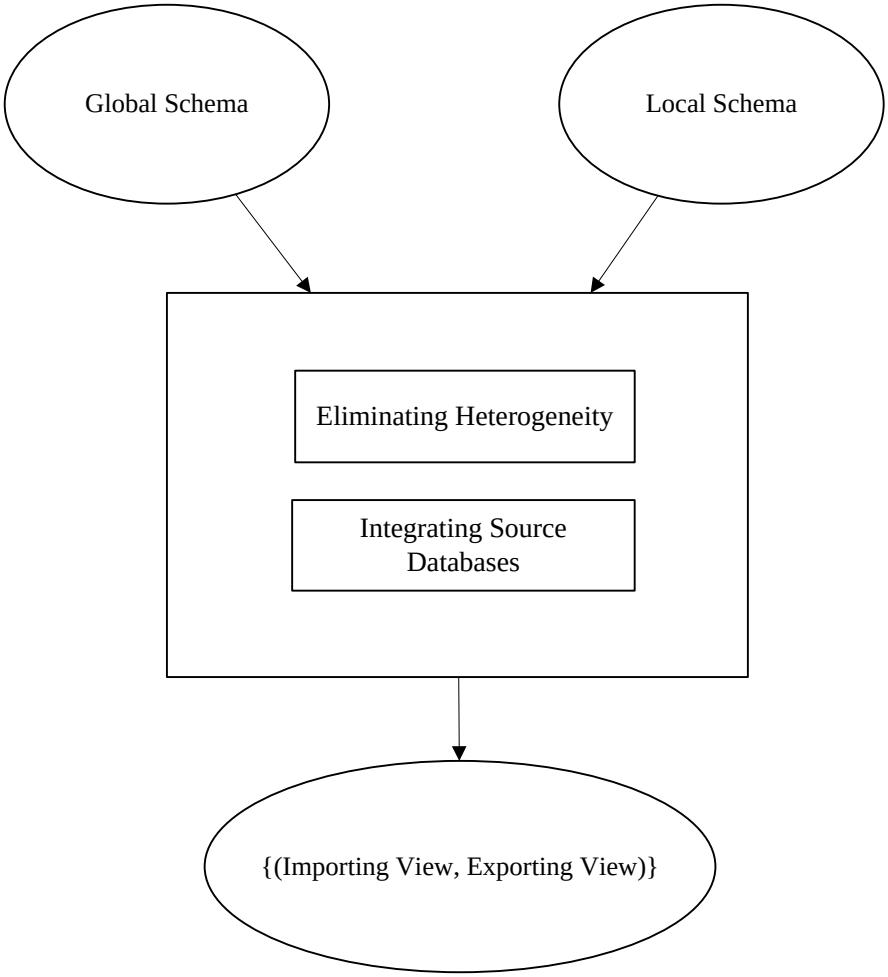

Figure 4-2 The Process of RSMV with Its Two Steps

In fact, the relationship between importing views and exporting views is a one-to-one relationship. That is, for each importing view, there is an exporting view such that they have the same schema (same name and attributes). The difference between them is their definition. The importing view is defined over the global schema, while the exporting view is defined in terms of the local schema. Consequently, the relationship between the global schema and the local schema is indirectly established.

Having introduced the importing views and the exporting views resulting from the above two steps, we can now formally define the mapping *M*. Let {*ImVi*} be a set of importing views defined in terms of the global schema *GS*, and {*ExVi*} be a set of exporting views defined in terms of the local schema *LS* of a source database *D* , the mapping from *LS* to *GS* is a set of pair (ImVi, ExVi), denoted as:

$$M^{LS \rightarrow GS}\{(ImVi, ExVi)\}$$

The function RSMV can then be denoted as:

$$RSMV: (GS, LS^{D}) \rightarrow M^{LS \rightarrow GS}\{(ImVi, ExVi)\}$$

We will describe importing views and exporting views formally in more detail in later sections.

The most important thing in this research is that the exporting views and importing views are stored in a meta-database along with the global schema and all local schemas and other information about the source databases and their DS. It turns out that integrating a source database requires no hard-coded programs or queries so that the automatic maintenance can be undertaken when a source database evolves. The meta-database is described in detail in a later section of this chapter and in Chapter 6.

4.3 Design of Global Schema

In order for the integration system to provide end users with a unified view of various distributed databases, a global schema needs to be created on which end users can raise queries as if there is only one database. Although different database designers may model real-world concepts in different ways, in one application area there must be similarity in those databases because they model the same concepts. The global schema should be derived from the requirements of the integration system.

Although there are a couple of models to semantically represent a global schema, in this work, for simplicity, the global schema is designed in a relational model. However, our approach has the potential to work together with other approaches which focus on other issues of database integration. The results of this step are:

- GS {Ri}: a set of relation schemas representing real-world concepts to be

modelled in the integrated system. These relation schemas are the schemas to which all local schemas are supposed to map.

- Organizational structure of all data providers: a hierarchical structure of all database providers which are involved in the integrated systems. This is represented as a tree in this work.
- A_G: a set of all attributes which appear in relations in the global schema, called **Global Attribute Domain**. The formal and more detailed definition is given shortly.

4.3.1 Global Schema

In [82], it is explained that the name of a relation and the set of attributes for a relation is called the schema for that relation. In the relation model, a database consists of one or more relations. The set of schemas for the relations of a database is called a relational database schema, or just a database schema.

In this work, the global schema can be referred to as a relational database schema, as the integrated system makes users feel that they are accessing a single database with the global schema. Therefore, as defined in the previous section, a global schema is a database schema which consists of relation schemas. Usually, a relation schema is represented as:

R (A1, A2,…,An)

Where (A1,A2,…,An) is a set of all attributes of the relation R. In a later section introducing the meta-database, we will try to represent and store the relation schema in the meta-database.

The third normal form of database may not need to be considered very much when designing the global schema because the integrated database system is not materialized. However, there are still a few rules for designing the global schema in order for schema reconciliation to be conducted easily.

- Every relation in the global schema must be in at least first normal form.

- No composite attributes exist in any relation in the global schema. For example, the property Name should be designed as a single attribute “Name” instead of three attributes “FirstName”,” MidName” and “LastName”.

However, the design of the global schema relies on, to a large degree, experience and expertise on both domain knowledge and database design. It also needs to refer to existing distributed databases to be integrated. To sum up, an effective global schema should be well designed to fulfil the requirements of queries on the integrated system as well as be easy to associate with various local database schemas.

4.3.2 Organizational Structure of Source Databases

The organizational structure of databases categorizes the local databases (more precisely database providers) by their various properties, such as location and business type. Take the example introduced in Chapter 3, a database provider (an enterprise) may be located in Newcastle, while another may be in Durham which further belongs to County Durham. Users should be able to raise a query asking for a computer from a store in Newcastle. Thus, the integrated system should be able to access only the databases held in Newcastle and examine if there is one required computer in stock.

The organizational structure of source databases is not an essential part of building the integrated system from a technical perspective. The goal of it is to facilitate various queries to narrow the search space or analyze and summarize the data. However, most database application systems and database integration systems require querying by various characteristics. It is taken into account in this research because in previous projects, where the queries were written in hard-coded programs, the evolution of the organizational structure gave rise to a large amount of maintenance work. This was mentioned in Chapter 1.

Database providers can be categorized by several properties. One categorization can

be represented as a tree. For example, given a set of locations L {a, b, c, d, e, f} and a set of databases D {D1, D2, D3, D4, D5}, the organizational structure is shown in Figure 4-3.

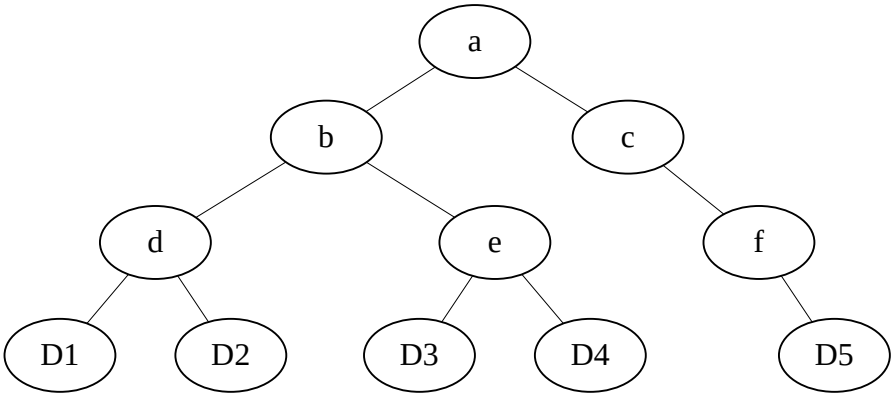

Figure 4-3 A Tree of Organizational Structure of Source Databases

It is illustrated in Figure 4-3 that each node of the tree represents a database or a location. A parent-child relation between two nodes represents a Locates-At relationship between the elements on the nodes. For example, the relation "D1 is a child of the parent d" indicates the relationship "D1 locates at d". All parent-child relations can be defined as a set P{(a, b), (a, c), (b, d), (b, e), (c, f), (d, D1), (d, D2), (e, D3), (e, D4), (f, D5)}, which is a 2-place relation over L $\cup$ D. It is shown apparently in Figure 4-3 that the leaves on the tree are all databases and vice versa. Such properties as locations are called Categorization Property (CP) in this work.

We can then define an organizational structure tree formally. Let D be a set of source databases and P is a set of CPs, the organizational structure tree is a pair **(T, R)** where T is the union of D and P (D $\cup$ P) and R is a 2-place relation over T such that for each element a, a $\in$ T, if a has a parent x, x $\in$ T, then (x, a) $\in$ R.

Having given the formal definition, the organizational structure tree of the above example can be then represented as:

({a,b,c,d,e,f,D1,D2,D3,D4,D5}, {(a, b), (a, c), (b, d), (b, e), (c, f), (d, D1), (d, D2), (e,

D3), (e, D4), (f, D5)})

4.3.3 Global Attribute Domain

Global attribute domain is in fact a ramification of the global schema because it is derived from the global schema GS. However, it is described explicitly as a separate set in this research as it is one of the criteria to examine the effectiveness of eliminating the heterogeneity between the global schema and the local schema. Assume that we have designed a global schema GS which is a set of relation schema {Ri}(i≤n), we can define the global attribute domain. Let Ai be a set of all attributes of Ri, Ri ∈ GS, the global attribute domain of GS is $\mathbf{D_{GA} = \cup_{i \leq n}\{Ai\}}$ such that for each Ri, Ri ∈ GS, if Ai is a set of all attributes of Ri then Ai $\in D_{GA}$.

It is called global attribute domain because the attributes of all the relation schemas of the global schema are in it. It will be used to examine if an exporting view over a local schema is effective and homogeneous to the global schema. This will be introduced shortly in the next section.

4.4 Eliminating Heterogeneities between Local Schema and Global Schema

Having explained how to design a global schema, this section introduces the first step of RSMV to map each local database schema to the global schema by defining exporting views on the local database. The aim of this step is to eliminate heterogeneities between the local schemas and the global schema. More rigorously, it is aimed at eliminating heterogeneity among the local schema. Therefore, the existence of the global schema is not only for providing the unified view to users, but also for providing a standard schema in order for the local schemas to be integrated. It only needs to be considered how to map from a local schema to the global schema when integrating a source database, instead of taking all other local schemas into account.

The various types of heterogeneity existing among database schemas have been formally defined in Chapter 1. Most of them need to be tackled in this step. As mentioned above, in this research, each local schema is mapped to the global schema individually. In order to eliminate the heterogeneities, the local schema needs to be reconciled so that it is homogeneous to the global schema. This is achieved by building views in terms of the local schema using a set of extended relational algebra operators. The resulted set of views, called Exporting views, must be homogeneous to the global schema.

A view is referred to as a derived virtual relation resulting from a query in terms of one or more relations which should then have a name and a set of attributes as its schema. Therefore, we can now formally define the term Homogeneous. Let R be a relation schema and S{S1, S2,…, Sn} be a set of relation schemas, and D_S is the global attribute domain of S, R is homogeneous to S, denoted as Homo (R, S) → TRUE, if:

- Let A be the set of all attributes of R, then A $\subseteq$ D_S.
- There is a set P{S1, S2…, Sm}, P $\subseteq$ S, such that R can be defined as a view in terms of P.

Consequently, Let R {R1, R2,…, Rn} and S{S1, S2,…, Sn} be two sets of relation schemas, and D_S is the global attribute domain of S, R is homogeneous to S, denoted as Homo (R, S) → TRUE, if for each Ri, Ri $\in$ R, Homo (Ri, S) → TRUE

Subsequently, we can describe the process of eliminating heterogeneities formally. Let GS be the global schema and LS be the local schema, the process of eliminating heterogeneities is described as a function:

EH: LS → ExV

where ExV is a set of views {ExVi} such that Homo (ExV, GS) → TRUE.

Although it is called Eliminating Heterogeneities, this step does not aim at eliminating all the heterogeneities and gaining a set of relations which are exactly identical to the relations in the global schema. It aims at building a set of exporting views which are homogeneous to the global schema so that they can be defined as views in terms of the global schema. The entire process of eliminating all the heterogeneities relies on not only this step but also the next step and query processing.

4.4.1 Relational Algebra Operators

Eliminating heterogeneities, in this work, relies on a set of relational algebra operators which is a basic query language. An algebra, in general, consists of operators and atomic operands. Relational algebra is a special algebra whose atomic operands are:

1. Variables that stand for relations.
2. Constants, which are finite relations.

Generally, relational algebra consists of some simple but powerful ways to construct new relations from given relations. When the given relations are stored data, then the constructed relations can be answers to queries about this data.[84,98]

A set of extended algebra operators is used to construct exporting views in terms of the local schema. Those operators have been slightly modified to fulfil the needs in this work. The complete description of the algebra operators used in this work can be found in section A.1 of Appendix.

4.4.2 Exporting views

4.4.2.1 Expression Tree of a View

Writing single algebra operations on one or two relations as queries does not show the power that the relational algebra has. However, the algebra operations take relations as operands and the result of an operation is still a relation. Therefore, it is allowed to form an expression of arbitrary complexity by applying operations to the result of other operations. Consequently, more complex queries can then be constructed by

forming complex expressions. An expression can be represented as an expression tree. The introduction and an example can be found in section A.2 of Appendix. The explanation of an *atomic view* is also presented in section A.2 of Appendix.

4.4.2.2 Representation of an Exporting view

An **exporting view** in this work is in fact a resulting virtual relation of a query that is constructed by an expression of relational algebra over relation schemas in a local schema. As the expression can be represented as an expression tree with a sequence of temporary views, for each exporting view, there is a tree structure to describe it formally. Let R {R1, R2, …, Rn} be a set of relation schemas in a local schema LS, if ExVi is an exporting view defined over R and V {V1, V2, .., Vn} is a set of all atomic views on the expression tree such that $R \cap V = \emptyset$, then ExVi is represented as a triple:

ExVi **(V, Vi, R)**

where Vi is an atomic view on the root of the expression tree, $Vi \in V$, and V is a set of all atomic views on the tree and R is a set of all the relation schemas on all the leaves.

If the expression tree shown in Figure 4-4 defines an exporting view, the exporting view is represented as ExVi ({V01, V02, V03, V04, ExVi}, ExVi, {R, S}).

Although an exporting view is actually a normal view which is introduced in most database texts, we represent it differently as a tree, for the reason that it is easy for the evolution detection to search and modify an exporting view automatically.

4.4.2.3 Rules of Building Exporting views

In this research, in order to represent the views in the meta-database, several rules need to be followed when constructing exporting views. Let ExV be a set of all exporting views of a local schema LS, ExVi (V, Vi, R) be an exporting view, $ExVi \in ExV$, then:

1. The schema of the exporting view is the schema of the view on the root Vi.

2. For each Ri, Ri ∈R, Ri must be a base relation schema in the local schema LS.
3. For each Vj, Vj ∈V, Vj must be an atomic view.
4. For each view Vj, Vj ∈ V, the attributes of Vj must be identical to the attributes of the resulting relation of Vj's expression. For example, V is a view derived from the expression R (a, b, c) ∩ S (a, b, c). The attributes of the resulting relation are (a, b, c), then the schema of V must be V (a, b, c).
5. Let ExVj be another exporting view, ExVj ∈ExV, and let B(B1, …, Bm) be a set of all attributes of ExVj, if A is a set of all attribute of ExVi, then for each Bk, Bk ∈B, Bk ∉ A. (Namely, there is no another exporting view in ExV such that there is one or more attributes that are also identical to the attributes of ExVi).

There is also another rule to follow in order for the evolution detection to modify the views more efficiently afterwards. The rule dictates that for each relation in R, a view must be built at first on the relation applying the projection operator on every attribute of the relation. This rule is to make sure that all the attributes of every relation in an exporting view must be renamed or combined to produce new attributes. Applying the projection operation is divided into three cases:

1. If an attribute of the relation is not chosen as an attribute in the root view, then this attribute is given a new name which is the same as its current name.
2. If an attribute of the relation is chosen as an attribute in the root view, then this attribute is given a new attribute name which is the same as the name of the corresponding attribute in the global attribute domain.
3. If two or more attributes of the relation are combined into a single attribute that will appear in the root view, then these attributes are combined into a single attribute that is given a new name which is same as the name of the corresponding attribute in the global attribute domain.

Having applied this rule, each relation will only be taken as an operand by a projection operator of an atomic view, in order to give each attribute of the relation a new name, it does not matter if the new name is the same to the current name.

4.4.2.4 Integration of Source Databases

4.4.2.5 Importing Views

So far, the approach to eliminating the heterogeneity between local schemas and global schema has been introduced. The local schema can then become homogeneous to the global schema by building exporting views. However, the source databases still have not been associated with the global schema so that a query raised on the global schema can be decomposed and delivered to the relevant source databases. The process of constructing relations between the global schema and local schemas is called integration of source databases.

The exporting views built on a local schema represent the information that is provided by that local schema. Therefore, integration of source databases can be referred to as integration of exporting views and the global schema. In order to integrate the exporting views to the global schema in this research, the set of importing views corresponding to the exporting views are presented. Each importing view and its corresponding exporting view have an identical set of attributes, although they can have different names. Thus, building a relationship between importing views and the global schema is the way the source databases are integrated. The approach, called LAV, is adopted to integrate source databases. Figure 4-5 shows the integration of a source database and the global schema.

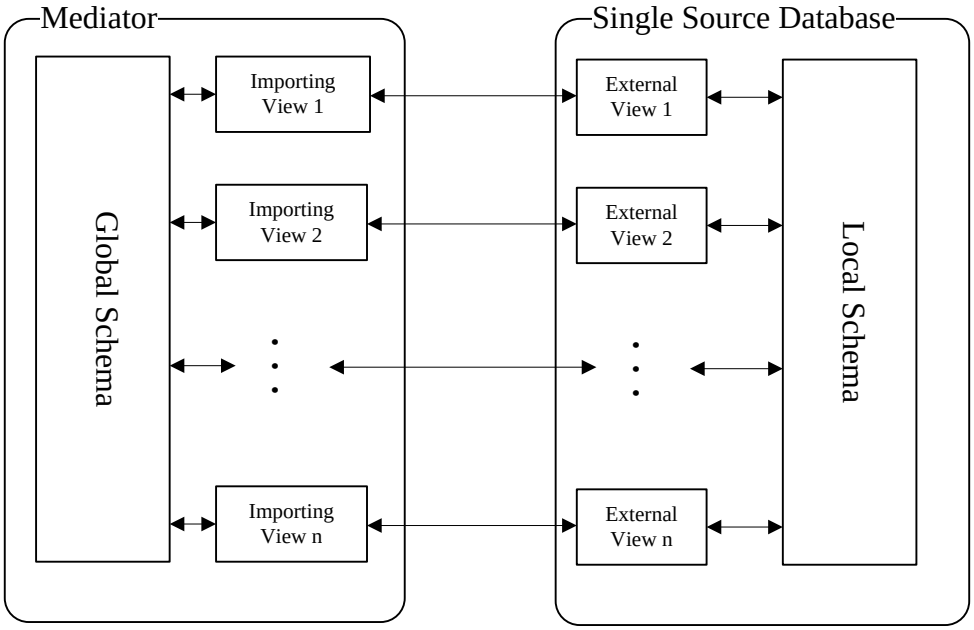

Figure 4-5 The integration of a source database and the global schema

In Figure 4-5, it is illustrated that the importing views and the exporting views have a one-to-one identical relationship. A related pair of an importing view and an exporting view has an identical set of attributes. The only difference between them is that the algebraic expression of an exporting view is defined over the local schema, while the expression of an importing view is defined over the global schema. Therefore, we can describe the relationship between a local schema and the global schema. Let ExV be an exporting view on LS and ImV be the corresponding importing view of ExV on GS, if Q is a query, the relationship between LS and GS is represented as:

$$\mathbf{Q(ExV) \subseteq Q(ImV)}$$

It means that the query Q on ExV provides a subset of answer to Q on ImV. In this way, the local schema is associated with the global schema.

4.4.3 Local as Views (LAV)

LAV is an approach to connecting sources with a global schema. In the local-as-view approach, the relation schemas in the global schema are referred to as global predicates. In order to avoid confusion, we still use the term global schema. The importing views representing the source database are defined as views in terms of the

global schema using an expression consisting of Datalog rules. A query that is a single Datalog rule is often called a conjunctive query. The query to define the importing view as a view over the global schema is a conjunctive query and we will use this term in the remaining chapters.

Assume that there are two relation schemas R (a, b, c, d) and S (a, e) in the global schema, and an importing view ImV (a, b, c, e) of a local schema, the query to define the importing view can be represented as:

$$\text{ImV}(a, b, c, e) \leftarrow R(a, b, c, d), S(a, e), d > 100$$

where $\leftarrow$ is regarded as "if".

In LAV, we do not define the schema mapping as views over source databases. Rather, for each source database, one or more importing views are defined over the global schema. However, a user still raises queries over the global schema. These queries are answered by discovering all possible ways to construct the query using the views provided by the sources. This process relies on the approach, called *containment test of conjunctive query*. It will be described in the next chapter.

A conjunctive query can be translated into a query using relational algebra. For instance take the above query, the relational algebra query of it can be:

$$\text{ImV}(a, b, c, e) :- \pi_{a,b,c,e}(\sigma_{d>100}(R \bowtie_a S))$$

Conjunctive queries and relational algebra have identical powers of describing queries. However, in LAV, the former is chosen because of its containment test which needs to be applied. LAV and how to build queries using conjunctive query language are not the focus of this research. We introduce it because the importing views defined by conjunctive queries need to be represented in the meta-database that is introduced in the next section.

4.5 Meta-database

The aim of this work is to construct an architecture for database integration systems which is easy to maintain when evolution of the source database occurs. In order to maintain a system more easily, hard-coded programs of queries for integrating source databases need to be avoided as much as possible. The maintenance of hard-coded programs is very complex and time-consuming, as they are difficult to comprehend. In this architecture, however, all the queries are stored as structured data in a meta-database which are easy for evolution detectors or humans to comprehend and modify. We choose relational algebra as the language to define exporting views as it is easily stored as structured data instead of hard-coded programs. This section introduces one of the core concepts of the research, Meta-database. The meta-database is a conceptual database in which we store all the data required to conduct RSMV.

4.5.1 Meta-data Representation in Meta-database

In order to describe the schema evolution detection clearly, the data that are stored in the meta-database need to be represented and termed formally together with their relevant concepts such as attributes and relational algebra. Metadata is the data that describes other data. Therefore, the information we store in the meta-database is all descriptive information (i.e. relation schema and expression description), not the content of the relations or the results of the expressions themselves.

4.5.1.1 Attributes and Relations Representation

In the relational model, an attribute has two properties: *name* and *domain*. Therefore, an attribute is a pair A (A, T) where A is the name and T is the type or domain of the attribute.

In the meta-database, the name and the type are represented as string:

Name: String

Type: String

An attribute can then be regarded as a pair:

Attribute: (*name***: String,** *type***: String)**

A relation may be a *base relation* or a *view* in the meta-database. In fact, both a base relation and a view in the actual local database schema are referred to as a *base relation* in the meta-database, because they are created by the local database administrators independently of the integrated system. A *view* in the meta-database is the *atomic view* described previously, derived from other relations or views. In the meta-database, base relation and view are both represented as *relation schema* involving three properties: *name* and *relation type* and *attribute list*. The *name* is a string that identifies the relation schema. The *relation type* is also a string that indicates whether the schema is a relation schema or a view schema. The *attribute list* is the list of attributes of the relation schema. Therefore, a *relation schema* representing a relation or a view can then be represented as a triple:

Relation Schema: (n: Name, t: Relation Type, list: {A| A: Attribute})

The Relation Type of a relation is:

Relation Type: String

∀X: Relation Type, X ∈ {“Relation”, “View” }

If a relation schema represents a view, it also has an algebraic operator expression which will be represented in a subsequent section.

4.5.1.2 Relational Algebra Operation Representation

As a *view* involves an *expression* of a relational algebra operation, a relational algebra operation needs to be defined formally. An *operation* is one of the following relation algebra operations:

Operation: Join | Selection | Projection | Grouping | Union | Difference | Cartesian Product | Intersection

Each operation has a different representation of the expression and is defined individually. A *view expression* of a view is an expression that consists of one of the above operations. A view expression has two properties: *operation type* and *operation*. The *operation type* is a string that indicates which one of the operations is used in the view expression. The *operation* is the actual operation consisting of the view expression. Therefore, view expression can be regarded as a pair:

VE: (t: Operation Type, o: Operation)

The *Operation Type* is defined as a string:

Operation Type: String

∀X: Operation Type, X ∈ **{"Join", "Selection", "Projection", "Grouping", "Union", "Difference", "Cartesian Product", "Intersection"}**

.

4.5.1.2.1 Common Join

The operation *common join* can be regarded as a triple:

Join: (r1: Relation Schema, r2: Relation Schema, {C| C: Attribute})

The first two properties are two relation schemas participating in the operation, termed *operand relation schema*, while the third is a list (termed *common attribute list*) of attributes (termed *common attribute*) on which these relations are joined. Each operation listed above has one or two operand relation schemas. Two constraints must be followed by join operation:

1) The common attributes must be the attributes of both operand relation schemas.
2) A join operation must have two operand relation schemas.
3) There must be at least one common attribute in the common attribute list.

The representation can be extended to capture the constraint.

Join: (r1(a, b, A): Relation, r2(c, d, B): Relatoin, C:{Ci | Ci: Attribute) |

$C \subseteq A \wedge C \subseteq B \wedge C$ is not Ø

4.5.1.2.2 Selection

The operator *selection* has one operand relation schema and a *condition*, and can be regarded as a pair:

Selection: (r: Relation Schema, c: Condition)

The first property is the *operand relation schema*. A *condition* is the condition under which the tuples of the operand relation will be evaluated. The condition consists of two operands and a predicate operator:

Condition: (a: Operand, p: Predicate, b: Operand)

An *operand* is either an attribute or a constant which is taken by the predicate operator. An *operand* consists of two properties: *class* and *content*. The *class* refers to an attribute or a constant. The *content* is the actual value written in the expression, which is either an attribute or the value of the constant. The class and operand are formally defined as:

Class: String

$\forall X$: Class, $X \in$ {"Attribute", "Constant"}

Operand: (c: Class, x: Content)

The *content*, which is the actual value of the operand, is either an *attribute* when the class is "Attribute" or a *constant* when the class is "Constant". The *content* is defined as:

Content: Attribute | Constant

The *constant* consists of a *value* and a *type*. The *value* is a string which is the actual value of the constant, while the *type* is same as the type of an attribute. The *constant*

is represented as:

Constant: (v: String, t: Type)

The *predicate* is a string representing the predicate operator applied to the *condition*.

Predicate: String

∀X**: Predicate,** X ∈ **{“>”, “<”,“≤”,“≥”,“=”}**

The set of operators can certainly be extended to involve other operators, in practice.

There are two constraints on the *condition*:

1) If the class of an operand is “Attribute”, then the content that is an attribute must be an attribute of the relation *r* of the *selection*.
2) The domains of the two operands must be the same.
3) A condition must have two operands.

The type of an attribute or a constant can be denoted as: ***T(a), a ∈ {Attribute, Constant}***. Therefore the representation of *selection* is extended as:

Selection: ((n, t, A): Relation, ((c, a): Operand, p: Predicate, (d, b): Operand): Condition) | (c = “Attribute”⟶ a∈A), (d = “Attribute”⟶ b∈A), T(a) = T(b)

4.5.1.2.3 Set Operators

The operations *Union* and *Difference* and *Intersection* have the similar formats and therefore are described together. These operations all have two operand relation schemas. They are represented as:

Union: ((n, t, A): Relation, (n, p, B): Relation) | A = B

Difference: ((n, t, A): Relation, (n, p, B): Relation) | A = B

Intersection: ((n, t, A): Relation, (n, p, B): Relation) | A = B

One constraint is defined from the above three operations:

Two operand relation schemas must have an identical set of attributes.

4.5.1.2.4 Cartesian Product

The operation Cartesian Product is represented as:

Cartesian Product: (r1: Relation, r2: Relation)

One constraint is defined for this operation:

A Cartesian Product operation must have two operand relation schemas.

4.5.1.2.5 Projection

The *projection* operator, consisting of a *relation* and a *projection list*, is represented as:

Projection: (r: relation, L: Projection List)

As introduced previously, the *projection list* is a set of projection elements that can be represented as:

Projection List: {element | element: (s: Source, a: Attribute)}

The *projection element* that is a pair of the list is an expression which combines or calculates attributes or constants, and assigns the result to a new attribute. The second property of the projection element, termed *output attribute*, which is an attribute, is the new attribute that accepts the result. The first property, termed *source*, refers to the attribute or constant or expression that produces a result for a new attribute.

The source has two properties: *source type* and *source value*. The source type is a string that indicates the type of the source that may be “Attribute”, “Constant” and “Expression”. The source value is the actual attribute or constant or expression.

Source: (t: Source Type, v: Source Value)

The Source Type is defined as a string as follows.

Source Type: String

∀X: Source Type, X ∈ **{“Attribute”,“Constant”,“Expression”}**

The Source Value is represented as:

Source Value: Attribute | Constant | Expression

An *expression* consists of *operand list* and an *operator*. The *operand list* is a list of operands that were defined previously. The *operator* is an arithmetic or string operator and is represented as:

Operator: String

∀X: Operator, X ∈ **{“+”, “-”, “*”, “/”, “||”}**

The set of operators can be extended in practice.

The expression is then represented as:

Expression: (o: Operator, operand list: {a| a: Operand})

There are three constraints on the above *projection* operator:

1) If the source is an attribute, then it must be an attribute of operand relation schema r and the domain of it must be identical to the domain of the output attribute. It is formally represented as:

   Projection: (r (n, t, A): Relation, {((“Attribute”, v): Source, a: Attribute)}: List) | v ∈A, T(a) = T(v)

2) If the source is a constant, then the domain of the constant must be identical to the domain of the output attribute *a* to which the constant is assigned. It is formally represented as:

   Projection: (r (n, t, A): Relation, {((“Constant”, v): Source, a: Attribute)}: List) | T(a) = T(v)

3) If the source is an expression, then all the operands that are attributes in the operand list expression must be attributes of the operand relation schema r and the domains of attributes and the constants must be identical to each other and to

the type of the corresponding output attributes.

Projection: (r (n, t, A): Relation, {(("Expression", (o, {(Ci, Ei)}:l): Source, a: Attribute)}: List) | Ci = "Attribute" → Ei ∈A, T(E1) = T(E2) =…= T(En)

Description: if there is an operand in the operand list whose type is e, then for all the operands in the list, the domains of them must be e.

4.5.1.2.6 Grouping

The grouping operation can be regarded as:

Grouping: (r: relation, g: {Ai | Ai: Attribute}, l: {e | e: (a: Aggregation, b: Attribute)})

As introduced in Chapter 3, the operand relation schema indicates the relation on which the grouping operation is applied. The second property is a list (termed *grouping attribute list*) of attributes (termed *grouping attribute*) by which the relation will be grouped. The third property of the grouping operation is a list of pairs, called *Aggregation List*, to average or aggregate an attribute and put a new attribute name on the result. The pairs in the list are termed *aggregation element*. The *aggregation* of an aggregation element in the aggregation list represents aggregation operator to aggregate an attribute, while the second element of the aggregation element is termed *resulting attribute* accepting the result of the aggregation operator.

The aggregation can be represented as:

Aggregation: (ao: Aggregation Operator, a: Attribute)

The *aggregation operator* refers to the type of the operator applied in this aggregation. The second property that is an attribute, termed *operand attribute*, refers to the attribute on which the operator is applied. The *aggregation operator* is represented as:

Aggregation Operator: String

∀X**: Aggregation Operator,** X ∈ {**“**MIN**”, “**MAX**”, “**SUM**”, “**COUNT**”, “**AVG**”}**

There are two constraints on the grouping operation:

1) In the grouping operation, every attribute that appears in the grouping attribute list g and appears in an aggregation must be an attribute of the operand relation schema r.
2) In each aggregation element of the aggregation list, the domain of the aggregation operand of the aggregation element must be identical to the domain of the resulting attribute paired to it.

The grouping operator is then extended as:

Grouping: ((n, t, A): relation, g: {Ai | Ai: Attribute}, l: {e | e: ((a, c): Aggregation, b: Attribute)}) | (∀Ai, Ai ∈g, Ai ∈A), (∀e ((a,c), b), e ∈l, c ∈A), T(c) = T(a)

4.5.1.3 Exporting views

An exporting view consists of a tree of atomic view schemas and relation schemas. The schema of an atomic view has been defined previously. However, a complete atomic view also has a view expression of relational algebra operations to define it. The *view expression* has been defined previously:

VE: (t: Operation Type, o: Operation)

Thus, the entire temporary view is regarded as a pair:

Atomic View: ((n, t, A): Relation, v: VE) | t = “View”

The first property is the *view schema* to be taken by another atomic view, while the second property is the *view expression*. However, there is as yet no relationship between the attributes of the schema of the atomic view and the attributes of operand relation schemas in the expression. Therefore, the following rules are defined in order

to establish matching between the schema of the view and the output relation schema of the expression. Let V (r(n, rt, A), v(ot, o)) be an atomic view,

1) If the operation type ot is "***Join"*** and o is represented as o(r1, r2, C), let B be a set of all attributes of r1 and D be a set of all attributes of r2, then the attributes A of the view schema r of the atomic view V is the union of B and D, denoted A = B ∪ D.
2) If the operator type ot is "***Selection"*** and o is represented as o(r1, C), let B be a set of all attributes of r1, then A is equivalent to B, denoted A = B.
3) If the operator type ot is "***Union"*** or ***"Difference"*** or ***"Intersection"*** and o is represented as o(r1, r2, {m | m: (Ai, Bi)}), Let B be a set of all attributes of r1 and C be a set of all attributes of r2, then A is equivalent to B and C, denoted A = B = C.
4) If the operator type ot is "***Cartesian Product"*** and o is represented as o(r1, r2), Let B be a set of all attributes of r1 and C be a set of all attributes of r2, then A is the union of B and C, denoted A = B ∪ D.
5) If the operator type ot is "***Projection"*** and o is represented as o(r, L{(Si, Bi)}), Let C be a set of attributes such that for each (Si, Bi), (Si, Bi)∈L, Bi ∈C and for each Ci, Ci ∈ C, there is a (Si, Bi), (Si, Bi)∈L and Bi = Ci, then A is equivalent to C, denoted A = C.
6) If the operator type ot is "***Grouping"*** and o is represented as o(r1, B, L{(Ai, Ci)}), Let D be a set of attributes such that for each (Ai, Ci), (Ai, Ci)∈L, Ci ∈D and for each Di, Di ∈ D, there is a (Ai, Ci), (Ai, Ci)∈L and Ci = Di, then A is union of B and D.

The above rules need to be followed not only to represent atomic views in the meta-database but also to conduct the automatic view modification that will be introduced in Chapter 6.

Recall that an exporting view was defined as tree. The exporting view tree is now extended to represent an exporting view in the meta-database as:

Exporting View: (v: {t | t: Atomic View}, r: Atomic View, l: {(n, t, A) | (n, p, A): Relation}) | p = "Relation" | r ∈ v

The first property is a list of all atomic views, termed *atomic view list*, involving the atomic view that is the final answer. The second property is the atomic view that is the final answer (the root of the tree), termed *root view*. The third property is the set of all base relation schemas which is in the Local Schema, termed *leaf list*. The relation schemas in the leaf list are termed *leaf relation schema*.

The constraints of the representation of an exporting view are:

1) There must be at least one leaf relation schema in the leaf list of *ExVi*. (*leaf list* is not Ø)
2) The root atomic view must be also in the atomic view list.
3) There is an atomic view V in the atomic view list of *ExVi* such that the operation type of view expression is not "Projection" and there is an operand relation schema R of V such that the relation type of R is "Relation".
4) An atomic view in the atomic view list of *ExVi* has two parent atomic views in the atomic view list of *ExVi*. Namely, there are two parent views such that they have the same atomic view as one of their operand relation schemas.

4.5.1.4 Importing Views

Recall that the conjunctive query of an importing view is:

ImV (a, b, c, e) ← R (a, b, c, d), S (a, e), d > 100

As with an exporting view, an importing view has a *view schema* that is shown on the left side of the "if" symbol and an expression on the right side of the "if" symbol. The expression consists of one or more *subgoal relation schemas* and one or more *conditions*. The importing view is then represented as:

ImportingView: ((n, t, A): Relation, relations: {(r, c, B) | (r, c, B): Relation}, conditions: {c | c: Condition)} | (1) t = "View", (2) (∀x(r, c, B), x

∈relations, c = “Relation”), (3) (∀y((a, b), o, (e, f)): Condition, y ∈conditions, a = “Relation” → (∃z(g, h, C):Relation, z ∈ relations, b ∈C), e = “Relation” → (∃z(g, h, C):Relation, z ∈ relations, f ∈C)), (4) T(b) = T(f)

The four constraints on the importing view are:

1) The type t of the view schema (n, t, A) of the importing view must be “View”.
2) For each subgoal relation schema x (r, c, B), x ∈ *relations* list, the c of x must be “Relation”. Namely, each subgoal relation schema in the subgoal relation schema list must be a base relation that is in the Global Schema.
3) For each condition y((a, b), o, (e, f)), y ∈conditions, if the class a of the first operand (a, b) is “Attribute”, then there must be a subgoal relation schema z (g, h, C), z ∈ relations, such that b ∈ C, and for each condition y((a, b), o, (e, f)), y ∈conditions, if the class e of the second operand (e, f) is “Attribute”, then there must be a subgoal relation schema z (g, h, C), z ∈ relations, such that f ∈ C. Namely, for each operand of the condition, if the operand is an attribute, then it must appear in one or more subgoal relation schemas in the subgoal relation schema list.
4) The domains for the attributes or constants of the two operands of a condition must be identical.

4.5.1.5 Global and Local Schema

The global schema can be regarded as a set of all relation schemas in it:

GS: {Ri | Ri: Relation}

The global attribute domain is then represented as:

GAD: {Ai | Ai: Attribute}

The local schema can be regarded as a pair, its *name* that is a string and a set of all relation schemas in it:

LS: (name: String, r: {Ri | Ri: Relation})

As an integration system can have more than one local schema, the set of all local

schemas of it termed *local schema list* that is represented as:

LSL: {LSi | LSi : LS}

4.5.1.6 Mappings

The *mapping* from a local schema to the global schema is represented as:

Mapping: (name: String, list: {(ExVi: ExportingView, ImVi: ImportingView)}

The mapping has two properties: *name* and *map list*. The *name* is a string that indicates which local schema is integrated into the global schema by the mapping. The name of the mapping is the same as the name of the local schema from which it maps to the global schema. The *map list* is a list of pairs of exporting views and importing views which represent the relationship between the local schema and the global schema. A pair of exporting views and importing views is termed a *map*.

The mappings from all local schemas to the global schema are termed *entire mapping list* and are then represented as:

MPS: {m | m: Mapping}

Note that for each local schema in the local schema list, there is only one mapping in the entire mapping list such that the name of the mapping is identical to the name of that local schema.

4.5.1.7 Organizational Structure

As introduced previously, the local databases may be grouped into a hierarchical structure by categorization properties (CP). In order to store the organizational structure tree in a meta-database, the new entity, called *organizational property*, is defined as:

Organizational Property: (name: String, type: String) | type ∈ **{“**Categorization Property**”, “**Local Schema**”}**

The *organizational property* consists of a *name* and a *type* which are both strings. The *name* of it is the name of the property, while the *type* indicates if the property represents a local schema or a property. If a property represents a local database, the name is the name of the local schema that must be in LSL. An organizational structure tree by a property of the local database (i.e. Location) is represented as:

Organization: (n: Name, t: Tree)

Tree: (nodes: {(n, t) | (n, t): Organizational Property }, root: Organizational Property, r: {a | a: Parent}) | ∀x (n, t): Organizational Property, x ∈nodes ∧ t = "Local Database" → (∃y (a, r): LS, y ∈LSS, n = a)

Parent: (a: Property, b: Property)

The *name* of the organization refers to which kind of property (i.e. "Location") the local databases are grouped. The *tree* refers to the organizational tree. The first property of the tree is a set of all organizational properties on the nodes of tree, termed *node list*. The organizational property on the root, termed *root property*, is referred to as the second property of the tree. The third property is a list of the pairs, called *parent relation*, which refers to the parent-child relationship between two organizational properties on the tree. In the parent relation, the first property is referred as to the parent (termed *parent property*) of the second property (termed *child property*).

The list of all organizational structure trees is termed *organization list* and is represented as:

Organization List: {o | o: Organization}

4.5.2 Representation of the Meta-database

After the representations of all the data are presented, the formal representation of the meta-database itself can be presented. The meta-database is represented as a set of all

the data defined above:

MD: {*lsl*: LSL, *gs*: GS, *domain*: GAD, *m*: MPS, *org*: Organization List}

It can be seen that the meta-database consists of a *local schema list* and the *global schema* and the *global attribute domain* and more importantly the *entire mapping list* and *organization list*. The meta-database will be extended to store other information in the remaining chapters of this thesis.

4.6 Summary

To sum up, this chapter introduces the algorithm, called RSMV, to integrate local schemas to the global schema. It makes each local schema homogeneous to the global schema by building views on it. It then integrates the local schema to the global schema by building a mapping between importing views over the global schema and the exporting views over the local schema. As such, the query processing can be conducted based on the mappings. More importantly, all the schemas of relations and views together with the expressions of the views are represented and stored in a meta-database which is a conceptual database. Consequently, there is no hard-coded program of queries to deal with the schema integration. This allows tools to search and modify the mappings automatically with little human intervention so that the cost of maintenance caused by the evolution of the database schema is minimized as much as possible. The query processing and the approach to modifying the mappings are introduced in subsequent chapters. Moreover, some other descriptive data such as URLs of a database service may be added into the meta-database in practice, which will be discussed in chapter 7.

Chapter 5 Schema Evolution Detection

5.1 Introduction

Chapter 4 introduced the approach to integrating source databases into the global schema with heterogeneity eliminated and representation of the data in the meta-database. The algorithm introduced in chapter 4 can be thought of as the preparation for the algorithm presented in this chapter. This chapter introduces the algorithm which identifies the affected views by evolution in the source databases, and then automatically maintains the system by modifying the view definitions stored in the meta-database.

Firstly it is explained how each database evolution can affect views. Rules are then introduced to identify and modify the affected views. Based on the rules, two processes of Schema Evolution Detection, Identification of Affected Views and Automatic View Modification, are described in detail. It is also shown that sometimes the views must be discarded following certain types of evolution.

5.2 Overview of Schema Evolution Detection

In a traditional software lifecycle, software maintenance is an important part accounting for at least 50 percent of the total lifetime cost of a software system [101]. Among the seven phases of software maintenance process defined by IEEE [102], design and implementation together with software comprehension require much more effort from maintenance programmers; understanding and modifying the existing programs is complex and time-consuming work.

A database integration system requires even more maintenance, because there is an additional factor leading to the maintenance of the system: database evolution. In a traditional integrated database system, a large amount of hard-coded queries over the local schemas exist in order to both integrate source databases and provide results to

users. Therefore, a change in a source database may lead to a large amount of work in modifying existing queries. Consequently, the system may become impossible to maintain if the number of the source databases involved become huge.

In our architecture, as there are no hard-coded queries directly over local schemas, the work caused by database evolutions is to understand and modify the data stored in meta-database. As the view definitions are represented as structured data in the meta-database, they are easier to understand by both humans and machines. Thus, a software tool or a function can be produced to help maintenance programmers to modify the existing views. Schema Evolution Detection is an algorithm that can be used by a software tool to modify the existing views based on some rules. There are two general processes that are undertaken in Schema Evolution Detection:

1) **Identification of Affected Views**: This process searches all views relevant to the evolved source database in order to find all the views affected by the evolution and therefore requiring modifications.
2) **Modification of Views**: This process modifies the affected views based on previously defined rules. In some cases, this process may require human interventions.

These two processes are actually stimulating the real activities taken by maintenance programmers when maintaining the system manually. When an evolution occurs, programmers need to find which programs are affected by the evolution based on their knowledge. Having found the affected views, the programmers are then able to modify these views in order for the system to work properly. This is again based on their knowledge. Although the knowledge is held by individual programmers, most of it is common knowledge and can therefore be defined as rules. The human activities of maintaining the system can then be undertaken by software tools.

5.3 Identification of Affected Views

This section introduces the algorithm of identifying affected views by database evolution based on rules. When an evolution occurs, either a programmer or a software tool needs to find which parts of the system are affected and require modification. In this research, as an evolution defined previously only has some impacts on the data stored meta-database, all the programmer or a software tool needs to do is to search in the meta-database. Figure 5-1 shows the process of Identification of Affected Views.

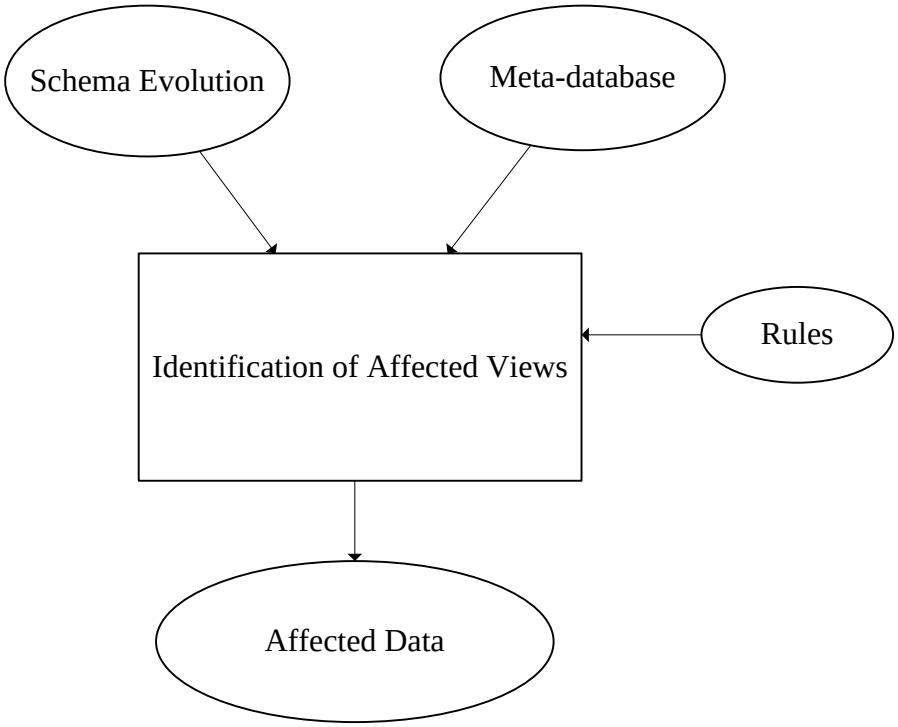

Figure 5-1 Identification of Affected Views

As shown in Figure 5-1, when a schema evolution occurs, the process of Identification of Affected Views takes the evolution and the meta-database as its inputs and produces a result which is a set of the affected exporting and importing views based on the pre-defined rules. The evolution and the rules and the affected data will be described in detail in the following sections.

5.3.1 Categorizations of Evolution

The evolution considered in this work can be generally categorized into three types:

- **Schema Evolution**: This refers to the evolution in local schemas of the source databases. The schema of relations and the schema of the source databases themselves may evolve over time.
- **Organizational Evolution**: This refers to the evolution in the organizational hierarchical structure of the source databases.
- **System Evolution**: This refers to the evolution in the descriptive information of the source databases or the services. For example, the name and URL of a source database, which is used for programs to access the source database, may change.

Schema evolution and organizational evolution have caused more maintenance costs in previous data integration projects and are the central issues tackled by RSMV and meta-database and Schema Evolution Detection. However, organizational evolutions do not have any impact on the view definitions in the meta-database. The impact of organizational evolutions is mainly on queries raised by end users. Therefore, the aim of Schema Evolution Detection in this chapter is to automatically tackle schema evolution. Therefore, the schema evolution is formally described in this chapter, the algorithm of resolving organizational evolution will be described in the next chapter introducing query processing.

System evolution can only lead to a tiny amount of work in our architecture due to the flexibility of the service-oriented architecture. Therefore, system evolution is not tackled by Schema Evolution Detection described in this chapter and will be described together with the solution to tackle it in chapter 7.

5.3.2 Schema Evolution

Schema evolution occurs frequently and brings a large amount of work on modification of hard-coded programs in traditional database integration systems. In

this work, the impact of schema evolution is on view definitions represented in a meta-database, as there are no hard-coded queries. Each evolution can be referred to as an operation applied on a source database. There is no material presenting detailed illustration of possible evolution for the time being. This working defines the possible evolution by extending the set manipulations provided for database administrators by relational database management systems. This section presents a descriptive definition of various schema evolution, while they are formally described using the data representation in the meta-database. The schema evolution can be further divided into three levels in this research:

- **Attribute Level Evolution**: This involves the changes in attributes. It means that an attribute of a relation may be added, removed, or given a new name. Moreover, the domain of an attribute may be changed.
- **Relation Level Evolution**: This involves the changes in relations. It means that a relation schema in the local schema may be added, removed or given a new name. This type of evolution is even more complex because a relation schema may also be decomposed into two or more relation schemas while two or more relation schemas may be merged into one relation.
- **Database Level Evolution**: This involves the changes in databases. It means that a new source database may be integrated into the system while an existing one may become unavailable.

5.3.2.1 Attribute Level Evolution

The following five evolutions are involved at this level:

1) **Attribute Addition**: A new attribute is added into a relation schema.
2) **Attribute Removal**: An existing attribute is removed from a relation schema.
3) **Attribute Rename**: The name of an attribute of a relation schema has changed.
4) **Attribute Domain Change**: The domain of an attribute of a relation has changed.
5) **Attribute Decomposition**: An attribute is partitioned into two or more attributes.

5.3.2.2 Relation Level Evolution

The following four evolutions are involved at this level:

6) **Relation Addition**: A new relation is added into a local schema.
7) **Relation Removal**: An existing relation is removed from a local schema.
8) **Relation Rename**: The name of a relation of a local schema has changed.
9) **Relation Decomposition**: A relation is partitioned into two or more relations.

In fact, relation decomposition is the combination of removal and addition of relations. It consists of two operations that are:

1. Removing a relation from a local schema and then
2. Adding one or more relations into the local schema.

However, relation decomposition is described explicitly in this work for the following reasons:

- It usually happens that a relation is removed because the database maintainer is going to decompose the relation so that the database can be in higher normal form.
- Removing a relation may lead to discarding some views.
- Adding new relations in local schema and building views on them have to be done manually.
- Replacing a relation with other relations in a view definition can be done automatically by software tools in this work.

Therefore, we believe that tackling this evolution automatically can reduce the work of system maintenance.

5.3.2.3 Database Level Evolution

The following two evolutions are involved at this level:

10) **Database Addition**: A new local schema becomes available to be integrated into the integrated system.

11) **Database Removal**: An existing relation is removed from a local schema.

The rename of a source database is referred to as a system evolution and therefore is described in chapter 7.

5.3.3 Evolution Impact on the Integrated System

Although various schema evolutions have been listed in the last section, some of them may have little or no impact on the integrated system so that they require no automatic maintenance of the existing system. Some of them can be tackled manually by a human requiring little work. This section discusses the impact of this evolution on the integrated system. It also indicates which schema evolutions require automatic maintenance by software processors.

5.3.3.1 Schema Evolution Having No Impact

Three evolutions are referred as to this type:

- Attribute Addition
- Relation Addition
- Database Addition

Adding new attributes, relations or source databases does not have any impact on the existing system, because they did not exist when building the system and the data in the meta-database. It does not require any automatic modification by a software processor. Therefore, human maintenance programmers need to integrate them into the integrated system manually. In the evaluation chapter, it will be shown that these manual works are not complex in the architecture of this work.

5.3.3.2 Schema Evolution Having Impact

Eight evolutions are referred as to this type:

- Attribute Removal
- Relation Removal
- Attribute Rename

- Attribute Domain Change
- Relation Rename
- Attribute Decomposition
- Relation Decomposition
- Database Removal

As views are all defined in terms of relations and attributes, removal and change of attributes and relations will have an impact on all the views whose definitions involve the removed attributes or relations. When a schema evolution occurs, it means that the relation schema in the local schema has changed. However, the existing views are defined on the relation schemas before the evolution. Namely, the relation schemas and the attributes in the view definition become inconsistent with the corresponding relation schemas and attributes in the local schema. Consequently, these views become invalid and cannot work properly so that the queries on these views cannot work properly any more. The removal of a source database has an impact on all its exporting and importing views as well as the organizational structure trees.

5.3.4 Representation of Evolutions in Meta-database

Various schema evolution has been described in the last section. Although the representations of this evolution are easy to understand by humans, they still need to be understood by software processors. This section represents this evolution further as structured data in the meta-database. As the aim of representing the schema evolution is for software tools to modify existing views, only the schema evolution having impact are represented formally in the meta-database.

A schema evolution can be referred to as an operation which applies on the schema in the meta-database. In order to indicate that an evolution must be of one of the schema evolutions introduced previously, a new data type is defined which is *Schema Evolution Operation*:

Schema Evolution Operation: Attribute Remove | Attribute Rename |

Attribute Domain Change | Attribute Decomposition | Relation Removal | Relation Rename | Relation Decomposition | Database Removal

The representation of each schema evolution operation is described as follows:

1) **Attribute Remove**: This is represented as:

   Attribute Remove: (*r****: Relation,*** *a****: Attribute)***

   It has two properties: *original relation schema* and *original attribute*. The first property, *original relation schema,* is the relation schema whose attribute is removed, while the second property, *original attribute*, is the attribute that is removed.

2) **Attribute Rename**: This is represented as:

   Attribute Rename: (*r****: Relation,*** *original****: Attribute,*** *evolved****: Attribute)***

   In addition to *original relation schema* and *original attribute*, it has the third property, *evolved attribute*, which is the resulting attribute. The original attribute is renamed to become the evolved attribute.

3) **Attribute Domain Change**: This has the same properties to attribute rename evolution. It is represented as:

   Attribute Domain Change: (*r****: Relation,*** *original****: Attribute,*** *evolved****: Attribute)***

   The domain of original attribute is changed to the domain of the evolved attribute.

4) **Attribute Decomposition**: This is represented as:

   Attribute Decomposition: (*r****: Relation,*** *original****: Attribute,*** *list****: {Bi | Bi: Attribute},*** *operator****: Operator)***

   It also has an *original relation schema* and an *original attribute* as the first and the second properties. The third property of attribute decomposition is a list of evolved attributes that is derived from the decomposition of the original attribute,

termed *evolved attribute list*. The fourth property is an *operator* by which the attributes in the evolved attribute list can be composed to produce the original attribute. Any attribute decomposition that cannot be composed by an operator to produce the original attribute is not considered as attribute decomposition in this research.

5) **Relation Removal**: This is represented as:

Relation Removal: (*r****: Relation)***

It only has one property: *original relation schema* that is the relation schema removed.

6) **Relation Rename**: This is represented as:

Relation Rename: (*r****: Relation,*** *r'****: Relation)***

The first relation is the *original relation schema* before the change. The second relation is the *evolved relation* after change.

7) **Relation Decomposition**: This is represented as:

Relation Decomposition: (*r****: Relation,*** *relations****: {****Ri* **|** *Ri****: Relation},*** *list****: {****Ai* **|** *Ai****: Attribute})***

The first element is the *original relation schema* to be decomposed. The second element is a list of evolved relation schemas, termed the *evolved relation schema list*. The evolved relation schemas are derived from the decomposition of the original relation schema. The third element is a list of common attributes, termed *common attribute list*. All the evolved relation schemas can be composed by a join operation on these common attributes.

8) **Database Removal**: This is represented as:

Database Removal: (*original****: LS)***

Database removal has one element that is the local schema (*original local schema*) that has been removed in this evolution.

Note that each schema evolution operation has one and only one original relation schema.

A *schema evolution* can then be represented as:

Schema Evolution: (*ls-name****: String,*** *type****: SchemaEvolutionType,*** *evolution****: Schema Evolution Operation)***

The first property is *local schema name*, a string, which indicates the local schema where the evolution occurred. The second property is *schema evolution type*, a string, which indicates which schema evolution operation applied.

SchemaEvolutionType: String

∀*X****: SchemaEvolutionType,*** *X* ***∈ {"Attribute Remove", "Attribute Rename", "Attribute Domain Change", "Attribute Decomposition", "Attribute Composition", "Relation Removal","Relation Rename", "Relation Decomposition", "Database Removal"}***

The third element of schema evolution is the schema evolution operation defined above.

All the schema evolution can then be represented as a *schema evolution list*, represented as:

SEL: {se: Schema Evolution}

5.3.5 Process of Identification of Affected Views

Having represented the schema evolution, the process of Identification of Affected Views of a schema evolution can be described as a function:

IAW: (SE, MPS) → Affected Map List

Affected Map List: {(*ExVi****: ExportingView,*** *ImVi****: ImportingView)}***

The process takes a schema evolution and the entire mapping list as its inputs, and produces a map list that involves the maps in which the exporting view (termed affected view) is affected by the schema evolution. The map is then termed *affected map*.

5.3.5.1 Affection Rule

In order to conduct the Identification of Affected Views, the rule to define an *affected view* is defined as follows:

- Given a schema evolution *se* on local schema *ls*, if the type of *se* is not "Database Removal", let ExVi be an exporting view of a map M of ls and R be the leaf list of ExVi and r be the original relation schema of se, then ExVi is an ***affected view*** of se if r $\in$ R.

The map that has an affected view is called an affected map. The *affected map list* is then a list of affected maps of a local schema. The above rule does not consider the schema evolution *database removal*, as database removal require the complete mapping of the local schema to be removed, and therefore does not undertake the process of identification of affected views.

5.3.5.2 Process of Identification of Affected Views

When a schema evolution is applied in a local schema, the following steps are taken as the process of identifying the affected views:

1) Let *se* be the schema evolution and *r* be the original relation schema of se, within the entire mapping list MPS, find the mapping M whose name is the same as the local schema name of se.
2) For each map mi of the map list of M, if there is a leaf relation schema Ri of the exporting view such that r = Ri, then store mi into the affected map list *AML*.

5.4 Automatic View Modification

Having obtained the affected map list, the automatic modification of the affected views can be undertaken. In this research, each schema evolution must be tackled individually and immediately before the next schema evolution takes place. The automatic view modification can be referred to as an operation to modify the affected map list according to the schema evolution:

AVM: (SE, Affected Map List)

Although it is not presented in the operation, the modification is still based on some rules. Generally, the aim of the automatic modification of the affected views is to make the relation schemas and the attributes in the affected views consistent again with their corresponding relation schemas and the attributes in the local schema in the meta-database. By doing so, the affected views can become valid and work properly again. The views that cannot become valid any more must be discarded by the process so that they are not considered by the integrated system any more.

Assumption

There are three assumptions made as follows:

1. Before the schema evolution, all the views (exporting views and importing views) are all syntactically valid and can work properly.
2. Modifying the affected map list will result in the update of the corresponding data in the meta-database immediately.
3. When a schema evolution occurs, the corresponding relation schema or local schema in the local schema list in the meta-database has been changed by the evolution before Schema Evolution Detection.

5.4.1 Equality Rules

In order to describe the algorithm precisely, some rules must be defined to describe the equality between two attributes and two relations. In addition, the association

between the local schema and its mapping is defined.

Attribute Equality: Two attributes are said to be equivalent if they have the same name and the same type. Let A(a, t1) and B(b, t2) be two attributes, A equals B, denoted A = B, if a = b and t1 = t2.

Relation Equality: Two relation schemas are said to be equivalent if they have the same name and the same type and the same set of attributes. Let R(r, t1, A) and S(s, t2, B) be two relation schemas, R equals S, denoted R = S, r = s and t1 = t2 and A = B.

Association between local schema and its mapping: A mapping is said to be the mapping of a local schema if the name of the mapping is the same as the name of the local schema. Let LS (n1, R) be a local schema and M (n2, list) is a mapping, M is the mapping of LS or M is LS's mapping, if n1 = n2.

5.4.2 Discard Rules

In some cases, the exporting views cannot be automatically modified and may need to be discarded, because the removal of some of an attribute or a relation schema can lead to the loss of semantic meaning of the views. By *discard* it means that the views are removed from the meta-database and then will not be used by the integrated system. We use the term *discard* rather than *remove*, because in practice a view may not have to be removed from the meta-database. It is just made unavailable so that the query processor and other software components in this architecture will not consider it until it is modified and made available again. The *discard rules* are defined, in this research, to examine in which cases the exporting views need to be discarded.

5.4.2.1 Validation Rules

The term *valid atomic view* must be defined before defining the discard rules. Recall that, in section 4.6, some constraints are defined for each operation of the view expression of an atomic view. An operation is said to be *valid* if all the constraints on

this operation are followed. An atomic view is said to be a *valid atomic view* if the operation of its view expression is valid. An exporting view is valid if all the constraints on the exporting view are followed.

5.4.2.2 Discard Rules for Atomic Views

An atomic view *Vi (relation, view expression)* should be discarded if one of the following rules is true:

a) If the operation of the view expression of the atomic view *Vi* is not valid (invalid).

5.4.2.3 Discard Rule for Exporting Views

An exporting view *ExVi (atomic view list, root, leaf list)* should be discarded if one of the following conditions is true:

a) The exporting is not valid (invalid).

5.4.3 Process of Automatic View Modification

Generally, the process of the automatic view modification is firstly to apply the schema evolution, which was applied on local schema, on corresponding atomic views of an exporting view in order to keep the relation schema and its attributes in the atomic views consistent with the actual relation schema in the local schema, and then examine whether the atomic views should be discarded. If the atomic views are not discarded, it means that the atomic views can work properly. Once all corresponding atomic views are processed, it examines whether the exporting view should be discarded. If so, the corresponding importing view should be discarded as well as the map in the affected list.

Once a schema evolution has been applied and the affected map list has been output by the process of Identification of Affected Views, the automatic view modification takes the following steps to modify the affected map list. Let *se* be the schema evolution and *AML* be the affected mapping list.

5.4.3.1 Process Tackling Attribute Rename

If the schema evolution type of *se* is "Attribute Rename", let *r* be the original relation schema of se and *A1* be the original attribute and *A2* be the evolved attribute, then for each map *Mi* in *AML*,

1) In the leaf list of the exporting view of Mi, attempt to find a leaf relation schema r' such that r' = r. If r' is found, then attempt to find the attribute A' in the attribute list of r' such that A' = A1, if A' is found, then substitute A2 for A'.
2) In the atomic view list of the exporting view of Mi, attempt to find the atomic view *v* such that r is identical to one of the operand relation schemas of the operation of the view expression of v. If v is found, then attempt to find the attribute A' in the attribute list of the operand relation schema of the view expression of v such that A' = A1, if A' is found, then substitute A2 for A'.
3) For each element in the projection list of the operation of the view expression of v: if the source type of the source of is "Attribute" and the source value is identical to A1, then substitute A2 for the source value; otherwise if the source type is "Expression", then for each *operand* of the source value, if the operand is an attribute and the content of the operand is identical to A1, then substitute A2 for the content of the operand.
4) Check whether the exporting view of Mi should be discarded based on discard rules. If any rule is true, discard Mi and repeat 1 for next map.

The attribute rename evolution only has an impact on the atomic view that has the original relation schema as its operand schema, because as defined in chapter 4, the atomic view that is defined on a base relation schema must be the atomic view using the projection operation. Therefore, other atomic views of the exporting view will use this atomic view and its attribute instead of the base relation schema itself. Therefore, the process first modifies the leaf relation schema in the leaf list of the exporting view to keep it consistent with the actual relation schema. The process then modifies the atomic view with the projection operation to keep the operand relation schema

consistent with the actual relation schema in the local schema. Next, if the renamed attribute is used by the projection operation in its projection list change it to keep it consistent. Finally, the process checks if the view becomes valid. This process does not require human intervention and requires no change in the importing views.

5.4.3.2 Process Tackling Relation Rename

If the schema evolution type of *se* is "Relation Rename", let *r1* be the original relation schema and *r2* be the evolved relation schema, then for each map *Mi* in *AML*,

1) In the leaf list of the exporting view of Mi, attempt to find a leaf relation schema r' such that r' = r1. If r' is found, then substitute r2 for r'.
2) In the atomic view list of the exporting view of Mi, find the atomic view *v* such that r is identical to one of the operand relation schemas of the operation of the view expression of v. If v is found, substitute r' for the operand relation schema of v.
3) Check whether the exporting view of Mi should be discarded based on discard rules. If any rule is true, discard Mi and repeat 1 for next map.

Similar to the attribute rename, this process only needs to change the name of the corresponding relation schema in the affected views. The affected views are those which have a projection operation. This process does not require human intervention and requires no change in the importing views.

5.4.3.3 Process Tackling Relation Removal

If the schema evolution type of *se* is "Relation Removal", let *r* be the original relation schema, then for each map *Mi* in *AML*,

1) In the leaf list of the exporting view of Mi, attempt to find a leaf relation schema r' such that r' = r. If r' is found, then remove r'.
2) If there is no leaf relation schema in the leaf list of the exporting view of Mi, then discard Mi and repeat from 1 for next map.
3) In the atomic view list of the exporting view of Mi, find the atomic view *v*

such that r is identical to one of the operand relation schemas of the operation of the view expression of v. If v is found, discard v and let s be the relation schema of v.

4) In the atomic view list of the exporting view of Mi, find the atomic view *v'* such that s is identical to one of the operand relation schemas of the operation of the view expression of v'. If v' is found, then:

   If the operation type of the view expression of v' is not "Union", discard v' and let s be the relation schema of v' and repeat from 4: otherwise remove the operand relation schema that is identical to s.

5) Check whether the exporting view of Mi should be discarded based on discard rules. If any rule is true, discard Mi and repeat 1 for next map.

This process firstly removes the original relation schema from the affected atomic view and then checks whether it is still valid. If not, it discards the atomic view. If the atomic view is discarded, it must affect its parent view. Therefore, the process removes it from its parent atomic view, and then checks if the parent view is valid. By doing the above steps recursively, the complete exporting view will be modified to work properly or be discarded. If the exporting view is discarded, the importing view and the map will be discarded as well.

5.4.3.4 Process Tackling Attribute Decomposition

If the schema evolution type of *se* is "Attribute Decomposition", let r be the original relation schema of se and A be the original attribute and L{Bi,…,Bn} be the evolved attribute of se, and op be the operator of se, then for each map Mi in AML,

1) In the leaf list of the exporting view of Mi, attempt to find a leaf relation schema r' such that r' = r. If r' is found, then attempt to find the attribute A' in the attribute list of r' such that A' = A, if A' is found, then remove A' and add all the evolved attributes in L into the attribute list of r'.

2) In the atomic view list of the exporting view of Mi, attempt to find the atomic view *v* such that r is identical to one of the operand relation schemas of the

operation of the view expression of v. If v is found, then attempt to find the attribute A' in the attribute list of the operand relation schema of the view expression of v such that A' = A1, if A' is found, then remove A' and add all the evolved attributes in L into the attribute list of the operand relation schema of the view expression of v.

3) Create a new source *sc(type, value)* such that the *type* is "Expression" and the value is (op', L') where op' = op and L' = L.
4) For each projection element in the projection list of v, if the source type of the source of the current element is "Expression" and there is one operand in the operand list of the source value such that the content of the operand is identical to A, then discard v and let s be the relation schema of v and go to 5; otherwise if the source type of the source of the current element is "Attribute" and the source value is identical to A, then substitute the new source sc for the current source. Go to 6.
5) In the atomic view list of the exporting view of Mi, find the atomic view *v'* such that s is identical to one of the operand relation schemas of the operation of the view expression of v'. If v' is found, then:

   If the operation type of the view expression of v' is not "Union", discard v' and let s be the relation schema of v' and repeat from 5; otherwise remove the operand relation schema that is identical to s.
6) Check whether the exporting view of Mi should be discarded based on discard rules. If any rule is true, discard Mi and repeat 1 for next map.

This process firstly finds the original attribute in the projection list of the atomic view. It then changes the source that is the original attribute into an expression that composes all the evolved attributes. This recomposes the original attribute. The output attribute will not be changed and the value of it is the result of the expression that is the same as the original attribute. This process does not require human intervention and requires no change in the importing views.

5.4.3.5 Process Tackling Relation Decomposition

If the schema evolution type of *se* is "Relation Decomposition", let r be the original relation schema of se, and L{R1,…,Rn} be the evolved relation schema list of se, and *AL* be the common attribute list of se, then for each map Mi in AML,

1) In the leaf list of the exporting view of Mi, attempt to find a leaf relation schema r' such that r' = r. If r' is found, then remove r' and add all the evolved relation schemas in L into the leaf list.
2) For an evolved relation schema in L:
    i. Create a view schema vs such that the attribute list of vs is identical to the attribute list of the current evolved relation schema and the type of view schema is "View".
    ii. Create an empty projection list PL.
    iii. For each attribute of the attribute list of the current evolved relation schema, create a project element pe(s, a) such that the source type of the source of s is "Attribute" and the source value of s is the identical to the current attribute and the output element a is also identical to the current attribute, store pe in PL.
    iv. Create a projection operation op(Ri, PL) where Ri is the current evolved relation schema in L.
    v. Create a view expression ve(type, op) where type = "Projection".
    vi. Create an atomic view Vi(vs, ve); store Vi into an atomic view list *F*.
3) Let vs' be the view schema of the first atomic view of F. For each atomic view in F:
    i. If the current atomic view is not the first one in F, then create an atomic view AVi(vs, ve) such that:
        a) the attribute list of vs is the union of the attribute list of vs' and the attribute list of the current atomic view,
        b) and the type of vs is "View" and the operation type of ve is "Join",
        c) and the two operand relation schemas of the operation of ve

are vs’ and the view schema of the current atomic view,

d) and the common attribute list of the operation of ve is AL.

ii. If the current atomic view is not the first one in F, then store AVi in list G and let vs’ be the view schema vs of AVi.

4) In the atomic view list of the exporting view of Mi, find the atomic view *v* such that r is identical to one of the operand relation schemas of the operation of the view expression of v. If v is found, then substitute vs’ for the operand relation schema of v (at this stage, vs’ is the view schema of the last atomic view in G).
5) Add all the atomic views in list F into the atomic list of exporting views of Mi.
6) Add all the atomic views in list G into the atomic list of exporting views of Mi.
7) Check whether the exporting view of Mi should be discarded based on discard rules. If any rule is true, discard Mi and repeat 1 for next map.

This process first finds the atomic view that performs a projection operation on the original relation schema. It then creates a new atomic view of each evolved relation schema such that the new atomic view performs projection operation on the evolved relation schema. The output attributes of the new atomic view are the same as that of the evolved relation schema. After that, the process joins these atomic views recursively by creating new atomic views that perform joins on the atomic views that perform projection. Consequently, there is an atomic view that is the result of joining all the projection atomic views. Substitute this atomic view for the original relation schema in the atomic view found at the beginning. This process can be considered as building a new exporting view joining those evolved views, and then substituting the root atomic view of the new exporting view for the leaf relation schema that is the original schema of the schema evolution. This process does not require human intervention and requires no change in the importing views.

5.4.3.6 Process Tackling Attribute Removal

If the schema evolution type of *se* is "Attribute Removal", let r be the original relation schema of se and A be the original attribute, then for each map *Mi* in *AML*,

1) In the leaf list of the exporting view of Mi, attempt to find a leaf relation schema r' such that r' = r. If r' is found, then attempt to find the attribute A' in the attribute list of r' such that A' = A, if A' is found, remove A'. Let vs be r.
2) In the atomic view list of the exporting views of Mi, attempt to find the atomic view *v* such that vs is identical to one of the operand relation schemas of the operation of the view expression of v. If v is found, then continue; otherwise go to 9.
3) If the operation type of the view expression of v is "Projection", then attempt to find the attribute A' in the attribute list of the operand relation schema of the view expression of v such that A' = A, if A' is found, then remove A'. For each projection element in the projection list of v:
    i. If the source type of the source of the current element is "Expression" and there is one operand in the operand list of the source value such that the content of the operand is identical to A, then discard Mi and go to 10.
    ii. Otherwise if the source type of the source of the current element is "Attribute" and the source value is identical to A, then let B be the output attribute of the current element and vs be the view schema of v and remove the current projection element and remove the attribute that is identical to B from the attribute list of the view schema of v. repeat from 2 for the parent view.
4) If the operation type of the view expression of v is "Join", then attempt to find the operand relation schema R' of the operation of the view expression of v such that R' = vs, if R' is found, attempt to find the attribute B' in the attribute list of R' such that B' = B; if B' is found, then remove B'. Attempt to find the attribute B' in the common list of the operations of the view expression of v such that B' = B, if B' is found, then remove B' and go to 10;

otherwise, attempt to find the attribute B’ in the attribute list of the view schema of *v* such that B’ = B, if B’ is found, then let vs be the view schema of v and remove B’ and repeat from 2 for the parent view.

5) If the operation type of the view expression of v is “Selection”, then attempt to find the attribute B’ in the attribute list of the operand relation schema of the operation of the view expression of v such that B’ = B, If B’ is found, then remove B’. Attempt to find an operand in the condition list of the operation of the view expression of v such that the content of the operand is identical to B. If one such operand is found, discard Mi and go to 10. Attempt to find the attribute B’ in the attribute list of the view schema of *v* such that B’ = B, if B’ is found, then let vs be the view schema of v and remove B’ and repeat from 2 for the parent view.
6) If the operation type of the view expression of v is “Cartesian Product”, then attempt to find the operand relation schema R’ of the operation of the view expression of v such that R’ = vs, if R’ is found, then attempt to find the attribute B’ in the attribute list of R’ such that B’ = B, if B’ is found, then remove B’. Attempt to find the attribute B’ in the attribute list of the view schema of *v* such that B’ = B, if B’ is found, then let vs be the view schema of v and remove B’. Go to 10.
7) If the operation type of the view expression of v is “Union”or “Difference”or “Intersection”, then discard Mi and go to 10.
8) If the operation type of the view expression of v is “Grouping”, then discard Mi and go to10.
9) Compare *vs* with the view schema *vsi* of the importing view to check if *vs* and *vsi* have the same name and type. If they have, then attempt to find an attribute B’ in the attribute list of vsi such that the name of the B’ is identical to the name of B. If B’ is found, then remove B’.
10) Check whether the exporting view of Mi should be discarded based on discard rules. If any rule is true, discard Mi and repeat 1 for next map.

This process removes the original attribute from the atomic view that involves the original relation schema and checks whether is still a valid view. If it is not, then discard the entire map. If it is a valid view, the process will find its parent view and remove the corresponding original attribute from the parent view. By performing the above three steps recursively, the complete exporting view is modified and discarded. If the attribute is removed from the root atomic view, the corresponding attribute in the importing view will be removed. Note that in this process, when an attribute is removed from an grouping operation, the exporting view is discarded because we consider it may have changed the semantic meaning of the exporting view and require manual modification.

5.4.3.7 Process Tackling Attribute Domain Change

If the schema evolution type of *se* is "Attribute Domain Change", let r be the original relation schema of se and A1 be the original attribute and A2 be the evolved attribute, then for each map Mi in AML,

1) In the leaf list of the exporting view of Mi, attempt to find a leaf relation schema r' such that r' = r. If r' is found, then attempt to find the attribute A' in the attribute list of r' such that A' = A1, if A' is found, then substitute A2 for A'. Let vs be r.
2) In the atomic view list of the exporting view of Mi, attempt to find the atomic view *v* such that vs is identical to one of the operand relation schemas of the operation of the view expression of v. If v is found, then attempt to find the attribute A' in the attribute list of the operand relation schema of the view expression of v such that A' = A1, if A' is found, then substitute A2 for A'.
3) If the operation type of the view expression of v is "Projection", then for each projection element in the projection list of v:
    i. If the source type of the source of the current element is "Expression" and there is one operand in the operand list of the source value such that the content of the operand is identical to A1, then discard Mi and go to 9.

ii. Otherwise, if the source type of the source of the current element is "Attribute" and the source value is identical to A1, then let B be the output attribute of the current element and vs be the view schema of v and substitute A2 for the source value and for the output attribute of the current projection element and for the attribute that is identical to B from the attribute list of the view schema of v. repeat from 2 for the parent view.

4) If the operation type of the view expression of v is "Join", then attempt to find the operand relation schema R' of the operation of the view expression of v such that R' = vs, if R' is found, attempt to find the attribute B' in the attribute list of R' such that B' = B; if B' is found, substitute A2 for B'. Attempt to find the attribute B' in the common list of the operations of the view expression of v such that B' = B, if B' is found, then discard Mi and go to 9; otherwise, attempt to find the attribute B' in the attribute list of the view schema of *v* such that B' = B, if A' is found, then let vs be the view schema of v and substitute A2 for B'' and repeat from 2 for the parent view.

5) If the operation type of the view expression of v is "Selection", then attempt to find the attribute B' in the attribute list of the operand relation schema of the operation of the view expression of v such that B' = B, If B' is found, then substitute A2 for B'. Attempt to find an operand in the condition list of the operation of the view expression of v such that the content of the operand is identical to B, if one such operand is found, discard Mi and go to 9. Attempt to find the attribute B' in the attribute list of the view schema of *v* such that B' = B, if B' is found, then let vs be the view schema of v and substitute A2 for B' and repeat from 2 for the parent view.

6) If the operation type of the view expression of v is "Cartesian Product", then attempt to find the operand relation schema R' of the operation of the view expression of v such that R' = vs, if R' is found, then attempt to find the attribute B' in the attribute list of R' such that B' = B, if B' is found, then substitute A2 for B'. Attempt to find the attribute B' in the attribute list of the

view schema of *v* such that B’ = B, if B’ is found, then let vs be the view schema of v and substitute A2 for B’. Go to 9.

7) If the operation type of the view expression of v is “Union”or “Difference”or “Intersection”, then discard Mi and go to 9.
8) If the operation type of the view expression of v is “Grouping”, then attempt to find an aggregation in the aggregation list of the operation of the view expression of v such that the operand attribute of the aggregation is identical to B, if one such aggregation is found, discard Mi and go to 9; otherwise, attempt to find the attribute B’ in the attribute list of the operand relation schema of the operation of the view expression of v such that B’ = B, If B’ is found, then substitute A2 for B’. Attempt to find the attribute B’ in the grouping attribute list of the operation of the view expression of v such that B’ = B, If B’ is found, then substitute A2 for B’. Attempt to find the attribute B’ in the attribute list of the view schema of *v* such that B’ = B, if B’ is found, then let vs be the view schema of v and substitute A2 for B’ and repeat from 2 for the parent view.
9) Check whether the exporting view of Mi should be discarded based on discard rules. If any rule is true, discard Mi and repeat 1 for next map.

It is similar to the process tackling attribute removal described in 5.4.3.6. This process changes the original attribute to the evolved attribute from the atomic view that involves the original relation schema and checks whether it is still a valid view. If is not, then discard the entire map. If it is a valid view, the process will find its parent view and change the corresponding original attribute to the evolved attribute from the parent view. By performing the above three steps recursively, the complete exporting view is modified and discarded. The difference from the process described in section 5.4.3.6 is that this process will not modify the corresponding attribute in the importing view.

5.4.3.8 Process Tackling Database Removal

This process is a special process that is different from the above seven processes. Instead of modifying the affected views in the affected map list, this process finds the mapping of the removed local schema from the entire mapping list and discards it. Therefore, this process can be represented as:

AVM: (SE, MPS)

The following steps are taken by the process when the database is removed. Let *se* be the schema evolution whose type is "Database Removal", and *ls-name* be the local schema name of the schema evolution, and *MPS* be the entire mapping list, then:

1) In the entire mapping list MPS, find a mapping M such that the name of M is identical to *ls-name.* If M is found, then discard M.

Although there is only one step for the time being, the process will be extended to modify the organizational structure tree and DSs in the meta-database in chapter 7.

5.5 Summary

This chapter introduces the algorithm to automatically modify the existing views in the meta-database in response to the schema evolution. The general idea of the algorithm is to modify the existing views based on the schema evolution and then check if the views are still valid based on pre-defined rules. The views that become invalid after the modification will be discarded so that the integrated system will no longer consider them until they are effectively modified by human programmers and made available again.

Different schema evolutions require different processes of automatic view modification. It can be seen from this chapter that the processes 1, 2, 4, 5 and 8 require no human intervention and will not discard any views so that they require no manual work afterwards. The processes 3, 6 and 7 may require human intervention and discard views in some cases so that they may require further manual maintenance

to make them work properly. In addition, adding relation schemas and attributes and local schema require no automatic modification on data in the meta-database. These will be further discussed in the evaluation chapter.

Chapter 6 Query Process

6.1 Introduction

Chapter 5 presented the approach to automatically modify the views in the meta-database. This chapter introduces the approach to processing user queries over the global schema and composing the final results for users. Query processing in this work basically involves four steps: Query Reformulation and Query Decomposition and Query Transformation and Result Composition.

The process of identifying the source databases is the first step of query reformulation to tackle organizational evolution. The problem of "Answering Queries Using Views" is then introduced. One of the reformulation techniques, the Bucket Algorithm [68], for the LAV approach of data integration is then described briefly. After the queries are decomposed into queries which refer to data sources, a straightforward approach to transforming those queries (which refer to a data source) into queries directly over the local schema of that source is illustrated. The resulting composition is finally described, taking into consideration domain conflicts. Although the query processing and result composition are not the focus of this research, they need to be explained to show how the entire approach works.

6.2 Query Processing

Query processing in an integrated database system involves many aspects such as (i) query translation which translates a high-level and more semantic query into a low-level query, (ii) query decomposition that decomposes a query written in terms of the global schema into queries that refer to the data sources, (iii) query transformation which transforms the queries to be executed by the source databases, and (iv) query optimization which optimizes the query to gain a better response time. In this research, however, the focus is on two steps of query processing, Query Reformulation and Query Transformation.

Figure 6-1 shows the query processing of this research. It illustrates that the input of Query Reformulation is an extended conjunctive query over the global schema. The conjunctive query is then decomposed into subqueries over exporting views that refer to particular data sources, by reformulation techniques for LAV approach. These queries over exporting views are still in a conjunctive query language and over the global schema. Therefore, the queries need to be transformed into queries directly over local schemas based on the definitions of exporting views. As the exporting views are defined using the relation algebra query language, there is another step between query reformulation and query transformation, called query translation. Query translation is to translate queries in a language into queries in another query language. Once queries have been translated into queries over local schemas, the queries then need to be translated into queries that are in the query language supported by the source databases.

In fact, a user query may be a high-level query such as Relational Calculus before it is translated into a conjunctive query. In this research, an extended conjunctive query is used as the user query language. The extended conjunctive query is simply the conjunctive query with an additional property, *Organizational Scope*. The aim of this property is to narrow down the scope of source databases that will be searched by the query processing. Therefore, the *user query* can be further represented as following:

User Query: (Q(GS), OS: (*Org****:String,*** *OP****:Stirng))***

The first property of the user query is a conjunctive query on the global schema, while the second property is the *organizational scope* that is a pair. The first property is the *organization name* indicating the organization in which the query processing will search for the source databases, while the second property is an organizational property name indicating the node on the tree, the children of which will be accessed. This user query is defined in order to illustrate how the query processing works to tackle organizational evolution.

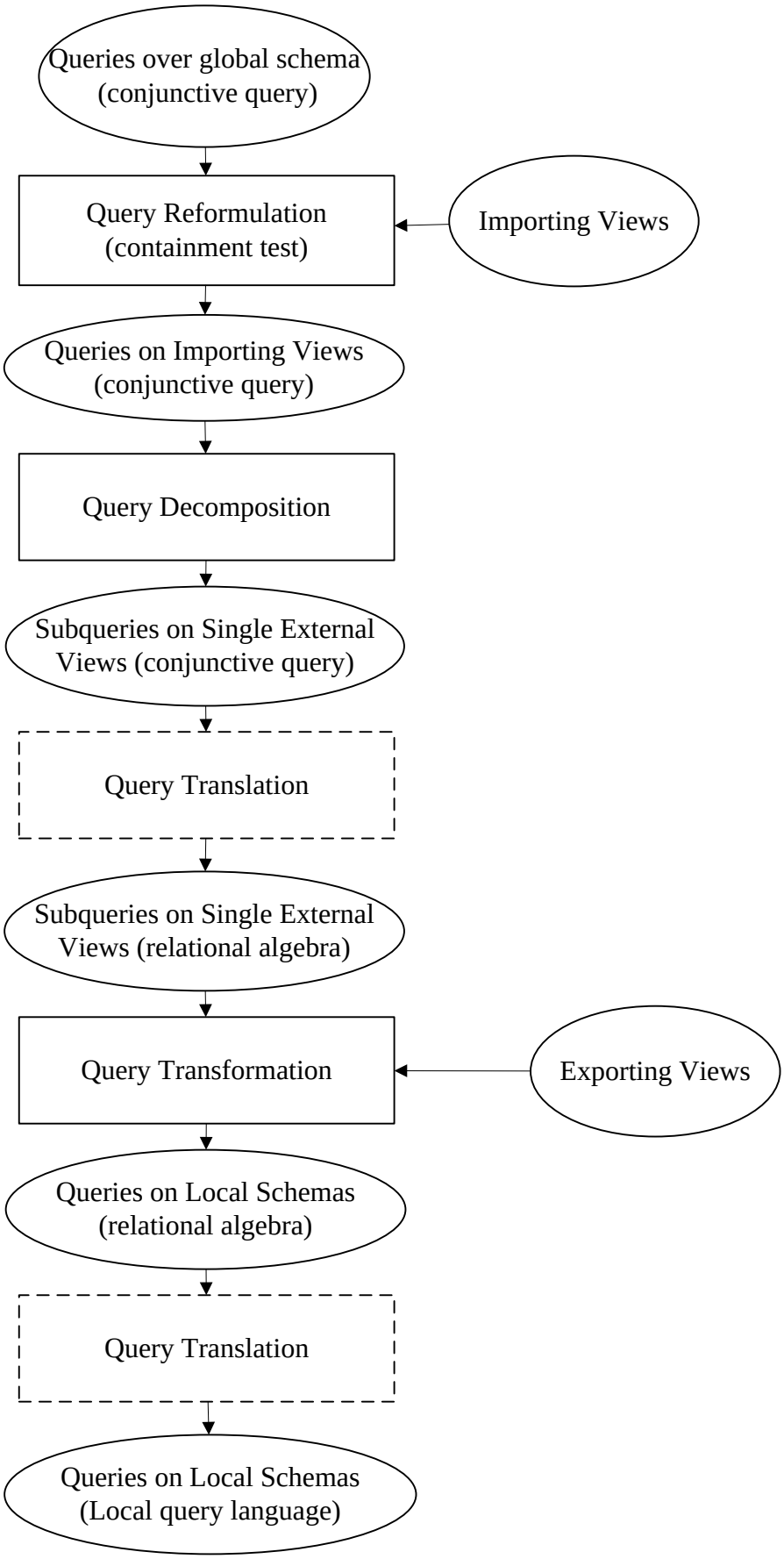

Figure 6-1 Query Processing

The translation between two query languages is not a focus of this research and therefore is not described.

6.2.1 Query Reformulation

Chapter 4 described how to associate each source database with the global schema by building importing views that represent the local schema over the global schema. Having built the association between the local schemas and the global schema, a user is able to raise a query (conjunctive query in this work) in terms of the global schema without knowing the underlying source database. The query reformulation is responsible for reformulating the query into a query that refers to the source databases. In this work, there is an additional process, Identifying Source Databases, which deals with organizational evolution that is one of the central issues in this research.

6.2.1.1 Identifying Source Databases

Once a user query has been received by the query processing, the first step is to identify the source databases that will be taken into account, according to the organizational scope of the user query. Therefore the process of identifying source databases can be regarded as a function:

ISD: (OS(Org, OP)) → Source Database List

The function takes the organizational scope of the user query as input and the output is a list of source database, termed *Source Database List*. As the organizational structure trees have been pre-defined, it is simple to get all the source databases that are the leaves of the branches under the given organizational property and the given organization. However, the process of identifying source databases in this work takes an additional step to tackle the organizational evolution that has been generally introduced in Chapter 5. In order to describe the process, the organizational evolution is first defined.

6.2.1.1.1 Organizational Evolution and Its Representation in Meta-database

Recall that in Chapter 4, an organizational structure is represented as a tree structure. Organizational evolution represents the changes in the organizational structure. Although this type of evolution does not have an impact on the view definitions, it has some impact on existing user queries. Two types of evolution are considered in this research:

1) **Organizational Property Removal**: An organizational property of the tree of an organization may be removed. It can be represented as a pair in the meta-database:

   OPRemoval *(Org****: Organization****, original:* ***Organizational Property****)*

   It consists of two properties: *original organization* and *original organizational property*. The first, *original organization*, represents the organization whose OP has been removed. The second, original organizational property, is the OP that has been removed by the evolution.

2) **Organizational Property Rename**: The name of an organizational property of the tree of an organization may change. It can be represented as a triple in the meta-database:

   OPRename *(Org****: Organization****, original:* ***Organizational Property,***
   *evolved****: Organizational Property*** *)*

   It consists of two properties: *original organization* and *original organizational property* and *evolved organizational property*. The first, *original organization*, represents the organization in which OP has changed. The second, *original organizational property*, is the OP before the change. The third, *evolved organizational property*, is the OP with the new name after the change.

3) **Organization Removal**: An organization itself may be removed. It can be represented as the following in the meta-database:

***Organization Removal** (Org**: Organization**)*

It has one property, *original organization*, which is the organization removed by this evolution.

4) **Organization Rename**: The name of an organization itself may change. It can be represented as a pair in the meta-database:

***Organization Rename** (Org**: Organization,** evolved**: Organization**)*

It has two properties: *original organization* and *evolved organization*. The original organization is the organization before the change, while the evolved organization is the organization with the new name after the change.

5) **Parent Change**: The parent property of an organizational property (a CP or a source database) of the tree of an organization may change.
6) **Organization Addition**: A new organization is added into the organization list of the meta-database.
7) **Organizational Property Addition**: A new organizational property is added into an existing organization.

Since the last three types of evolution have no impact on the user queries, they are not considered and formally represented in the meta-database. This is one of the advantages of the design of the organizational structure tree in the meta-database. It will be discussed in the evaluation chapter.

In order to store the organizational evolution into the meta-database, a data type that represents the organizational evolution is defined as follows:

Organizational Evolution Operation: OPRemoval | OPRename | Organization Removal | Organization Rename

An *organizational evolution* can then be represented as:

Organizational Evolution: (*type****: OrganziationalEvolutionType,*** *evolution****:*** **Organizational Evolution Operation*)***

The first property is *schema evolution type*, a string which indicates which organizational evolution operation applied.

OrganziationalEvolutionType: String

∀*X****: SchemaEvolutionType,*** *X* ***∈ {*** *"Organizational Property Removal", "Organizational Property Rename", "Organization Removal", "Organization Rename"* ***}***

The second element of organizational evolution is the organizational evolution operation defined above.

All organizational evolution can then be represented as an *organizational evolution list*, represented as:

OEL: {*oe****: Organizational Evolution }***

6.2.1.1.2 The Extended Meta-database

The meta-database is then extended to involve organizational evolution:

MD: {*lsl***: LSL,** *gs***: GS,** *domain***: GAD,** *m***: MPS,** *org***: Organization List,** *oel***: OEL}**

6.2.1.1.3 Process of Identifying the Source Databases

Having defined and represented organizational evolution, the process of identifying the source databases can be discussed. It has been realized that a user query will be affected only when the organizational scope involves the organization or the organizational property that has evolved. Namely, there is an organizational evolution in the organizational evolution list of the meta-database such that the original organization or the original organizational property of it is involved in the user query. The general process of identifying the source databases is as follows:

1) Check if the organization or the organizational property required in the user query has changed. If it has changed, change the user query to access the correct organization and organizational property. It the organization or the organizational property does not exist anymore, it means that the user query is not valid anymore and cannot be answered.
2) Find all the source databases under this organizational property in the organization required by the user query and output them for use for the next step of query processing.

Therefore, when a user query UQ is received, the following steps are taken by the process of identifying the source databases.

Let OS(org, op) be the organizational scope of the user query, where org is the organization and op is the organizational property designated by the user query, then:

1) In the organizational evolution list, attempt to find an organizational evolution oe such that the original organization org' of the organizational evolution operation of oe is identical to the organization *org* of *OS*. If org' is found, then continue; otherwise go to 6.
2) If the type of oe is "Organization Remove", then reject the user query UQ and stop the process.
3) If the type of oe is "Organization Rename", then change org to the evolved organization of the organizational evolution operation of oe and repeat from 1.
4) If the type of oe is "Organizational Property Rename", then check if the organizational property op is identical to the original organizational property of the organizational evolution operation of oe. If identical, then change op to the evolved organizational property of the organizational evolution operation of oe and repeat from 1.
5) If the type of oe is "Organizational Property Removal", then check if the organizational property op is identical to the original organizational property of the organizational evolution operation of oe. If identical, then reject the user query UQ and stop the process.

6) Attempt to find org in the organization list of the meta-database. If found, attempt to find op in the node list of the tree of org. If found, find all the organizational properties that are source databases of the branches under op by recursively traversing the parent relations in the parent relation list and store them into the list F. output F.

Note that if the user query does not designate any organization, it means all the local schemas in the local schema list of the meta-database will be considered. In this case, the process of identifying the source database will not be taken

6.2.1.2 Query Reformulation

In this research, the resulting query of the query reformulation is over the exporting views representing the source databases, and is still in a conjunctive query language. There are two important criteria to be met in query reformulation [103]:

- Semantic correctness of the reformulation: The answers obtained from the sources will be correct answers to the original query.
- Minimizing the source access: Sources that cannot contribute any answer or partial answer to the query should not be accessed.

As the LAV approach is used to describe source databases, the query reformulation is not straightforward and is one of the applications of an important problem called "Answering Queries using Views". In the next sections, the source databases being considered are those outputted from the process of identifying the source databases in cases where the user query designates a specific organization.

6.2.1.2.1 Answering Queries using Views

Informally, the problem is defined as follows: Giver a query Q over a database schema, and a set of view definitions V1,…,Vn over the same schema, rewrite the query using the views as Q' such that the subgoals in Q' refer only to view predicates. If such a rewriting of Q into Q' can be found, then to answer Q, it is enough that Q' is

answered using the answers of the views [68].

In our architecture, the query reformulation means that by using the exporting views describing the source databases in terms of the global schema, the integrated system can answer a user query written in terms of the global schema by rewriting this query as another query referring to the exporting views rather than the global schema itself. Basically, the user query is decomposed into several subqueries each of which is referring to a single source database.

The ideal rewriting we expect to find would be an "equivalent" rewriting. However, this may not always be possible. Particularly in data integration systems, source database incompleteness and limited source capability would lead to rewritings that approximate the original query. Among the many possible approximate rewritings, the "best" one needs to be found. The technical term for the best rewriting is "maximally-contained" rewriting. These terms are formalized as following [68]:

Query Containment and Equivalence: A query Q' is contained in another query Q if, for all database D, Q'(D) is a subset of Q(D). A query Q is equivalent to another query Q' if Q' and Q are contained in one another.

Equivalent Rewritings: Let Q be a query and V = V1,…, Vm be a set of view definitions. The query Q' is an equivalent rewriting of Q using V if:

- Q' refers only to the views in V,
- Q' is equivalent to Q.

Maximally-contained Rewritings: Let Q be a query and V = V1, …, Vm be a set of view definitions in a query language L. The query Q' is a maximally-contained rewriting of Q using V with respect to L if:

- Q' refers only to the views in V,
- Q' is contained in Q, and

- there is no rewriting Q1 such that Q' ⊆ Q1 ⊆ Q and Q1 is not equivalent to Q'.

6.2.1.2.2 Completeness and Complexity of Finding Query Rewritings

Theoretical issues related to the problem of finding query rewritings using views including completeness and complexity of the query rewriting algorithms, are now considered. [68] discussed the issues in detail.

Completeness of a query rewriting algorithm is defined as follows in [68]: Given a set of views V and a query Q, will the query rewriting algorithm always find a rewriting of Q using V if there exists such a rewriting? The answer to this question also depends on the query language used to express the query rewriting. Sometimes the limited expressiveness of the language may prevent the algorithm from finding a query rewriting although one exists. In the case where no equivalent query rewriting exists, a maximally-contained rewriting need to be found. [68] also points out that sometimes recursive Datalog rules need to be used to come up with a maximally-contained rewriting. This exemplifies the dependence of the algorithms on the expressiveness of the query language.

The complexity of the query rewriting algorithm can be discussed under different language and modeling assumptions. In general, they are NP-Complex [68].

6.2.1.2.3 The Bucket Algorithm

Given a query Q and a set of views V1,…, Vn, to rewrite Q in terms of Vis, we have to perform an exhaustive search among all possible conjunctions of m or fewer view atoms where m is the number of subgoals in the query. In order to avoid the exhaustive search, the Bucket Algorithm [68] is used in this work. The main idea underlying the Bucket Algorithm is that the number of query rewritings that need to be considered can be reduced if each subgoal in the query is considered separately to determine which views may be relevant to each subgoal. Given a query Q, the Bucket

Algorithm finds a rewriting of Q in two steps:

1. The algorithm creates a bucket for each subgoal in Q which contains the views (i.e., source databases) that are relevant to answering that particular subgoal.
2. The algorithm tries to find query rewritings that are conjunctive queries, each consisting of one conjunct from every bucket. For each possible choice of element from each bucket, the algorithm checks whether the resulting conjunction is contained in the query Q or whether it can be made to be contained if additional predicates are added to the rewriting. If so, the rewriting is added to the answer. Hence, the result of the Bucket Algorithm is a union of conjunctive rewritings.

The following simple example shows how the algorithm works:

Assume that there are three sources databases S1, S2 and S3. S1 contains information about cars produced after 1990. S2 contains cars sold by the dealer named "ACME". S3 contains car reviews. Assume that the global schema has the relations with the following schemas:

CAR (vin, status)

MODEL (vin, model, year)

SELLS (dealer_name, vin, price)

Review (vin, review)

Furthermore, the importing views defined for the data sources are:

S1 (vin, status, model, year) ← CAR (vin, status),
MODEL (vin, model, year), year ≥ 1990

S2 (vin, status, model, price) ← CAR (vin, status), MODEL (vin, model, year), SELLS (dealer_name, vin, price), dealer_name = "ACME"

S3 (vin, review) ← REVIEW (vin, review)

Assume that a user is looking for used cars produced before 1990, their reviews and where they are sold. The following query is posed by the user to the global schema:

Q(vin, dealer, review) ← CAR(vin, status), MODEL(vin, model, year),

SELLS(dealer_name, vin, price), REVIEW(vin, review), year < 1990, status = "used"

For ease of presentation, the initial letters of the attributes are used. The first step of the Bucket Algorithm constructs the following buckets per subgoal in Q:

CAR(V, S)	MODEL(V, M, Y)	SELLS(D, V, P)	REVIEW(V, R)
S2(V, S, M', P')	S2(V, S', M, P')	S2(V, S', M', P)	S3(V, R)

Notice how views are mapped to each query subgoal by the buckets. It is important to note that we did not insert S1 into buckets CAR(V, S) and MODEL(V, M, Y) because of the constraint on the year attribute in the query. Since S1 contains cars which are produced after 1990 and the query asks for the ones produced before 1990, S1 cannot answer the query.

The second step of the algorithm chooses one view from each bucket and combines them into a new query. Since for this simple example there is already one entry per bucket, there will be one combination of views. In general, we would have to construct one query per possible combination of the entries and we would test for containment in the original query. Then the result would be the union of all the contained queries.

The following new query is obtained written in terms of the importing views rather than the relations schemas in the global schema:

Q'(vin, dealer, review) ← *S2(vin, status, model, price), S3(vin, review), year < 1990, status = "used"*

Note that there are two redundant references to view S2 and we also added the extra constraints on the *year* and status attributes since without these predicates, Q' would

not be contained in Q. In terms of completeness and complexity, [68] mentions that the Bucket Algorithm is guaranteed to find maximally-contained rewriting of a query if the query does not contain arithmetic comparison predicates. However, the second phase may take exponentially long.

There are some alternative approaches to answering queries using views, such as the Inverse-Rules Algorithm [68], the MiniCon Algorithm [103] and the Shared-Variable-Bucket Algorithm [104]. The Inverse-Rules Algorithm is completely different from the Bucket Algorithm. The key idea underlying the algorithm is to construct a set of rules that invert the view definitions, i.e., rules that show how to compute tuples for the database relations [68]. It has a significant drawback for our research, since it attempts to recomputed the extensions of the database relations. Namely, the tuples in the relations need to be accessed before the rewritings of a query are found. Hence, it significantly increases the access to the source databases and therefore is not suitable of our research.

The MiniCon Algorithm and the Shared-Variable-Bucket Algorithm are both improved algorithms based on the idea of the Bucket Algorithm, providing extra steps in order to reduce the number of views to be considered for the rewriting step. The Bucket Algorithm was chosen is this research, because it is a basic algorithm which is easier to implement in the case study. The extra steps in the MiniCon Algorithm and the Shared-Variable-Bucket Algorithm require much more complex design and programming for implementation, and cannot help in the major problem in this research which is evolution.

6.2.1.2.4 Summary

To sum up, a query in terms of the global schema raised by a user in a conjunctive query language is reformulated into a query that refers to the importing views representing the source databases. The query reformulation techniques for LAV approach are used, which is not straightforward and is one of the applications of an

important problem called “Answering Queries using Views”. The Bucket Algorithm, is used to conduct query containment tests and rewrite the query. The resulting query consists of only the importing views that represent the data sources. These resulting queries are ready to be sent to corresponding source databases where they are transformed into queries that refer to the local schema directly. This is introduced in the next section.

6.2.2 Query Decomposition

As mentioned in the previous section, the query over the global schema is reformulated into several queries consisting of only the importing views. The union of these queries together produces the final result for the original query. Each query resulting from the query reformulation is then decomposed into subqueries each of which contains only one importing view representing a single source database. Namely, each subgoal of the conjunctive query, except the predicate subgoal, will become a subquery. Take the car-dealer example used in the last section, the new query is:

Q’(vin, dealer, review) ← S2(vin, status, model, price), S3(vin, review),
year < 1990, status = “used”

The query can be decomposed into two subqueries:

Q01 (vin, status, model, price) ← S2 (vin, status, model, price), status = “used”

Q02 (vin, review) ← S3(vin ,review)

Note that Q01 involves the predicate status = “used”, because S2 has the attribute status so that putting this predicate can reduce the tuples transferred from the source database. Furthermore, each subquery will be sent to a source database site where an exporting view corresponding to the importing view is defined. As the exporting view has the same set of attributes as that of the importing view, the above two queries can be rewritten as the following queries if ExV2 and ExV3 are exporting views corresponding to S2 and S3 respectively:

S2(vin, status, model, price) ← ExV2 (vin, status, model, price), status = "used"

S3(vin, review) ← ExV3 (vin, review)

The names of Q01 and Q02 are replaced by the names of importing views because it is more straightforward to show how the results from the source databases will be composed. The subqueries S2 and S3 are then sent to the source databases represented by the importing views S2 and S3 where they will be transformed into queries that refer directly to the local schemas.

6.2.3 Query Transformation

The subqueries that are sent to source databases to be executed consist of an exporting view, which means they are still in the global schema because the exporting views are defined in terms of the local schema but are homogeneous to the global schema. Thus, those subqueries need to be further reformulated into queries that are over the local schema, in order for them to be executed. This process is called *query transformation*.

Before the transformation, there is one more step, query translation. As the subqueries are still in a conjunctive query language and the exporting views are defined in a relational algebra language, those subqueries need to be translated into a relational algebra language. For example, query S2 can be translated into the following query:

$$S2(vin.\ Status, model, price) := \sigma_{status="used"}(ExV2)$$

How to translate a query in one query language into an equivalent query in another query language is not the focus of this work. Therefore, it will not be described in any detail. However, it is expected that the translation in this architecture is very simple, because each query only has one exporting view.

Sequentially, at a source database site, the resulting subqueries from the translation need to be transformed into queries that are directly over the local schema. The query transformation is based on the definitions of the exporting views and is very

straightforward when compared with the query reformulation.

In this research, the query transformation of a query on views can be described as: Given a query Q over a set of views V1, …, Vn, the query transformation of Q is to substitute the expressions of the views for the views themselves. This may be recursive because a view may be defined over others views. Therefore, the views need to be replaced recursively until the query becomes a query which consists of only base relations.

Recall that in Chapter 4, an exporting view is described as an expression tree that has a number of atomic views on its nodes. Each atomic view, in turn, has an expression that only has one relational algebra operator. The subquery over an exporting view can be represented as an expression tree as well with the exporting view being its leaf. Therefore, to transform the subquery, the exporting view that is the leaf of the expression of the subquery is simply substituted by the expression of the exporting view. For example, we have two relation schemas R (a, b, c, d) and S (a, e, f,), an exporting view can be:

$$ExV\ (a, b, e) := \pi_{a,b,e}\ (\sigma_{b>100}\ (R \bowtie_a S) \cap \sigma_{e=50}\ (R \bowtie_a S))$$

The linear representation of the expression tree of ExV is:

$$V01\ (a, b, c, d, e, f) := R \bowtie_a S$$

$$V02\ (a, b, c, d, e, f) := \sigma_{b>100}\ (V01)$$

$$V03\ (a, b, c, d, e, f) := \sigma_{e=50}\ (V01)$$

$$V04\ (a, b, c, d, e, f) := V02 \cap V03$$

$$ExV\ (a, b, e) := \pi_{a,b,e}(V04)$$

Assume that there is a query over ExV, Q (a, b, e):

$$Q\ (a, b, e) := \pi_{a,b,e}\ (\sigma_{b>150}\ (ExV))$$

Sequentially, in order to transform Q into a query that is directly over R and S, the exporting view ExV is substituted by its expression. Consequently, the expression of

Q becomes:

$$Q(a, b, e) := \pi_{a,b,e}(\sigma_{b>150}(\pi_{a,b,e}(\sigma_{b>100}(R \bowtie_a S) \cap \sigma_{e=50}(R \bowtie_a S))))$$

Using this approach, the subquery is transformed into a query that refers directly to the local schema. However, the query is still written in relational algebra and the DBMS of the source database may support various query languages. Even if the source databases are all relational databases, the DBMSs may support versions of SQLs that are slightly different. Therefore, in order for the subquery to be recognized as a valid query by the DBMS, it needs to be translated into a query that is in the version of SQL supported by the DBMS. For example, the above query Q may be translated into the following query:

Select a, b, e

From R, S

Where R.a = S.a and R.b > 150 and S.e = 50

As mentioned above, query translation is not the focus in our work and therefore will not be discussed in more detail. Also, the query optimization, which chooses a better query plan and rewrites the query, will not be introduced in this thesis, although it is most important, forming the topic of much research elsewhere.

6.3 Result Composition

In practice, result composition is a parallel process to query processing, because the result of each subquery is combined to produce the result for the preceding query. As introduced above, the query over the global schema is reformulated into a set of queries that are written in terms of the importing views representing data sources. The result of the original query is the union of all the resulting queries from query reformulation. Furthermore, each of the reformulated queries is decomposed into subqueries each of which has only one importing view representing a single source database. These subqueries are then sent to relevant source databases, respectively.

The result of each subquery is a set of tuples that can be referred to as a relation instance whose schema is the same as the subquery. Once the results of all the subqueries have been returned from source databases, the query is then able to be evaluated. Those results that need to be combined to produce a result for the preceding query are called intermediate results in this work.

6.3.1 General Process of Result Composition

The process of the result composition in our architecture is very similar to that of a centralized DBMS. To describe the process formally, given a query Q in terms of the global schema, let ImV(ImV1,…,ImVn) be a set of all importing views and ExV(ExV1,…, ExVn) be a set of exporting views corresponding to the importing views, Q is reformulated into a set of queries Q1,…,Qm that consist of only a set of importing views, denoted ImV(Qi) ⊆ ImV. The result of a query is denoted Result (Q). In order to obtain Result (Q), each result of Q1,…,Qm, denoted Result (Qi) $1 \leqslant i \leqslant m$, needs to be obtained by evaluating Q1,…Qm. In order to evaluate a query Qi, each importing view ImVj, ImVj $\in$ ImV(Qi), needs to be sent to each source database where its relevant exporting view ExVj is evaluated. Having been evaluated, the result of ExVj Result(ExVj) that is a relation instance is obtained and is then assigned to ImVj. Once all the importing views ImV (Qi) of a query Qi are assigned a result, Qi is able to be evaluated to obtain a result. The result of Q, Result(Q), is the union of Result(Q1),…,Result(Qm). Figure 6-1 shows the bottom-up process of result composition. The process of result composition in each source database is not described in Figure 6-2, as it is no different from the process of a centralized DBMS.

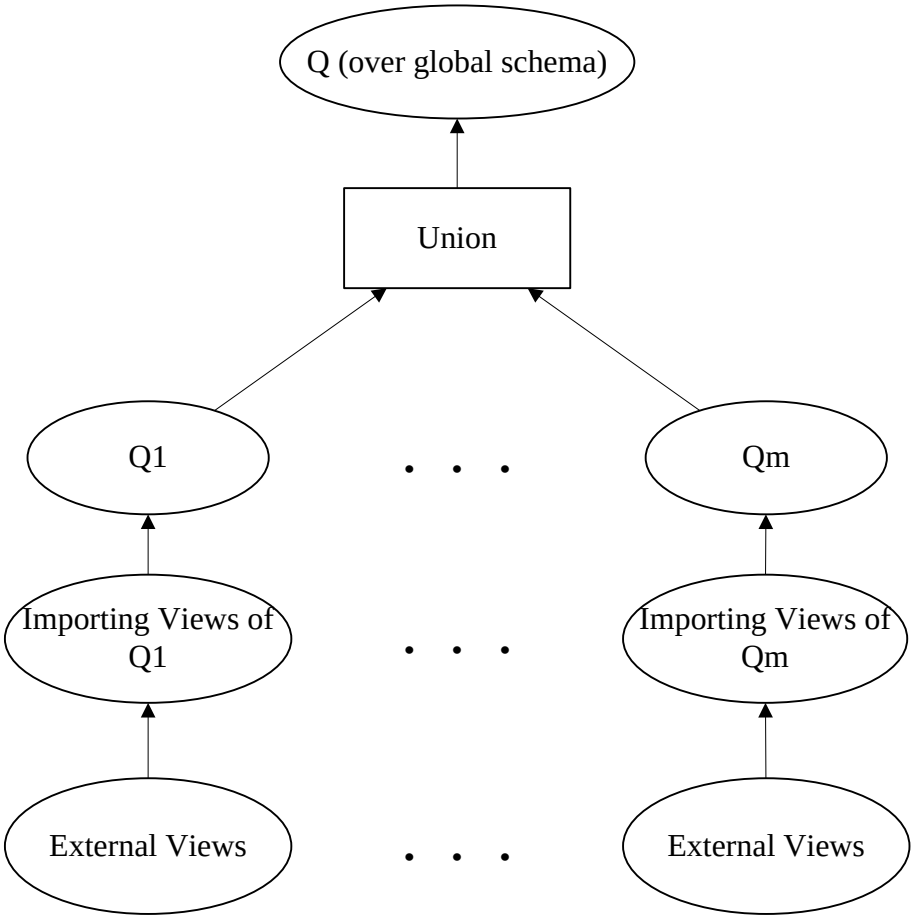

Figure 6-2 The process of result composition

6.3.2 Domain Conversion of Result Composition

One heterogeneity that needs to be tackled during result composition is *domain conflicts*. During result composition, the importing views are evaluated and are assigned results from the corresponding exporting views. However, the attributes of the results from different source databases may have the same names but different domains (or types). Consequently, the results cannot be combined because the attributes with different domains cannot be taken as operands by some operators (i.e. join and algebraic operators) so that the query cannot be further evaluated.

In order to tackle domain conflicts, the process called *Domain Conversion* is added to the process of assigning results of exporting views to importing views. It means that the attributes of the exporting views need to be converted into attributes which have the same domains as those of the attributes of the importing views. To describe the

domain conversion more formally, let ExV be an exporting view that has a set of attributes A(A1,…,An) and ImV be the corresponding importing view of ExV, ImV has a set of attributes B(B1,…,Bn), let relation R(A1,…,An) be a result of ExV, when assign R to ImV, the domain conversion is described as following:

- Compare each pair (Ai, Bi), Ai $\in$ A and Bi $\in$ B and 1 $\leqslant$ i $\leqslant$ n.
- If the domain of Ai is not identical to the domain of Bi, then convert the value of Ai into a value with the same domain to Bi.

Therefore, the domain conversion applies for every exporting view when the result of the exporting view needs to be assigned to an importing view.

6.4 Summary

To sum up, a query posed by a user in terms of the global schema is reformulated into queries that only consist of importing views. These queries are further decomposed into subqueries that refer to single source databases. The subqueries are sent to source databases where they are transformed into queries that can be directly evaluated upon the local schemas. Finally, the result composition that is a bottom up process composes the results to produce the final result for the original query. Domain conversion is applied during result conversion to convert the attributes of exporting views into attributes that have the same domain as that of importing views, in case they are different.

Chapter 7 Services Design and Implementation

7.1 Introduction

Chapter 6 presented the query process step in EA-SODIA and therefore completed the description of EA-SODIA and all the algorithms. This section describes the design of each service in this architecture. It also presents a case study including an experimental implementation for evaluating EA-SODIA and its algorithms.

Firstly a combined design method of each service is described. Then, the case study method is discussed, followed by the research questions and its hypotheses. The experimental implementation is presented with a short evaluation. The test data for the case study is also shown.

7.2 Overview of the Service Incorporation

This section illustrates how the services incorporate to conduct the processes introduced in chapter 4, 5, and 6. It is explicitly indicated which steps of a process are taken by an integrator service or a data service.

7.2.1 The Allocation of the Meta-database

Chapter 4 introduces the algorithm, RSMV, to establish mapping between the local schemas and the global schema. The result is the data in the meta-database that is regarded as a central conceptual database. In this architecture, the meta-database is allocated to both the integrator service and each DS. Figure 7-1 shows how the meta-database is managed by the services.

In a DIS, the meta-database contains the global schema (GS), the importing views, the global attribute domain (GAD), the organization list, and the organizational evolution list (OEL) that are defined formally in chapter 4. The importing views involve the importing views of all the DSs. Therefore, each DS needs to register its importing

views into each integrator service.

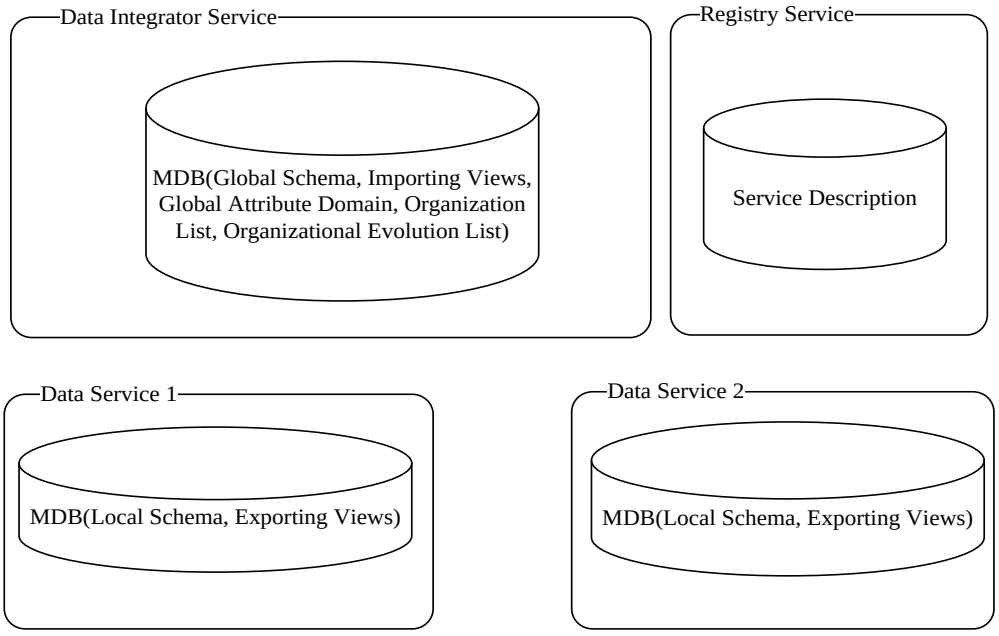

Figure 7-1 Allocation of the Meta-database

The meta-database in each DS stores its own local schema (LS) and exporting views that are also defined in chapter 4. The exporting views refer to the importing views of the DS in the meta-database at the DIS site.

This is slightly different from the conceptual meta-database defined in chapter 4, as each DS contains its own local schema and exporting views. Therefore, the local schema list (LSL) in the conceptual meta-database is in fact the union of the local schemas of all the DSs. In addition, the mapping and the entire mapping list (MPS) are also segmented. The mapping of a local schema can be composed by the importing views that are stored at the DIS and the exporting views that are stored at the DS. In order to indicate the relationship between the importing view and the local schema, the definition importing view is extended to have another property: *name* which is a string. The *name* represents the DS to which the importing view belongs. The MPS is apparently the union of the mappings of all the DSs.

The registry service contains the service description of each service based on WSDL,

including the information such as the name of the service, and URL. The source database providers need to publish their DSs into the registry service.

7.2.2 Query Processing

Chapter 6 introduces the query processing that involves four steps: Query Reformulation and Query Decomposition and Query Transformation and Result Composition. Query processing is a major issue in distributed database system and data integration, leading to much research. In this research, the focus rather is on establishing a data integration architecture that is easy to evolve. Therefore, only the parts relevant to the evolution purpose are addressed. Figure 7-2 shows how the services cooperate to conduct the query processing introduced in chapter 6.

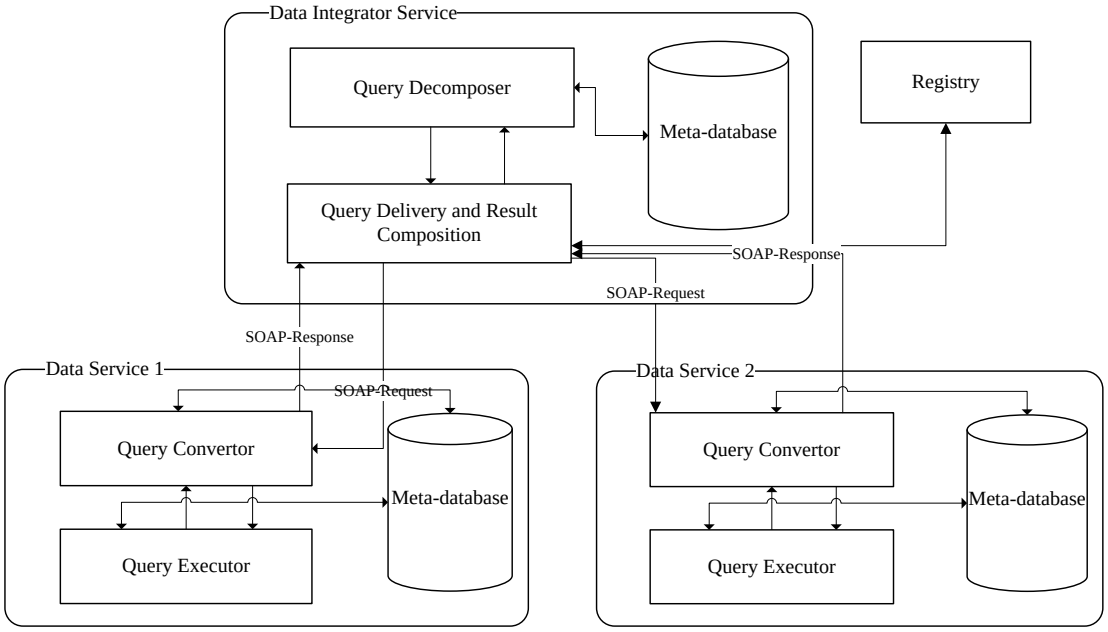

Figure 7-2 Query processing by services

The DIS involves the following components:

- Query Decomposer: This takes the query reformulation step which identitfies the scope of source databases (DSs), and finds the relevant DSs, and then decomposes the query into subqueries that are in terms of importing views of the DSs through the query containment test.
- Query Delivery and Result Composition: This takes the steps: Query

Decomposition and Result Composition. It further divides queries received from the query decomposer into subqueries each of which contains only one importing view representing a single source database (DS). Sequentially, it accesses the registry service to obtain the information to access the relevant DSs, and delivers those subqueries to the relevant DSs where the subqueries are further processed. Finally, it composes the results sent back from each DS to produce the result for the user.

The DS involves the following components:

- Query Convertor: This takes the Query Transformation step. Recall that the subqueries which are sent to DSs to be executed, consist of an exporting view (referred to as importing view at the DIS). Therefore, this step transforms the subqueries sent from the DIS into queries that are in terms of the local schema.
- Query Executor: This executes the reformulated queries and produces a result that is then sent back to the data integrator.

Although there are two DSs and one DIS in Figure 2 due to the space, in practice there could be much more of them.

7.2.3 Schema Evolution Detection

It has been shown in Figure 4 in chapter 3 that both the DISs and the DSs have a component, called Schema Evolution Detection. A system based on our architecture performs as follows:

1) When the administrator of a source database changes the local schema, he or she needs to invoke the schema evolution detection of the relevant DS.
2) The schema evolution detection of the DS changes the data (exporting views) in its meta-database, and examines whether the relevant importing views need to be changed as well. If it does, it accesses the registry service to find all the DISs and invokes the schema evolution detection of them.
3) The schema evolution detection of each DIS changes the relevant importing

views based on the request from that DS.

To sum up, the schema evolution detection of a DS launches a schema evolution detection activity. It changes the exporting views in the meta-database of the DS. The schema evolution detection of a DIS changes the importing views if requested by the DS. The administrator of the source service needs to change the local schema in the meta-database as well to keep it consistent with the schema in the DBMS of the source database. In future work, a software tool can be provided to change a local schema and then trigger the schema evolution detection function of a DS.

7.3 Service Design

As the architecture of this research is service-oriented and aligned with Web services, the service-oriented analysis and design method introduced in [90] is used to design the system. The introduction to the services in both chapter 3 and the previous section of this chapter so far describes the process steps (or application logic) of the services. It is generally referred to as the analysis of service in the service-orientated method. A service (service provider) in the system usually has one or more operations that have input and output to communicate with service requestors. Therefore, during the design stage, some processes are combined to become a single operation, considering the general features of the services, such as *autonomy* and *reusability*. The result is the abstract definition of the WSDL of each service in which the following parts are formally defined:

- definition of all service operations
- definition of each operation's input and output messages
- definition of associated XSD schema types used to represent message payloads

7.3.1 Design of Data Integrator Service

As described previously, a DIS involves three process steps which are *query decomposition, query delivery and result composition* and *schema evolution detection*.

During the design stage, the query delivery and result composition is combined with the query decomposition to produce an operation, called *query perform*. As the communication between services is through SOAP, the data types of input and output messages of each operation are defined based on XSD schema. Table 7-1 shows the result of the design of a DIS.

Table 7-1 the result of the design of a data integrator service

Operation	Input (Request Message)		Output(Response Message)	
	Message Name	Type	Message Name	Type
QueryPerform	QueryRequest	xsd:String	QueryResult	WebRowSet
SchemaEvolutionDetection	SEDRequest	xsd:complexType	SEDResponse	xsd:String

A DIS is designed to have three operations:

- QueryPerform: This is produced by combining the *query decomposition* and the *query delivery and result composition* processes. By applying the principles of a service which are mainly reusability and autonomy, it is observed that the query delivery and result composition are dependent on the output of the query decomposition. Therefore, other operations are not likely to access the query delivery and result composition process individually. The input and output messages of the QueryPerform are also defined in XSD schema. The input message is an xsd:string, which is a primitive data type in XSD schema, representing a conjunctive query raised by a user. The output message is a WebRowSet that is an importing complex data type used to store the result of a query. The WebRowSet will be introduced later in this section.
- SchemaEvolutionDetection: This operation is derived directly from the *Schema Evolution Detection* process. The input message is a complex type in XSD schema which is defined as following:

<xsd:schema xmlns:xsd=http://www.w3.org/2001/XMLSchema
targetNamepace=http://www.xmltc.com/dur/di/schema/dis/>

```
<xsd:element name=”SEDRequestType”>
    <xsd:complexType>
        <xsd:sequence>
            <xsd:element name=”ServiceName” type=”xsd:string”/>
            <xsd:element name=”ViewName” type=”xsd:string”/>
            <xsd:element name=”ModifyAction” type=”xsd:string”/>
        </xsd:sequence>
    </xsd:complexType>
</xsd:element>
</xsd:schema>
```

This input message is comprised of three elements: ServiceName, ViewName and ModifyAction. The ServiceName indicates the DS whose importing view requires modification. The ViewName indicates the import view that is to be modified. The ModifyAction represents the modification to be applied to the importing view. There are actions that can be recognized by the DIS:

1) “Discard”: means that the relevant importing view needs to be discarded.
2) “Delete <attribute name>”: means that the designated attribute needs to be deleted from the importing view. The <attribute name> will be replaced by the actual attribute name in practice.

The output message is also a string that represents the status of the execution of this operation (e.g. “successful” or “fail”).

One of the problems of a data-intensive service is transmitting the result of the query in SOAP between services. As the result of a query cannot be encapsulated into SOAP directly, it requires transformation. Therefore, the result of a query is translated into WebRowSet before being encapsulated in the body of SOAP message. The standard WebRowSet XML schema definition is available at the following URI:

http://java.sun.com/xml/ns/jdbc/webrowset.xsd

7.3.2 Design of Data Service

A DS undertakes three processes: *query transformation* and *query execution* and schema evolution detection. The query transformation and query execution are combined to produce a single operation, QueryPerform. The reason for the

combination is the same as that of the combination of the query decomposition and the query delivery and result composition of the DIS. Table 7-2 shows the result of the design of a DS.

Table 7-2 the result of the design of a data service

Operation	Input (Request Message)		Output(Response Message)	
	Message Name	Type	Message Name	Type
QueryPerform	QueryRequest	xsd:String	QueryResult	WebRowSet
SchemaEvolutionDetection	SEDRequest	xsd:complexType	SEDResponse	xsd:String

- QueryPerform: This is similar to the QueryPerform operation of a DIS. The difference is that the input message of a DS represents the subquery sent from a DIS. The QueryPerform of a DS is most likely to be invoked by the QueryPerform operation of a DIS, while the QueryPerform operation of a DIS is usually invoked by a user.
- ScehmaEvolutionDetection: This modifies the exporting views in the meta-database of a DS in response to a schema evolution. The input message is a complex type that is defined in XSD schema as following:

```
<xsd:schema xmlns:xsd=http://www.w3.org/2001/XMLSchema
    targetNamepace=http://www.xmltc.com/dur/di/schema/ds/>
    <xsd:element name=”SEDRequestType”>
        <xsd:complexType>
            <xsd:sequence>
               <xsd:element name=”EvolutionType” type=”xsd:string”/>
               <xsd:element name=”SchemaEvolution” type=”xsd:string”/>
            </xsd:sequence>
        </xsd:complexType>
    </xsd:element>
</xsd:schema>
```

The first element of the input message is a string representing the type of the schema evolution (e.g. “Attribute Removal”), while the second element of the input message is a string that represents the schema evolution operation (e.g. “(r1,

a)"). The output message is also a string that represents the status of the execution of this operation (e.g. "successful" or "fail").

7.4 Case Study

In order to evaluate EA-SODIA and the approaches in this research, a single case study [93,94,95] has been conducted. The case study is one of the research methods which has been commonly used in such application areas as psychology, sociology, political science, anthropology, social work, business, education, nursing, and community planning. Although there is little formal documentation available on how to conduct a proper case study, [93] presents guidance on when and how to conduct a case study to evaluate software methods or tools.

It is unrealistic to implement the complete data integration system based on EA-SODIA in a single research by a single researcher, as it may involve several research issues some of which are beyond the scope of this research, such as translation between queries language and query optimization. Therefore, an experimental implementation of EA-SODIA is carried out to undertake the case study for evaluation of this work. The methods presented in previous sections are embodied in the experimental system.

7.4.1 Context and Analysis Unit

The objective of the case study is to evaluate EA-SODIA and the approaches described in previous chapters. It examines whether a system based on EA-SODIA is able to solve the evolution problems defined in this thesis while eliminating the heterogeneities among involved, distributed and autonomous databases. Therefore, we developed an experimental system based on EA-SODIA in the automobile trade application area. For convenience, the experimental system is called EA-SODIAS. The case study assumes that, in the automobile trade application area, there are various organizations which can provide partial information about an automobile

which is either new or used. Each organization publishes its database as a DS and registers the DS into the DISs.

Thus, the context and analysis unit of the case study is as following:

- Context of the case study: automobile trade application area
- Unit of analysis: the experimental implementation system (EA-SODIAS).

It can be seen that this case study is a single project case study with a single analysis unit. EA-SODIAS is described in detail later in this chapter.

7.4.2 Question and Hypothesis

In order for the case study to be effective, the research questions and the hypotheses of the case study must be clearly defined in advance. Since it is proposed in this research that EA-SODIA is able to dynamically integrate heterogeneous and distributed source databases aiming to minimize the cost of the maintenance caused by database evolution, the main research questions of the case study are:

1) How and why the RSMV approach can integrate distributed databases, eliminating the heterogeneities defined in Chapter 1.
2) How and why the RSMV and the meta-database can help solve the evolution problems defined in this thesis.
3) How and why SED can help solve the evolution problems defined in this thesis.
4) How and why SOA can help solve the evolution problems defined in this thesis.

The hypotheses of the case study are then:

A. The heterogeneities defined in Chapter 1 can be eliminated using the RSMV approach and the query processor.
B. The RSMV approach and meta-database can reduce the cost of modification work caused by schema evolution, and the query processor can reduce the

number of the queries which require modification when any organizational evolution occurs.

C. If any schema evolution occurs in one source database, the views of other source databases do not require modification so that the system can still work properly.

D. The SED can reduce the cost of modification work caused by schema evolution.

E. SOA and web services can help reduce the cost caused by database evolutions and system evolution because they provide high reusability, autonomy and discoverability.

By setting the hypotheses, some response variables are also listed in Table 7-3

Table 7-3 Response Variable

Response Variable	Description
Number of user queries explicitly designating source databases	The number of user queries in which the actual names of particular local schemas are included
Number of user queries involving local schema	The number of user queries in which the actual relations and attributes of particular schemas are designated
Correctness of the data retrieved by a user query	The correctness of the results of a user query
Number of the affected user queries	Number of the existing user queries which are affected by a schema evolution.
Human invention	Whether an automatic view modification process require human invention
Number of hard-coded queries	The number of queries which are used to integrate local schemas and to eliminate heterogeneity

Number of views requiring modification	Number of exporting views and importing views which require modification when a schema evolution occurs
Number of source database considered	Number of source databases which need to be considered when a single schema evolution occurs
The work of identifying the affected views	The effort to find the views affected by a schema evolution.
The work of identifying the affected DISs	The effort to find the DISs which require modification when a schema evolution occurs
The work of modification on the views	The effort to modification the views affected by a schema evolution

In general, the case study will focus on answering the above research questions and demonstrate the above hypotheses. Therefore, if the hypotheses are well supported by the results of the case study, EA-SODIA and the approaches proposed in this thesis are generally considered to be successful and the aim of this research has been achieved. However, in order for the evaluation to be more complete, other aspects of the system such as performance and scalability and reliability will also be examined and discussed in Chapter 8.

7.4.3 Experimental Implementation

As mentioned in previous sections, the experimental implementation system (EA-SODIAS) implements EA-SODIA to integrate various source databases in an automobile trade domain. EA-SODIAS implements the software components of the DIS and the DS and the registry. This section introduces the development environment and the design of EA-SODIAS in detail.

7.4.3.1 Development and Test Environment

The experimental system runs on Microsoft Windows XP operating system. The development was undertaken in the following environment:

- Programming Platform: The Java 2 Platform, Enterprise Edition (J2EE)
- Development Tool: Java Studio 8.0
- Web Container: Apache Tomcat 5.0
- Web Service Toolkits: Apache Axis
- Database Management System: MySQL

The Java 2 Platform is a development and runtime environment based on the Java programming language. The Java 2 Platform Enterprise Edition (J2EE) is built to support large-scale, distributed solutions. The J2EE was chosen for several reasons:

- It is currently the major programming language supporting Web services, providing the feature of platform independence.
- The J2EE is one of the two primary platforms currently being used to develop enterprise solutions using Web services.
- The J2EE platform provides a development and runtime environment (APIs) through which all primitive SOA characteristics (e.g. WSDL and SOAP) can be realized.
- Much like the Web services specification landscape, the J2EE platform consists of a series of technologies that are based on open standards. This allows vendors to build proprietary tools and server platforms around a standardized foundation (e.g. Tomcat and Axis).
- Database connectivity is very well supported.
- The integrated development environment, Java Studio 8.0, supports easy testing and debugging.

Apache Axis is an implementation of the SOAP and has proven itself to be a reliable and stable base on which to implement Java Web services. It was planned to use Apache jUDDI to implement the registry service. Apache jUDDI is an open source

java implementation of the UDDI specification for Web services. The registry published by jUDDI is itself a service, providing service requestors with some technical information such as the URIs and the names of methods of service providers. For the time being, the search for proper service providers and the binding between services is still manual work. However, in this research, the access methods of the DISs and the DSs are standard and unified. Therefore, each integrator knows how to access the DS, while each DS knows how to access the DIS. Thus, the registry is only used to record the URIs of the services. Instead of using Apache jUDDI, a simple database is chosen as a registry to achieve our goal. The database is published as a DS as well. This is further described in later sections.

In addition, the experimental system was tested under following hardware conditions:

- Number of Personal Computers (PCs): 2 (with same capacity).
- CPU: Pentium 4 (3.20 GHz).
- Memory: 1.5GB.
- Hard-Disk: 320GB.

These two PCs were connected through a local network which was an ethernet running at 10 mega bits per second (10Mbps). One of the PCs was used to publish all the DSs, while the other one was used to publish all the DISs and the registry service.

7.4.3.2 Experimental System Architecture

In previous sections, the service-oriented design of the system was introduced. However, in practice, the object-oriented design method [90] is usually used to design the internal components and process of a web service. Figure 7-3 shows the architecture of EA-SODIAS in terms of its services and databases and the clients. The diagram shows the internal classes and process of the DIS and the DS and omits the input and output of each service and messages transferred between services.

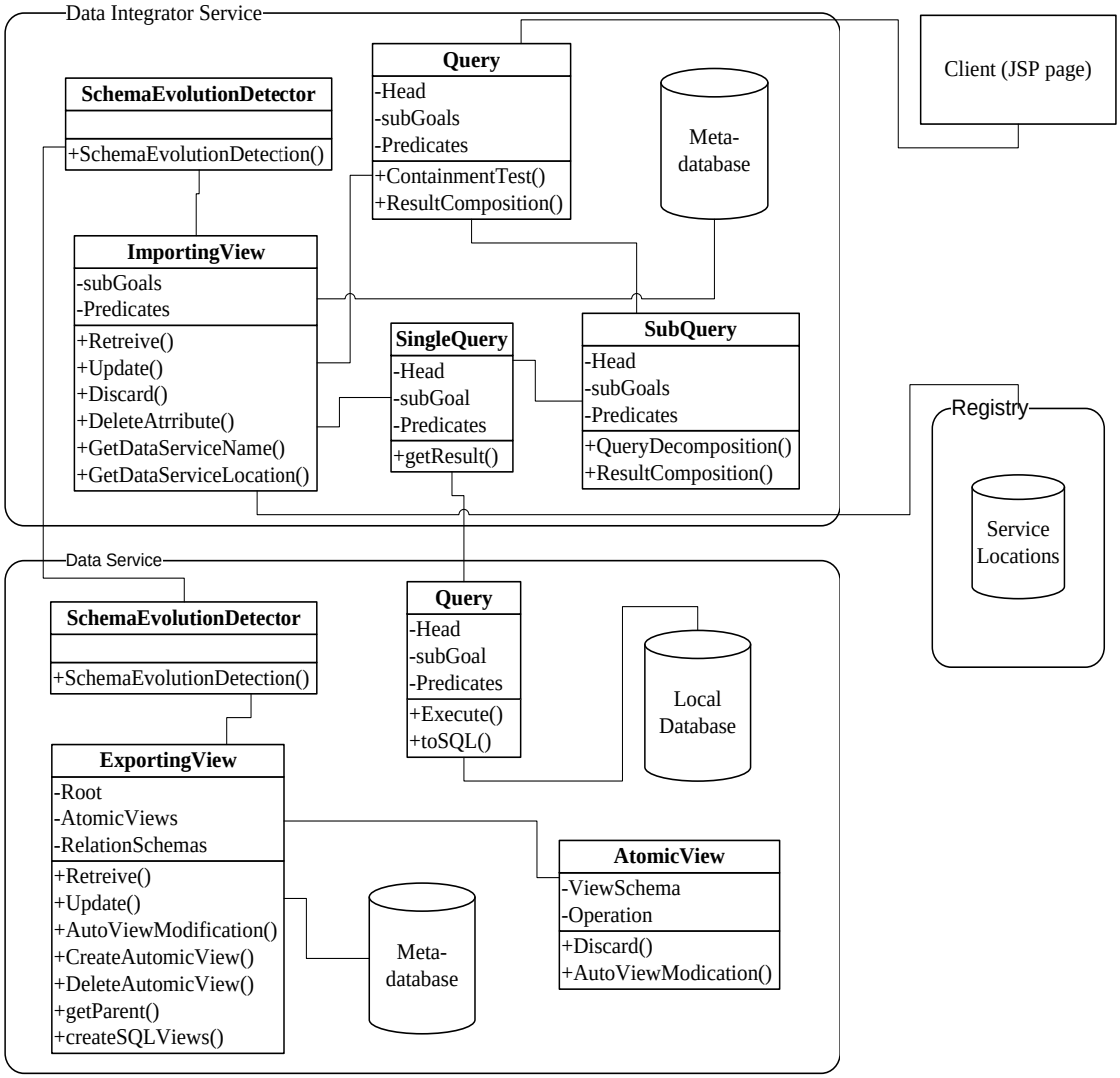

Figure 7-3 Architecture of EA-SODIAS, Showing the Classes of each service and the Databases

This architecture reflects the design of the services in EA-SODIA. In the DIS, the Query and ImportingView and SubQuery and SingleQuery classes encapsulate the query reformulation step and query decomposition and result composition, while the SchemaEvolutionDetector class and ImportingView encapsulate the schema evolution detection in the DIS. In the DS, the Query class encapsulates query transformation step and also executes the reformulated query on the local database, while the SchemaEvolutionDetector and ExportingView and AtomicView classes encapsulate the schema evolution detection step performed in the DS.

In the implementation, a simple JSP web page client is developed to accept user queries and show results obtained from the DIS. The meta-databases represent the meta-databases in the DIS and the DS. The local database in Figure 7-3 represents the actual DBMS of the source database. As mentioned previously, the registry service is in fact another DS that explores a database storing the locations of the services in the system.

The design of EA-SODIAS has also been influenced by a number of considerations:

- Optimizing the formulation at the DS site: not only save the definitions of the views, but also build actual views in the local databases. Like program compilation, the views need to be re-built every time when views definitions change as the result of the schema evolution.
- Reducing the communication across network: the registry only stores the locations of the services.
- Providing high reusability of the service: all the DSs have the unified method; all receive SQL as a parameter, when publishing new databases, no programs are required.

7.4.3.3 Meta-database Structure and Management

The meta-database is stored and managed as a MySQL 5.0 relational database constructed using the MySQL Administrator tool. The meta-database is divided into two parts as described in section 7.2.1. One of them is located and managed at the integrator service site, while another one is at the DS site. Figure 7-4 shows the structure which reflects the meta-database located in the DIS. The structure of the meta-database which is located at the DS site is depicted in Figure 7-5.

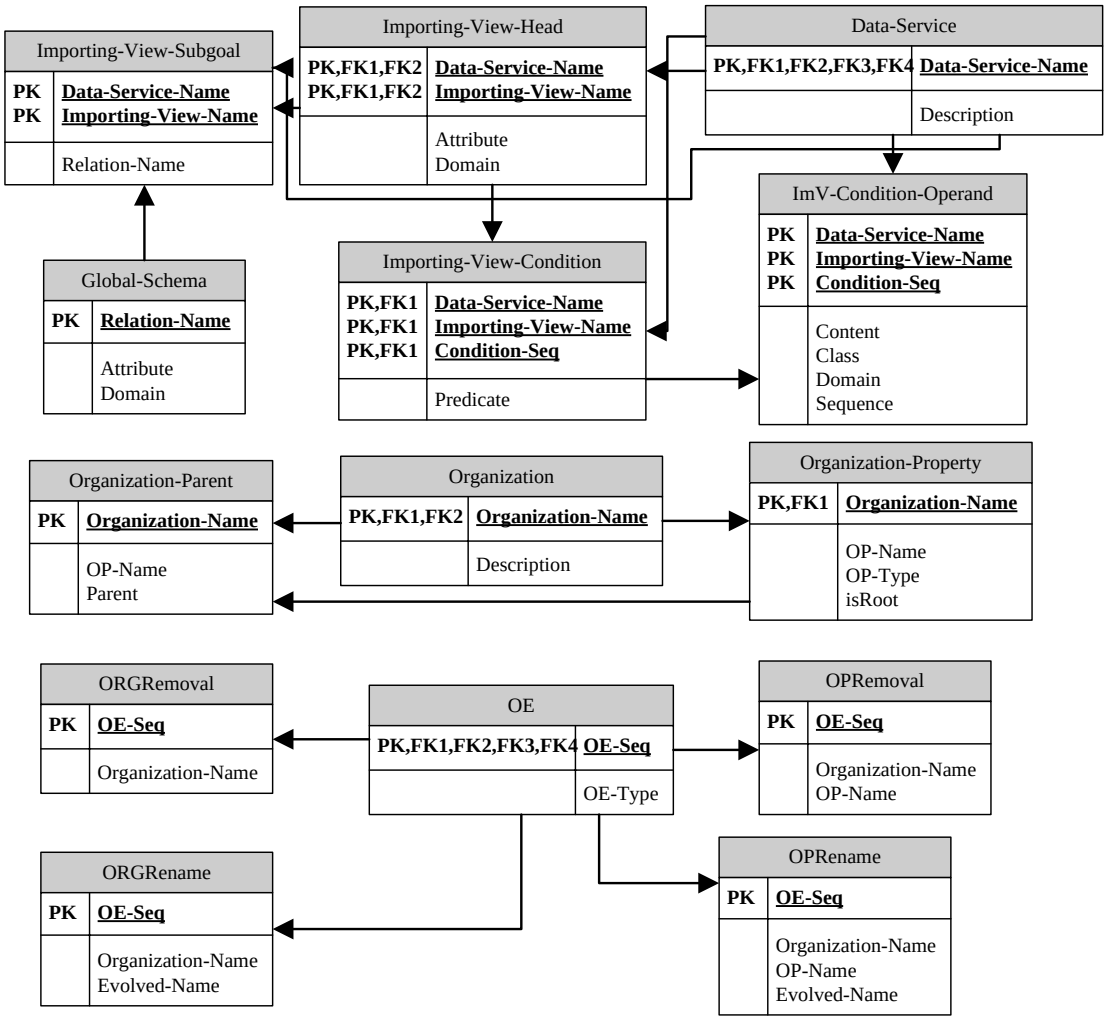

Figure 7-4 The Structure of the meta-database of Data Integrator Service

Although it is not shown in Figure 7-3, a small software tool called the Metadata Creator is provided in order for the investigator to build exporting views and importing views. One of the advantages of the RSMV is that views are easier to manipulate than hard-coded programs. Without the software tool, novice maintainers have to understand the structure of the meta-database in order to store the view definition manual. The software tool is not aiming to automatically build the views based on the local schema. It is more like an editor and compiler of the views, accepting the view definitions from a maintainer and storing them in the meta-database properly. A maintainer can enter all the atomic views of an exporting view or the subgoals of an importing view on the interface of the Metadata Creator

which sequentially stores them in the meta-database. The validity of the views has to be checked manually.

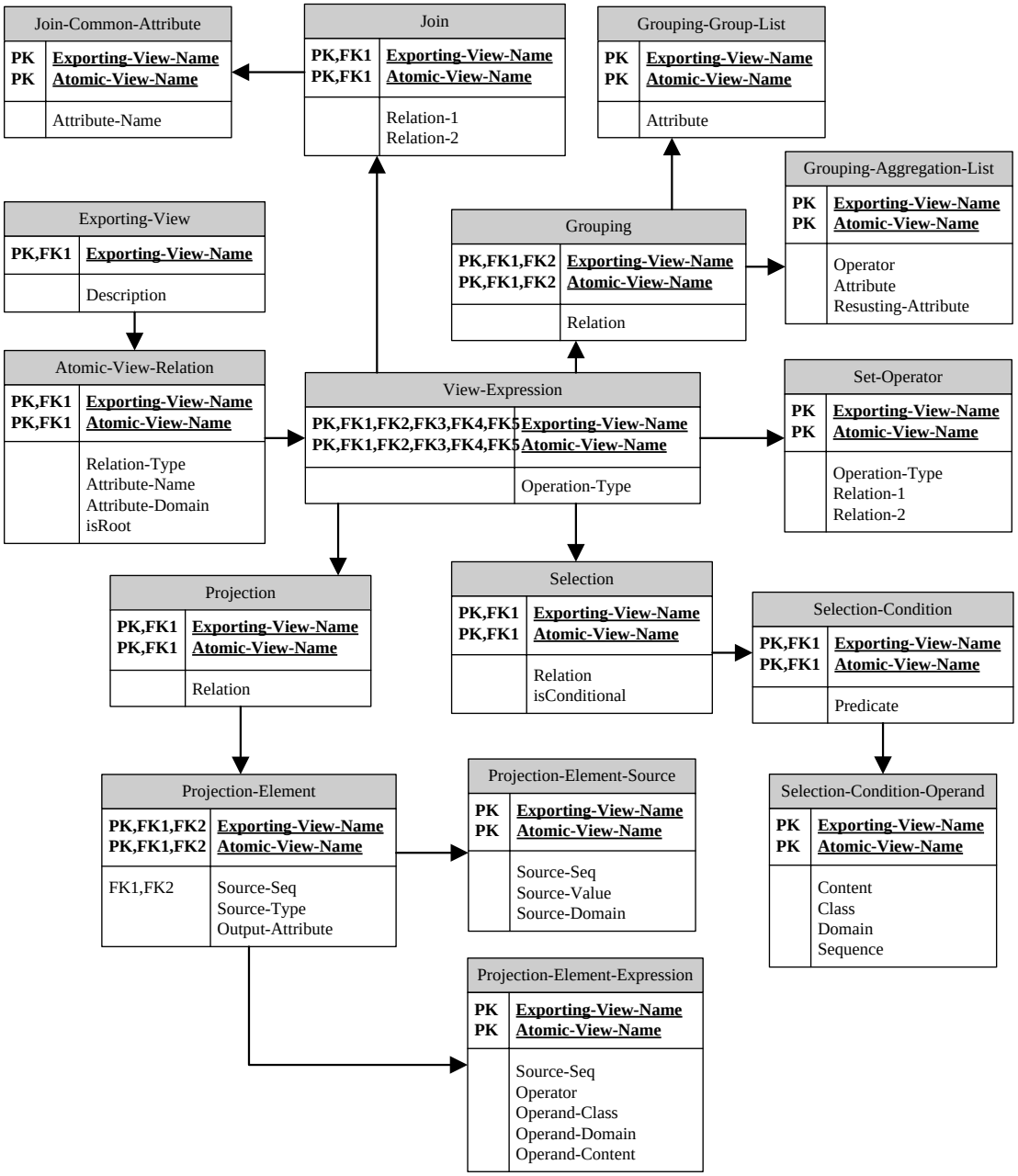

Figure 7-5 The Structure of the meta-database of Data Service

Although the Metadata Creator can help build a new exporting view or a new importing view for the first time, the meta-database is managed manually using MySQL Administrator. If the system was released for large-scale use this would need to be rectified, however the current situation is acceptable for case study research.

7.4.3.4 Data Integrator Service

The DIS shown in Figure 7-4 can be referred to as a software component which can be published as a web service. The system may have more than one DISs which all have exactly the same internal processes. The following classes are designed:

- The query class represents a user query, providing methods to perform query containment test and result composition. It will be created when the user query is received from the client, the JSP web page in this case. It also encapsulates the processes of identifying the possible source databases and dealing with the organizational evolutions.
- The ImportingView class represents the importing views stored in the meta-database, responsible for retrieving and updating an importing view in the meta-database.
- The ImportingView class also provides methods to get the name and location of the DS of the importing view by accessing the registry service.
- The SubQuery class represents the subqueries resulting from the query containment test process, providing methods to further decompose a subquery into single queries and compose the results for the current subquery.
- The SingleQuery class represents the single queries finally produced from the containment test, providing methods to get the required data from a relevant DS. Therefore, the SingleQuery is the class which communicates with the relevant service.
- The SchemaEvolutionDetector class provides the method for the schema evolution detection step. It will be invoked by a DS when any schema evolution that affects the importing views occurs. Then, it creates an ImportingView object to undertake the retrieving or the updating of the affected importing view.

The DIS is implemented by modifying an existing software product, OGSA-DAI WSI 2.1 [31]. OGSA-DAI is a middleware product which supports the exposure of data

resources, such as relational or XML databases. It is a free, open source software available on the web [31]. OGSA-DAI enables users to expose a relational database as a DS which can receive SQL queries in a string from a client and execute the queries on the exposed database and then send the result back to the client. A DS published using OGSA-DAI provides various activities such as retrieving and updating the underlying database. All the OGSA-DAI DS have a unified method, called Perform, which receives an XML document as a parameter in which the user query (in SQL) and the designated activity are encapsulated.

The OGSA-DAI can be extended by adding new activities, which are in fact Java classes, to perform new functions. Therefore, the DIS is implemented by adding two new activities, called the Query and the SchemaEvolutionDetector, which perform the tasks of the Query class and the SchemaEvolutionDetector class, respectively. Thus, the user queries encapsulated in the perform document is in a conjunctive query language instead of SQL. Other classes introduced above are invoked by these two activities.

Rather than implementing a complete native version of the DIS, it was designed to use the OGSA-DAI and modify it to embed the processes of the DIS for the following reasons:

- It has the advantage of using a proven implementation. It is important to establish the reliability of a third-party implementation. Establishing confidence in the OGSA-DAI was achieved by experimenting with various SQL queries on pre-defined data to successfully produce predicted results. In addition, the OGSA-DAI has been used in a variety of research projects with no reported problems.
- The connection with various DBMSs products is well supported by the OGSA-DAI. A DS needs to connect to the underlying DBMS whenever the user query is received. The OGSA-DAI provides XML document to store the connection information so that it is easy to maintain.

- The transformation between the results of queries and the SOAP document is well supported by the OGSA-DAI. The result of a query in Java is usually stored in a object called the ResultSet. However, the messages between web services are encapsulated in SOAP documents. The OGSA-DAI provides a well-optimized method to convert the ResultSet to the WebRowSet format before being sent back to the client in the body of a SOAP message.
- The management of configuration XML files of both services and connection of the exposed database is well supported by the OGSA-DAI.
- A set of software tools to publish DSs are provided, which can reduce the manual work.

Since the case study aims to examine whether EA-SODIA can solve the evolution problems, the parallel query process is not considered and implemented in this research. Therefore, the relevant DSs are accessed in sequence, which may affect the performance of this experimental system. In addition, the Bucket Algorithm is implemented without consideration of optimization, because there is no proven existing implementation available. After the first step of the Bucket Algorithm, the possibly relevant importing views are found. Each importing view found is referred to as a single query which is immediately sent to the relevant DS to get results. The results are stored in temporary tables which are created based on the importing views in the meta-database at the DIS site. Namely, the relevant importing views are materialized before the second step of the Bucket Algorithm. Once the contained queries have been found in the second step, they are executed on those temporary tables to obtain the results. Although this may increase the amount of data transferred every time, it avoids repeatedly accessing the same DSs. More importantly, it enables the investigator to check whether each DS is able to provide the correct answer, especially when some schema evolution occurs.

7.4.3.5 Data Service

As with the DIS, each DS in this system has the same internal structure shown in

Figure 7-4. The following classes are designed:

- The Query class represents the single conjunctive query received from a DIS. It translates the single query into a SQL query and then executes it on the underlying local database. Since the exporting views have been created in the local databases by the Metadata Creator as mentioned previously, the query class does not perform the query transformation step introduced in Chapter 6.
- The ExportingView class represents an exporting view, providing methods for retrieving and updating the exporting view, and for manipulating its atomic views. It also provides method for automatic view modification.
- The AtomicView represents an atomic view, providing methods for discarding and automatically modifying the atomic view.
- The SchemaEvolutionDetector class provides the method for the schema evolution detection step using the methods described in Chapter 5. It finds the affected exporting views based on the schema evolution operation provided by the user. This class will create the instances of the exporting view class to retrieve and modify those exporting views. If any atomic view of the exporting view has been modified, the corresponding views in the local database are re-built as well.

The method used in the SchemaEvolutionDetector to re-build the views in the local database is the same as that of the metadata creator. The complete translation between a query in SQL and a query in relational algebra would be very complex and therefore not suitable for this single case study. Therefore, each atomic view is translated into the view in SQL individually and then stored in the local database. As an atomic view of an exporting view only involves a relational algebra expression that has one operation, the translation becomes much easier to undertake. It means that for each atomic view of an exporting view, there is a view that has the same name as it involves an SQL query in the local database. Thus, the root atomic view is the view where the query received from the DIS is performed.

The Query class receives a conjunctive query from the DIS. It is required that this conjunctive query is translated into an SQL query, in order to execute on the local database. By using the architecture and the methods in this research, the conjunctive query sent to the Query class only involves one subgoal. Although it may have one or more predicates, it is simple to translate this conjunctive query into a "select-from-where" query with one relation.

As with the DIS, the DS is also implemented by modifying the OGSA-DAI WSI 2.1. This is also acomplished by adding two new activities, called the Query and the SchemaEvolutionDetector, which perform the tasks of the Query class and the SchemaEvolutionDetector class, respectively. Again, the user queries encapsulated in the *perform* document are in a conjunctive query language instead of SQL. Other classes introduced above are invoked by these two activities.

7.4.3.6 Registry Service

As explained previously, a DS exposing a simple database is used as a registry service. The database has only one relation called Service-Location where the locations of all the services in this system are stored. The DS is deployed and published using OGSA-DAI without any modification. Therefore, it provides one method which receives a string representing an SQL query as a parameter. In this system, all the service has to know is the location of this registry service in advance. This is achieved by adding a simple relation with one tuple into the meta-database to record the location of the registry service. However, the method provided by the registry service and the relation storing the locations are encapsulated in the hard-coded program of each service.

7.4.4 Test Data

Experimental implementation is used to examine whether EA-SODIA and its methods can solve the evolution problems defined in this thesis while eliminating the heterogeneities between databases. Therefore, building the implementation alone does

not provide any evidence to prove the hypotheses of the case study. It is essential to apply this implementation to a particular application domain where the heterogeneities and the evolutions defined in this research occur. Consequently, a well-designed set of data is required.

7.4.4.1 Principles of Test Data

In order to obtain values for the variables to examine whether the hypotheses can be supported, the following test data need to be included:

- The information in an application domain: the general information which may be recorded and manipulated in databases. This is standard information which can be understood by any organization in the same domain. As described in Chapter 1, this information can be represented as an E-R model.
- The data in the relational model: these are different sets of data which are designed differently based on the general information in E-R model. All the heterogeneities defined in Chapter 1 need to be covered among these sets of data. In practice, each set of data is designed independently by each different organization for its own purpose. Although the query process is not the focus of this research, it has been taken into account that the data sets should be designed to examine whether the query process method in this research can find the relevant DSs correctly.
- Various types of evolution: the types of evolution defined in this research are covered. Not only are the evolution operations designed to perform on the system, but a plan is devised to perform each evolution on different databases to cover various possible cases. Without this plan, the test cannot be repeatedly performed with the same results.
- A set of user queries: a set of user queries are designed to examine whether the relevant DSs can be located and whether the correct results can be produced, especially after each evolution occurs.

In addition to the temporary relations for storing the resulting data from each DS as

mentioned in the previous section, some log files which are plain text files are used to record the outputs of each process of the system. The results are measured and compared with the pre-defined “correct” answers. The test data are set using SQL scripts so that they can be repeated easily. Entering the user queries to the system and performing those evolution operations are manual work. Also, measurement and comparison of the results are conducted manually due to the nature of the evolution problems.

7.4.4.2 Application Domain

The application domain in this case study is the automobile trade industry in China. Both new cars and used cars are being sold by different organizations or persons. The organization may include dealers and garages. Personal owners may leave their cars in a dealer or garage for sale. Those organizations may locate at different cities which belong to different provinces in China. Each organization holds its owner database in which the car sale information is stored. In addition to the above organizations, other organizations such as insurance companies and register centres may provide relevant information about used cars. EA-SODIAS aims at enabling users to find information from various organizations. Although this case is much simpler than a practical case, it is enough for answering the questions of the case study.

7.4.4.3 Design of Test Data

The information being managed in this application domain is about automobiles. It is assumed that every organization in this domain understands this information. Therefore, organizations’ own application systems store this information in different ways. The E-R model of the general information of this application domain is shown in Figure 7-6 and Figure 7-7.

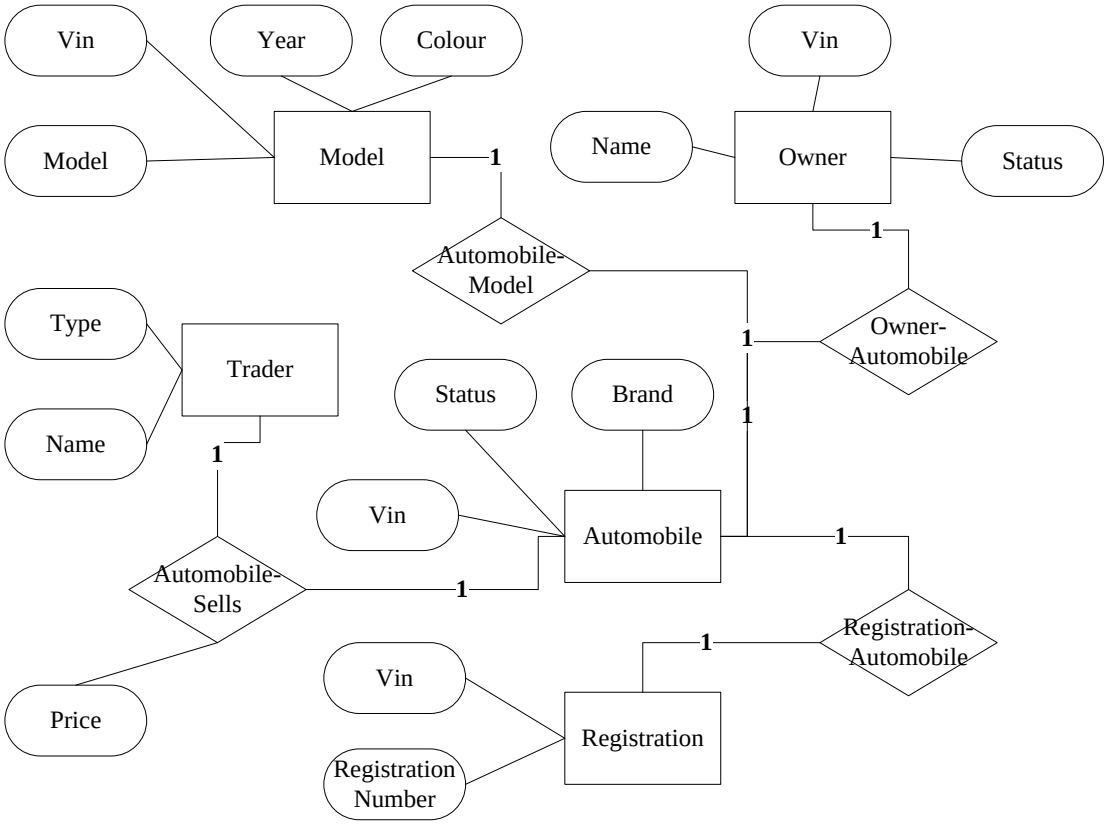

Figure 7-6 E-R Model for EA-SODIAS

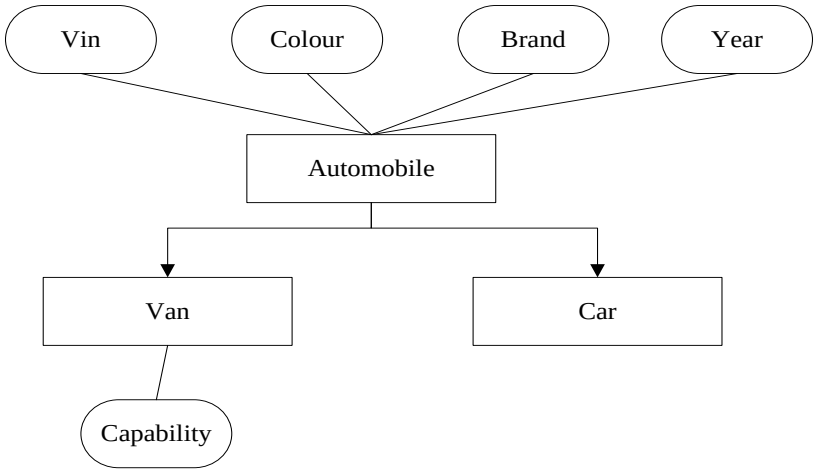

Figure 7-7 Subclasses of Automobile Entity

It is shown in Figure 7-6 that the information regarding an automobile may involve its model, registration, traders and previous owners (for a used car). This information is represented by five entities and four relationships in this E-R model. Therefore, the

E-R model indicates that an automobile of a brand may have a vehicle identification number which uniquely identifies the automobile. An automobile may also be new or used, and be an automatic or manual one represented by the model entity. In addition, either a new or a used automobile can be sold by a dealer or a person, represented by the trader entity. A used automobile also has registration information, represented by the registration entity. Finally, Figure 7-7 shows that the automobile has two subclasses, indicating that an automobile can be either a van or a car. A van has a attribute capability which is not a attribute of a car. Apparently, this E-R model is simple and only includes some of information which is to be managed in practice. However, this is enough for a case study to evaluate a research, because it is enough to involve various heterogeneities.

Global schema

A global schema can be designed as follows, mapping the above E-R model.

AUTO (vin, status, brand, type)

VAN (vin, capability)

MODEL (vin, model, year, colour)

SELLS (name, type, vin, price)

Registration (vin, registration-number)

OWNER (vin, owner, status)

This design follows the rule introduced in Chapter 4 that the global schema is in a higher normal form and has no composite attributes.

Local schema

The same E-R model may be mapped into different relational models by different organizations. Therefore, this case study designs the following possible local schemas to cover all heterogeneities defined in Chapter 1.

1) VEHICLE *(vin, status, brand, type, cap, model, year)*

   SALE(name, type, vin, price)

CUR_OWNER (vin, owner)

2) *AUTOMOBILE (vi-n, status, brand, type)*

VAN (vi-n, capability)

MODEL (vi-n, transition, year, colour)

SELLS (trader_name, type, v-in, price, discount)

3) *CAR (vi_number, status, brand)*

VAN (vi_number, status, brand)

MODEL (vi_number, model, year, colour)

OWNER (vi_number, first_name, mid_name, last_name, price)

4) AUTO *(vin, status, brand, type, year)*

Registration (vin, number)

OWNER (vin, owner_name, status)

The first and second local schemas may be held by organizations such as the online car trade market where various sellers show their automobiles, providing actual traders' information. The third local schema may be designed by dealers or garages without holding trader information. The fourth local schema, however, may be designed by the registration center, providing registration information of used cars and no sale information about the cars. In this case study, four source databases are published as DSs initially to examine whether the heterogeneities and the evolution problems can be solved. Each of them has one of the local schemas above, respectively. Then, more source databases having the above local schemas are published to evaluate the scalability of the system. The heterogeneities existing among the above local schemas are further discussed in Chapter 8.

7.4.5 Evaluation of Implementation

This section discusses some of the issues arising from the implementation of EA-SODIA in EA-SODIAS. The EA-SODIA and methods themselves are evaluated in the next chapter.

7.4.5.1 Design Evaluation

7.4.5.1.1 Combined Design Approach

The design approach combining Object-Oriented and Service-Oriented methods has proved effective and the system architecture has not been changed for any version of EA-SODIA, although some of the methods of the classes have been changed over time. Using the service-oriented method can help to define the operations and the messages of the services correctly and reduce the modifications on them afterwards. The object-oriented method helped to design the internal process of each service clearly. As the system is developed using Java which is an object-oriented programming language, the object-oriented design method also made the programming easier. In addition, combining these two methods also helped to expand the system with minimal effort.

There were some disadvantages, in particular, the query processing. Both query reformulation step and result composition steps consider little optimization. The query reformulation step creates as many subqueries objects as the resulting conjunctions of the Bucket Algorithm, which may waste computing resources. However, it made the program elegant and easy to test. Also, all the relevant DSs were accessed and the results were stored in the temporary relations in the meta-database of the DIS. It may to some extent increase the cost of communication across network and the communication with DBMSs, when the number of source databases becomes very large or the amount of data in individual databases becomes huge. However, in this case study, it is not necessary to make the number of DSs extremely large, because we are not focussing on performance. Therefore, this method actually increased the performance, as it reduced the times of communicating with DSs.

The DSs were accessed in sequence, therefore, the parallel query processing problem was not considered. Java threads offer better performance of query processes, but the current method works satisfactorily for the case study. Moreover, no validation check on input data is provided. Therefore, it places responsibility on the investigator to make the input data valid. However, it was not a problem after a complete set of test data is produced.

7.4.5.1.2 Third-Party Software

There were good reasons for using the OGSA-DAI provided by the OGSA-DAI team, the most important being that the code can be trusted as correct (see section 7.5.3.4 and section 7.5.3.5 for research citing use of the OGSA-DAI). In addition, a substantial amount of time was saved by not handling the management of the underlying database, nor providing tools to publish DSs.

Another crucial reason is that the OGSA-DAI provides a method of converting a Java resultset into a Webrowset. It also provides a method to encapsulate the webrowset into a SOAP document. It certainly provides methods to convert repeatedly and to manipulate the results easily. This is complex and time-consuming work for a research with one researcher.

The OGSA-DAI provides good extendibility by allowing users to add new activities which are Java classes. This allows us to add our programs into it by simply modifying one of its activities and adding our Java classes into the package. Therefore, it also makes the EA-SODIAS extendible.

The main disadvantage of the OGSA-DAI is that it lacks documents for developers. Although it provides useful guidance for a user to install it and publish a DS, there are few documents showing the internal Java classes.

7.4.6 Test and Validation

Each class was tested individually before being included in the system. The classes were tested in sequence because each class takes an input which is the output of another. Each class program was slightly modified to gain high visibility of input and output data so that very few problems were found during integration. Individual programs were mostly checked by hand to ensure that the output generated was as expected, e.g. the relevant DSs found by the Query class and the subqueries were compared to a manually performed analysis. Also, the affected views resulting from the SchemaEvolutionDetector class were also compared to the results produced by hand.

The Query class and the ExportingView class were more complex and required the use of Java Studio's debugging tools. These allowed the internal state of various data structures to be displayed at appropriate points during the execution of the classes. SQL queries were also printed out to check the correctness before they were executed on the database. Single-step tracing of the routines was used to ensure correct implementation of the algorithms.

Due to its nature, the query containment test of the Query class cannot be tested using a large number of DSs, because producing the results by hand was very time-consuming. However, the experiments with typical test data are enough to ensure its correctness even with a large number of DSs.

7.5 Summary

This chapter has presented the design of the services in EA-SODIA. A case study including an experimental implementation is also described for evaluating the EA-SODIA and the algorithms in this research. Various technical issues relating to the system's design have been discussed. The test data of the case study is described in detail.

Chapter 8 presents an extensive evaluation of EA-SODIA, and the RSMV and the SED algorithms using the results of the case studies. The main focus is on the capability of solving the heterogeneity and the evolution problems, although various basic characteristics of EA-SODIA are also examined.

Chapter 8 Evaluation

8.1 Introduction

Chapter 7 described the design rationale of the services in EA-SODIA and a case study including an experimental implementation called EA-SODIAS.

Having shown the integration method RSMV and the schema evolution detection and the query processing in EA-SODIA, this chapter presents an extensive evaluation of all the methods introduced. It also shows how the service-based architecture can help with the evolution problems.

The evaluation begins with one of the most essential properties of EA-SODIA: the capability of eliminating heterogeneity. EA-SODIA is intended to solve some evolution problems on the premise of integrating distributed databases with heterogeneity. Therefore, it is important that the system is able to eliminate various heterogeneities defined in Chapter 1. The chapter then discusses the capability of solving evolution problems, which is the focus of this research. The issues relating to query processing are also discussed. These properties are examined by answering the research questions and demonstrating the hypotheses defined in Chapter 7. Finally, some general characteristics of EA-SODIA such as scalability and expandability and domain independence and language independence are also discussed.

8.2 Capability of Eliminating Heterogeneity

Recall that the motivation of the research is to provide an evolvable integrated database system to provide users with a unified view of various distributed databases. Therefore, in advance of dealing with the evolution problems, the heterogeneity problems must be solved.

The algorithm in this research dealing with the heterogeneity problems is called

RSMV. Databases with various heterogeneities defined in Chapter 1 should be integrated into EA-SODIAS successfully. In the case study, in order to examine the capability of eliminating heterogeneity, a research question was defined in Chapter 7:

> How and why the RSMV approach can integrate distributed databases, eliminating the heterogeneities defined in Chapter 1.

Also, three hypotheses were defined, as follows, in terms of the above questions.

8.2.1 Hypothesis A

The hypothesis is:

> The heterogeneities defined in Chapter 1 can be eliminated using the RSMV approach and the query processor.

This is examined by integrating four pre-designed databases into the system, mapping their schemas to the global schema. All heterogeneities exist among these four databases each of which is mapped to the global schema individually. In principle, if these typical databases can be integrated, there is no reason why a large number of databases with these heterogeneities cannot be integrated as each of them is mapped to the global schema separately. The integration of a database is regarded as a "success", if

- a set of valid exporting views are constructed based on the schema of the database using relational algebra so that the exporting views are homogeneous to the global schema (as explained in Chapter 4)
- and a set of corresponding importing views of the database can be constructed validly based on the global schema
- and the system is able to find the relevant databases based on the importing views
- and the values of relevant attributes can be obtained as long as the attributes are involved in the database

8.2.1.1 Test Data

In order to verify the capability of the RSMV algorithm for reconciling heterogeneity, an investigation was undertaken using EA-SODIAS and the pre-designed test data introduced in Chapter 7. Recall that there are four local schemas involving various heterogeneities and one global schema in the test data set. In this case study, the global schema is stored in the relation called Global-Schema in the meta-database residing in the DIS. Four source databases, D1 and D2 and D3 and D4, were built, each of which holds one of the local schemas respectively. The source databases were then published as DSs. The meta-database of each DS resided in the source database (local schema) because this saved some work from connecting to a separate database and made no difference to the results.

At this stage, only one DIS was built because it was enough for evaluating the capability of solving the heterogeneity problems. The global schema that stored at the DIS is shown in Table 8-1:

Table 8-1 Content of the Global-Schema Relation in the Meta-Database of the Data Integrator Service

Relation-Name	Attribute	Domain	Remark
AUTO	vin	String	Vehicle Identification Number
AUTO	status	String	“New” or “Used”
AUTO	brand	String	
AUTO	type	String	“Car” or “Van”
VAN	vin	String	
VAN	capability	Double	
MODEL	vin	String	
MODEL	model	String	“Automatic” or “Manual”
MODEL	year	String	
MODEL	colour	String	

SELLS	name	String	
SELLS	s_type	String	“Dealer” or “Garage” or “Person”
SELLS	vin	String	
SELLS	price	Double	
Registration	vin	string	
Registration	registration-number	String	
OWNER	vin	String	
OWNER	owner	String	
OWNER	status	String	“Current” or “Previous”

Although the global attribute domain (GAD) was not stored in a separate relation, it could obviously be obtained from the above table. It is assumed that each source database is held individually by an organization. The organizations holding those four databases were stored in the Organization relation and Organization-Property relation and Organization-Parent relation, which are shown in Table 8-2, Table 8-3 and 8-4, respectively.

Table 8-2 Organization Relation in the Meta-database of the Data Integrator Service

Organization-Name	Description
Location	This organizational structure categorizes the organizations by location

Table 8-3 Organization-Property Relation in the Meta-database of the Data Integrator Service

Organization-Name	OP-Name	OP-Type	isRoot
Location	China	Categorization Property	Y
Location	GuangDong	Categorization Property	N
Location	Beijing	Categorization Property	N

Location	GuangZhou	Categorization Property	N
Location	D1	Local Schema	N
Location	D2	Local Schema	N
Location	D3	Local Schema	N
Location	D4	Local Schema	N

Table 8-4 Organization-Parent Relation in the Meta-database of the Data Integrator Service

Organization-Name	OP-Name	Parent
Location	GuangDong	China
Location	Beijing	China
Location	GuangZhou	GuangDong
Location	D1	Beijing
Location	D2	Beijing
Location	D3	GuangZhou
Location	D4	GuangZhou

The corresponding DSs of those databases, which are also represented as D1, D2, D3 and D4, were stored in the Data-Service relation.

The local schema of each source database is listed as follows:

1) Local schema of D1:

   VEHICLE *(vin: String, status: String, brand: String, type: String, cap: String, model: String, year: String)*

   SALE(name: String, type: String, vin: String, price: Double)

   CUR_OWNER (vin: String, owner: String)

2) Local schema of D2:

   AUTOMOBILE (vi-n: String, status: String, brand: String, type: String)

VAN (vi-n: String, capability: Double)

MODEL (vi-n: String, transition: String, year: Integer, colour: String)

SELLS (trader_name: String, type: String, v-in: String, price: Double, discount: Double)

3) Local schema of D3:

CAR (vi_number: String, status: String, brand: String)

VAN (vi_number: String, status: String, brand: String)

MODEL (vi_number: String, model: String, year: String, colour: String)

OWNER (vi_number: String, first_name: String, mid_name: String, last_name: String, price: Double)

4) Local schema of D4:

AUTO *(vin: String, status: String, brand: String, type: String, year: String)*

Registration (vin: String, number: String)

OWNER (vin: String, owner_name: String, status: String)

Each local schema was created individually with a set of tuples in order to examine whether the system is able to produce expected answers to pre-defined queries. In order to evaluate the system later in this chapter, it is helpful to examine whether the sample local schemas have covered all the types of heterogeneity defined in Chapter 1. Table 8-5 lists all the heterogeneities and discusses how each one exists among four sample local schemas.

Table 8-5 Heterogeneities among the Sample Source Databases

Heterogeneity	Covered	Explanation
Naming Conflicts	Yes	The local schemas use different relation names and attribute names to represent the same entity. For example, D1 uses relation VEHICLE to represent automobiles, while D2 and D4 use

		relation AUTOMOBILE and relation AUTO respectively. Also, the attribute vin in D1 and the attribute vi-n in D2 represent the same property.
Semantic Conflicts	Yes	D1, D2 and D3 provide an attribute to indicate whether an automobile is an auto one or manual one, while D4 does not.
Structural Conflicts	Yes	1) D1 has one relation VEHICLE to include all the basic information of an automobile, while D2 has a separate relation MODEL to provide information such as model, year and colour of an automobile. It fulfills the Condition (1) and (2). 2) Both D1 and D4 have one attribute to represent the name of an owner, but D3 has three attributes (*first_name, mid_name* and *last_name*) to represent it. Also, the attribute in D1 is composed of price and discount (price * discount). It fulfills the Condition (3). 3) D1, D3 and D4 have a relation to represent owner information, while D2 does not. It fulfills the Condition (4) 4) D1and D2 store the capability information of vans, while D3 and D4 do not. It fulfills the Condition (5)
Metadata Conflicts	Yes	In D1, D2 and D4, the subclasses Van and Car are mapped into one relation with an attribute type to indicate the classification; while in D3, they are mapped into two relations.

Domain Conflicts	Yes	The attribute capability in D2 is in Double, while the attribute cap in D1 is in String.

8.2.1.2 Resulting Views

Having constructed the global schema and the local schemas and the relevant organizations, it can be examined whether a set of views can be built in order to eliminate the heterogeneities. Recall that RSMV includes basically two steps:

- Eliminating the heterogeneities by building exporting views to make the local schemas homogeneous to the global schema.
- Integrating the source databases into the global schema by building importing views in terms of the global schema.

In fact, the elimination of some heterogeneity occurs during the process of both steps. Moreover, some types of heterogeneity are solved during the query process. Therefore, the full discussion requires that both the two steps and the query process be complete. The resulting exporting views and the importing views of the local schemas are shown in Table 8-6 and 8-7, respectively.

Table 8-6 Exporting Views Defined in Source Databases

Source Database	Exporting View	Atomic View
D1	D1-Auto-V	1) D1-Auto-V01 := $\pi_{vin \to vin,\ status \to status,\ brand \to brand,\ type \to type,\ cap \to capability,\ model \to model,\ year \to year}$ (VEHICLE); 2) D1-Auto-V02 := $\pi_{name \to name,\ type \to type,\ vin \to vin,\ price \to price}$ (SALE); 3) D1-Auto-V03:= $\pi_{vin \to vin,\ owner \to owner}$ (CUR_OWNER); 4) D1-Auto-V04 := D1-Auto-V01 $\bowtie_{vin}$

		D1-Auto-V02; 5) D1-Auto-V := D1-Auto-V04 $\bowtie_{vin}$ D1-Auto-V03;
D2	D2-Car-V	1) D2-Car-V01 := $\pi_{vi\text{-}n \to vin,\ status \to status,\ brand \to brand,\ type \to type}$ (AUTOMOBILE); 2) D2-Car-V02 := $\pi_{vi\text{-}n \to vin,\ transition \to model,\ year \to year,\ colour \to colour}$ (MODEL); 3) D2-Car-V03 := $\pi_{vi\text{-}n \to vin,\ trader\_name \to name,\ type \to type,\ price \to price}$ (SELLS); 4) D2-Car-V04 := $\sigma_{type="Car"}$ (D2-Car-V01); 5) D2-Car-V05 := D2-Car-V04 $\bowtie_{vin}$ D2-Car-V02 6) D2-Car-V := D1-Auto-V05 $\bowtie_{vin}$ D1-Auto-V03
D2	D2-Van-V	1) D2-Van-V01 := $\pi_{vi\text{-}n \to vin,\ status \to status,\ brand \to brand,\ type \to type}$ (AUTOMOBILE); 2) D2-Van-V02 := $\pi_{vi\text{-}n \to vin,\ capability \to capability}$ (VAN); 3) D2-Van-V03 := $\pi_{vi\text{-}n \to vin,\ transition \to model,\ year \to year,\ colour \to colour}$ (MODEL); 4) D2-Van-V04:= $\pi_{vi\text{-}n \to vin,\ trader\_name \to name,\ type \to type,\ price*discount \to price}$ (SELLS); 5) D2-Van-V05:= $\sigma_{type="Van"}$ (D2-Van-V01); 6) D2-Van-V06:= D2-Van-V05 $\bowtie_{vin}$ D2-Van-V02 7) D2-Van-V07:= D2-Van-V06 $\bowtie_{vin}$ D2-Van-V03 8) D2-Van-V:= D1-Van-V07 $\bowtie_{vin}$ D1-Van-V04
D3	D3-Auto-V	1) D3-Auto-V01 := $\pi_{vi\text{-}number \to vin,\ status \to status,\ brand \to brand}$ (CAR); 2) D3-Auto-V02 := $\pi_{vi\text{-}number \to vin,\ status \to status,\ brand \to brand}$ (VAN); 3) D3-Auto-V03 := $\pi_{vi\text{-}number \to vin,\ model \to model,\ year \to year,}$

		colour→colour (MODEL); 4) D3-Auto-V04 := $\pi_{\text{vi-number}\rightarrow\text{vin, (first\_name, mid\_name, last\_name)}\rightarrow\text{ name, tprice}\rightarrow\text{price}}$ (OWNER); 5) D3-Auto-V05 := D3-Auto-V01 ∪ D3-Auto-V02; 6) D3-Auto-V06 := D3-Auto-V05 ⋈ vin D3-Auto-V03 7) D3-Auto-V := D3-Auto-V06 $\bowtie_{\text{vin}}$ D3-Auto-V04
D4	D4-Auto-V	1) D4-Auto-V01:= $\pi_{\text{vin}\rightarrow\text{vin, status}\rightarrow\text{ status, brand}\rightarrow\text{brand, type}\rightarrow\text{type, model}\rightarrow\text{model, year}\rightarrow\text{year}}$ (AUTO); 2) D4-Auto-V02:= $\pi_{\text{vin}\rightarrow\text{vin, number}\rightarrow\text{registration-number}}$ (Registration); 3) D4-Auto-V03:= $\pi_{\text{vin}\rightarrow\text{vin, owner}\rightarrow\text{owner, status}\rightarrow\text{o\_status}}$ (CUR_OWNER); 4) D4-Auto-V04:= D4-Auto-V01 ⋈ vin D4-Auto-V02; 5) D4-Auto-V:= D4-Auto-V04 $\bowtie_{\text{vin}}$ D4-Auto-V03;

The attributes of each atomic view are omitted from Table 8-6 in order to simplify the table, because they can be obtained from the view definitions. Obviously, there may be more than one set of atomic views which compose valid exporting views that produce the same results, depending on the designer. Table 8-6 only shows one form of definition of the exporting views. However, it is enough to prove the effectiveness of the RSMV algorithm if there is at least one set of exporting views which are effective.

It is apparent that the exporting views representing the local schemas were all made homogeneous to the global schema, because they satisfy the rules defined in Chapter 4. Therefore, the naming conflicts were eliminated directly, because they were

changed to the attributes in the GAD using the projection operation. The structural conflicts which fulfill the Condition (3) were also eliminated directly, because the composite attributes were composed into a single attribute. In addition, the metadata conflicts were eliminated, because the van relation and the car relation were combined into one view using the union operation. An attribute called type was also added in order to distinguish between "Van" and "Car". Although others were also partially addressed, they required the importing views to be complete.

Table 8-7 shows the corresponding importing views of the source databases, which were defined in terms of the global schema. Those exporting views and the importing views were successfully stored into the meta-databases in the DSs and the DIS respectively.

Table 8-7 the Importing Views of the Source Databases

Source Database	Head Goal	Subgoal
D1	D1-Auto-V(vin,status,brand,type,capability,model,year,name,s_type,price,owner)	AUTO(vin,status,brand,type),VAN(vin,capability),MODEL(vin,model,year,colour),SELLS(name,s_type,vin,price), OWNER(vin,owner,o_status), o_status="Current"
D2	D2-Car-V(vin,status,brand,type,model,year,colour,name,s_type,price)	AUTO(vin,status,brand,type), MODEL(vin,model,year,colour),SELLS(name,s_type,vin,price), type="Car"
D2	D2-Van-V(vin,status,brand,type,capability,model,year,colour,name,s_type,price)	AUTO(vin,status,brand,type), VAN(vin,capability),MODEL(vin,model,year,colour),SELLS(name,s_type,vin,price),type="Van"
D3	D3-Auto-V(vin,status,br	AUTO(vin,status,brand,type),

	and,model,type,year,colour,owner, price)	VAN(vin,capability),MODEL(vin,model,year,colour), SELLS(name,s_type,vin,price), OWNER(vin,owner,o_status), o_status=”Current”, s_type=”Person”
D4	D4-Auto-V(vin,status,type,model,year,registration-number,owner,o_status)	AUTO(vin,status,brand,type), MODEL(vin,model,year,colour), Registration (vin, registration-number), OWNER(vin,owner,o_status)

Having built the importing views, some of the remaining heterogeneity was eliminated. The structural conflicts which fulfill the Condition (1) and (2) were eliminated by building exporting views in terms of the set of relations of the local schema and then building importing views in terms of the set of relations of the global schema. In principle, since the exporting views and the corresponding importing views have the same schema, this indicates that the structural conflicts between the two set of relations have also been eliminated.

It can be found that the semantic conflicts, the domain conflicts and the structural conflicts which fulfill the Condition (4) and (5) were not addressed explicitly by the views. The semantic conflicts and the structural conflicts which fulfill the Condition (4) and (5) are also referred to as missing information problems. The views which are actually queries cannot involve the missing information directly. The domain conflicts were not addressed because the extended relational algebra does not provide operations to address them directly. However, the above three types of conflicts were considered during the query process which will be discussed later in this section.

In principle, most of the heterogeneities were eliminated by building views. However, the system still requires tests to examine whether all those views are correct. In order to examine whether those source databases have been successfully integrated into the

system, the validity of the exporting views was tested first. As mentioned in Chapter 7, a set of SQL exporting views obtained by translating the exporting views into SQL were stored into the source databases as well. This enabled the investigator to conduct tests on the exporting views individually. Each exporting view was tested using a set of queries which involves all the attributes of the root view. No errors came up, indicating that the translation between relational algebra and SQL was correct.

8.2.1.3 User Query and Results

Having tested the exporting views individually, a set of pre-defined user queries were raised on the global schema in order to examine whether the EA-SODIAS could find the relevant source databases and produce the expected results. A set of results for each query were produced manually in order to compare with the results of the test. These involve:

- A set of relevant source databases which can produce complete or partial information for the user query. By partial information, we mean the tuples which provide some of the attributes asked by the user query
- A set of tuples which involve the tuples providing partial information.

The results of the tests are summarized in Table 8-8.

Table 8-8 User Queries and Results

User Query	Found Databases	Relevant Databases	Number of Tuple	Manual Result
Q1	2	4	3	5
Q2	4	4	5	5
Q3	3	3	4	4
Q4	3	3	4	4
Q5	3	4	4	5
Q6	1	1	2	2

Q7	2	2	4	4
Q8	1	1	1	1
Q9	2	2	2	2
Q10	2	2	1	1

For some of the user queries, the source databases found by the system were not identical to the manually produced results; also, the number of the tuples produced by the system was not identical to the number of manually produced tuples. However, this is not surprising, because the manually found source databases which were not found by the system are those which do not provide some attributes or relations required by the user query. This is due to LAV and the Bucket Algorithm adopted in this research. During the process of the Bucket Algorithm, if there are one or more attributes of the user queries which are not provided by the importing views of a source database, the source database will not be considered as relevant and will not be accessed. Consequently, the tuples in this source database will not be obtained. This is also the way in which the method deals with the missing information problem. The manually produced results were then modified following the Bucket Algorithm exactly and the results became the same as the system produced ones.

The local schema which has semantic conflicts may have to be excluded from a query if the query puts any conditions on the attributes that it fails to provide. For example, a query asks for a manual car, but D4 does not provide an attribute to distinguish automatic cars and manual cars. Thus, D4 may not be accessed by the query. However, the semantic conflicts can still be tackled if any other attributes can help to provide this information indirectly.

As mentioned above, the structural conflicts and the semantic conflicts (referred to as missing information) were addressed during the query process by not accessing the source databases that have missing information when the missing information is

required by the user query. Therefore, the system can work without errors. In addition to the missing information, domain conflicts are another type of heterogeneity which is tackled during the query process. Although it is not shown in Table 8-7, the intermediate results showed that the attributes returned from the source databases, if not consistent, were converted to be in the domain of the attributes of the global schema.

8.2.1.4 Summary

The results of the tests showed that most types of heterogeneity defined in Chapter 1 were addressed successfully by the RSMV (building exporting views and importing views). Some of them were eliminated by the relational algebra operations directly, while some of them were eliminated by building both exporting views and importing views. There is no strict rule as to which views need to be built to tackle a particular heterogeneity. It relies, to a large extent, on the experience of the database designers.

The structural conflicts which fulfill the Condition (4) and (5) and the domain conflicts and the semantic conflicts were not eliminated by the views. The RSMV and the query process in this research addressed the former by not taking the source databases with these conflicts into account. Although this can make the system work properly without errors, it means that the system may sometime only provide incomplete results to a user query. However, this is due to the adoption of both the LAV approach and the Bucket Algorithm. It is not the issue raised from building views instead of hard-coded programs, because hard-coded programs cannot involve the missing information as well. The possible solution is that the database provider can add an extra attribute into the exporting views which has null values so that the source database can be considered as relevant although it provide null values. Therefore, the system built based on RSMV most suits the application where the user query requires the latest information and the completeness of the information is vital.

Although the domain conflicts were eliminated, the method was still naive. It can be

improved by defining a set of domains which can be converted effectively to each other so that the unrecognizable value after the conversion can be avoided.

Moreover, the exporting views and the importing views were successful stored into the meta-database in which they were correctly retrieved. Therefore, hypothesis A is well supported and the RSMV is proved effective and the EA-SODIA has the capability of solving the heterogeneities defined in this thesis.

8.3 Capability of Solving Evolution Problems

Recall that the aim of the architecture and the algorithms involved is solving some evolution problems. In Chapter 7, research questions and hypotheses were listed in order to evaluate the architecture and the algorithms in terms of the capability of solving evolution problems. This section presents the results to answer the relevant questions and examines whether the hypotheses are supported. The hypotheses are discussed following the result of each response variable listed in Chapter 7. Finally, research questions are answered.

The research questions regarding the evolution problems are:

2) How and why the RSMV and the meta-database can help solve the evolution problems defined in this thesis.
3) How and why SED can help solve the evolution problems defined in this thesis.
4) How and why SOA can help solve the evolution problems defined in this thesis.

The hypotheses defined for answering the above questions are:

B. The RSMV approach and meta-database can reduce the cost of modification work caused by schema evolutions.
C. If any schema evolution occurs in one source database, the views of other

source databases do not require modification so that the system can still work properly.

D. The SED can reduce the cost of modification work caused by schema evolutions.

E. The query processor can reduce the number of the queries which require modification when any organization evolution occurs.

F. SOA and web services can help reduce the cost caused by the database evolutions because they provide high reusability, autonomy and discoverability.

In order to test the correctness of the SED, a new source database D5 was added:

CAR (vi-number: String, status: String, brand: String)

VAN (vi-number: String, status: String, brand: String)

This source database only aims to produce an exporting view which only includes a Union operation so that the relation removal evolution can be tested. The exporting views shown in the previous section were slightly modified to involve a Group operation. The modified exporting views include all the relational algebra operations so that the test on the SED can be complete. The modified views, however, can produce the same results as that before modified. Table 8-9 shows only the modified exporting views.

Table 8-9 Modified Exporting Views

Source Database	**Exporting View**	**Atomic View**
D1	D1-Auto-V	1) D1-Auto-V01 := $\pi_{vin \to vin,\ status \to status,\ brand \to brand,\ type \to type,\ cap \to capability,\ model \to model,\ year \to year}$ (VEHICLE); 2) D1-Auto-V02 := $\pi_{name \to name,\ type \to type,\ vin \to vin,\ price \to price}$ (SALE);

		3) D1-Auto-V03:= $\pi_{vin \rightarrow vin,\ owner \rightarrow owner}$ (CUR_OWNER); 4) D1-Auto-V04:= $\gamma_{name,\ type,\ vin,\ SUM(price) \rightarrow price}$ (D1-Auto-V02) 5) D1-Auto-V05 := D1-Auto-V01 $\bowtie_{vin}$ D1-Auto-V04; 6) D1-Auto-V := D1-Auto-V05 $\bowtie_{vin}$ D1-Auto-V03;
D5	D3-Auto-V	1) D5-Auto-V01 := $\pi_{vi\text{-}number \rightarrow vin,\ status \rightarrow status,\ brand \rightarrow brand}$ (CAR); 2) D5-Auto-V02 := $\pi_{vi\text{-}number \rightarrow vin,\ status \rightarrow status,\ brand \rightarrow brand}$ (VAN); 3) D5-Auto-V:= D3-Auto-V01 $\cup$ D3-Auto-V02;

The next sections discuss each of the hypotheses by showing the results of the relevant response variables.

8.3.1 Hypothesis B

The hypothesis is:

> The RSMV approach and meta-database can reduce the cost of modification work caused by schema evolutions.

In order to examine whether this hypothesis can be supported, the following response variables need to be obtained:

- Number of user queries involving local schema
- Number of user queries explicitly designating source databases.
- Number of hard-coded queries
- Number of the affected user queries
- Views requiring modification
- Number of source databases affected

In the theory of this research, the RSMV approach and the meta-database should be able to reduce the account of components of the system which require modification when any schema evolution occurs. The meta-database should also reduce the complexity of the modification by avoiding hard-coded programs. Each response variable will be shown and discussed in this section.

Number of user queries involving local schema

The number of the user queries involving local schema is as follows:

- Total User Queries: 10
- User Queries Involving Local Schema: 0
- Percentage: 0

It can be seen that there are no user queries involving local schemas among the 10 pre-defined user queries. It needs to be clarified that the user queries were designed by a colleague of the investigator for examining whether they can obtain expected answers without consideration whether they intend to involve local schema. There is no bias when designing them. In fact, it is easy to explain why there are no user queries involving local schema. It is due to one of the features of the RSMV which is that the users can only see the global schema and do not know the local schemas when raising queries.

It is presented in [63] and [36] that, in a loosely coupled federated database system, all the user queries include the local schemas. Including the local schemas means that the user query requires modification if the local schemas have changed. Therefore, the RSMV ensure that the user queries require no modifications so that it reduces the cost of maintenance caused by schema evolution regarding this aspect. As a mediated [63] system use GAV or LAV to integrate source databases, the user queries of a mediated system do not involve local schemas as well. Therefore, EA-SODIA provides similar advantage at this stage.

Number of user queries explicitly designating source databases

The number of user queries explicitly designating source databases is listed as follows:

- Total User Queries: 10
- User Queries Designating Source Databases: 1
- Percentage: 10

There is one user query which explicitly designates a source database. Recall that a user query can involve a conjunctive query and an organizational scope which indicates an organizational property. The organizational property may designate a categorization property (e.g. Beijing) or a source database (e.g. D1). In this test, the organizational scope of one user query designates a source database. It means that the user wants data only from a single source database. Thus, this user query may be discarded and requires modification when and only when the designated source database is removed. Note that other schema evolutions of the source database will not lead to modification on this user query. In addition, a user query can at most involve one source database in case the user wants to access a single source database. This make the possibility of modifying the user query much smaller.

In a federated system, a user query designates all the source databases required. Consequently, the user query requires modification when one of the involved source databases change. Therefore, the RSMV reduces, to a large extent, the possibility and cost of modification caused by database evolution and hypothesis B is supported by this response variable. Still, EA-SODIA provides no improvement to the maintenance cost at this point compared to a mediated system, because the user queries of a mediated system does not designate source databases either.

Number of hard-coded queries

As mentioned throughout the thesis, one of the most important reasons that an integrated database system is difficult to maintain is that it involves a huge amount of

hard-coded programs for schema reconciliation. Therefore, if the number of hard-coded queries is reduced, the cost of the maintenance can also be largely reduced. The number of hard-coded queries is as follows:

- Number of Source Databases: 5
- Java Classes For Schema Reconciliation: 0

As the EA-SODIAS was built using Java, the hard-coded programs means Java classes in this system. As shown above, there were no Java classes for the purpose of schema reconciliation in both the DIS and the DSs. This is due to the use of the meta-database which stored all the views result from the RSMV approach. Therefore, there are only structured data in the meta-database, rather than hard-coded programs.

The Java Classes of the DIS and the DSs are of course hard-coded programs and may be considered as programs for schema reconciliation. However, these Java classes are not specific to any local schema and do not embed any queries because they are only intended to implement the algorithms by manipulating the meta-database. Therefore, schema evolutions will not lead to modification on them.

In a federated system, all the schema reconciliation work is undertaken by hard-coded programs which can be huge in length and very complex when the number of source database becomes very large. It is also one of the most important reasons that schema evolution is prohibited in a federated system. Although in a mediated system, the integration of the source database may require no hard-coded programs depending on the design, the elimination of the heterogeneity is conducted by a particular hard-coded wrapper of each source database. Therefore, in this aspect, EA-SODIA and RSMV, and the meta-database reduce the modification work compared with the two architectures above. Hypothesis B is supported by this response variable.

Number of affected user queries and Views requiring modification and Number

of source databases affected

In order to examine how EA-SODIAS can deal with evolution problems, a set of evolutions were designed covering all the possible evolutions defined in Chapter 5. Recall that there were three types of evolution defined in Chapter 5: schema evolution, organizational evolution and system evolution. Some of the schema evolutions may have an impact on the system, while other schema evolutions do not. Similarly, some of the organizational evolutions have impact on existing user queries, while other organizational evolutions do not. The system evolutions involved all have some impact on the system, but they should be tackled easily. In order to demonstrate the hypothesis in this thesis, all the evolutions were involved in the test data. The design of the schema evolutions was more complex than that of others, because the same schema evolution on a different attribute or relation may result in a different automatic view modification process. Therefore, the schema evolutions were designed to consider each possible route of the process. The results are shown in Table 8-10.

Table 8-10 the Results of Evolutions

Evolution	Number of Evolutions	Number of the Affected User Queries	Exporting Views Requiring Modification	Importing Views Requiring Modification
Attribute Addition	10	0	0	0
Attribute Removal	10	0	13	13
Attribute Rename	10	0	13	0
Attribute Domain Change	5	0	6	0
Attribute Decomposition	2	0	2	0
Relation Addition	4	0	0	0
Relation Removal	5	0	6	5

Relation Rename	4	0	5	0
Relation Decomposition	1	0	5	0
Database Addition	1	0	0	0
Database Removal	1	0	0	2
Organizational Property Removal	3	5	0	0
Organizational Property Rename	3	0	0	0
Organization Removal	1	10	0	0
Organization Rename	1	0	0	0
Parent Change	1	0	0	0
Organization Addition	1	0	0	0
Organizational Property Addition	1	0	0	0
Service Name Change	1	0	0	0
Database Name Change	1	0	0	0
Service URL Change	1	0	0	0
Total	67	15	50	20

Note that the evolutions in Table 8-10 were designed to cover every typical evolution which has a different impact on the system. More evolutions were also designed and randomly applied to the system in order to further examine whether the system can produce correct results. The system was modified until no programming errors occurred.

It can be seen from Table 8-10 that the user queries required no modification when schema evolution and system evolution occurred. This is due to the adoption of the LAV approach. The LAV ensures that the user queries do not involve local schemas so that they will not be affected by schema evolution.

Table 8-10 also shows that the only type of evolution which may have an impact on the user queries is organizational evolution. There were two types of organizational evolution: organizational property removal and organization removal, which led to some user queries being discarded. As a user query may designate an organizational property in which the relevant source databases will be accessed, the user query may become invalid when the exact organizational property is removed. Also, when a whole organization is removed, all the user queries which designate that organization will be invalid. Making the user queries valid again completely depends on the manual work of system maintainers and users. However, these two types of organizational evolution only account for a very small proportion. Other organizational evolution will not require any modification on the user queries. This is due to the query processor introduced in this research.

The main components of the system which will be affected by schema evolution are the exporting views, because they involve all the local schemas which will be accessed in order to eliminate heterogeneities. Having applied the evolutions, some of the exporting views required automatic modification, while some of them were discarded and required manual modification. How the automatic view modification can help reduce the maintenance cost is discussed later in this chapter. It may be realized that the number of affected exporting views was greater than the number of evolutions when some schema evolutions were applied. This was because more than one exporting view involved one attribute or relation which was changed.

The system has the advantage that the importing views which integrate source databases were rarely affected by the evolutions. They required modification only when three types of schema evolutions occurred. This means that the DISs will rarely be modified.

Another advantage of EA-SODIA is that no system evolution has impact on the user

queries and the views. This is due to the use of web services and the registry which are discussed later in this chapter.

In a federated system, the user queries designate all the local schemas being accessed. Hard-coded programs are used for both eliminating heterogeneities and integrating source databases. The organizational information is also embedded into the hard-coded user queries. Therefore, when an evolution occurs, all the user queries and hard-coded programs must be modified. Hence, the RSMV reduces the components which require modification compared with a federated system based on the schema integration approach. Hypothesis B is thus supported by these response variables.

8.3.2 Hypothesis C

The Hypothesis is as following:

> If any schema evolution occurs in one source database, the views of other source databases do not require modification so that the system can still work properly.

One of most important reasons that a federated system and a mediated system based on GAV do not allow evolution is that there are many hard-coded programs or views which deal with the relationship between local schemas. Therefore, when a schema evolution occurs, not only do the local evolved local schema need to be considered, but also the relationship between other local schemas and it will also be considered. It makes evolution impossible when the number of local schemas becomes very large. In addition, because the relationship is embedded in the hard-coded programs directly, the system may crash when a schema evolution happens to any of the related local schemas. However, the system may allow evolution if the system does not include an explicit relationship between the views of different source databases at design-time.

The case study looked at this aspect by apply a schema evolution to a local schema to see if the system can work properly. The case study undertook the following steps:

- It discarded the evolved source database by discarding the importing views

of the source database. Until the relevant exporting views and importing views were properly modified, the evolved source schema remained discarded. The pre-designed user queries were entered into the system to examine whether the relevant source databases could be found and the results could be obtained as expected.

- It then modified the affected importing views and exporting views to examine whether only the importing views and exporting views of the evolved source database were modified.

The result shows that the system worked properly excluding the evolved source databases during the first step. The expected relevant source databases were found without considering the evolved source database. The results obtained were the same as the manually produced results. Therefore, the RSMV approach ensures that the local schema be integrated into the system independently of other local schemas so that a schema evolution on one local schema will not have any impact on other local schemas. Consequently, the system can work properly in terms of other local schemas when a schema evolution occurs in one local schema. The results also indicate that only the exporting views and the importing views of the evolved local schema were modified. Modifying exporting views and importing views of a single local schema independently is much simpler. This is due to the nature of the LAV approach. Each local schema is integrated independently and therefore only has a relationship with the global schema. The relationship among the local schemas is worked out during the run-time using Bucket algorithm. Thus, the RSMV reduces the modification work caused by schema evolution and hypothesis C is supported.

8.3.3 Hypothesis D

The hypothesis is:

The SED and Meta-database can reduce the cost of modification work caused by schema evolution.

Recall that SED provided seven processes in order to automatically modify the exporting views and the importing views. The aim of the SED is to reduce the manual modification work. Therefore, in order to examine whether Hypothesis D is true, one needs to determine how much manual modification work is still required. The case study evaluated this aspect by applying the types of schema evolution listed in Table 8-10 and find out how many views cannot be automatically modified. The results are shown in Table 8-11. The discarded exporting views and importing views are those views which require manual modification.

Table 8-11 the Number of Discarded Views Resulted from Schema Evolutions

Evolution	Number of Evolutions	Number of Discarded Views	Percentage (%)
Attribute Addition	10	0	0
Attribute Removal	10	2	20
Attribute Rename	10	0	0
Attribute Domain Change	5	1	20
Attribute Decomposition	2	1	50
Relation Addition	4	0	0
Relation Removal	5	6	120
Relation Rename	4	0	0
Relation Decomposition	1	0	0
Database Removal	1	2	150
Total	52	12	23

Table 8-11 shows that most of the schema evolutions (77%) did not require manual modification on the views. The only schema evolutions which led to manual modification were attribute removal, relation removal, attribute domain change and attribute decomposition. This is due to the fact that a manual decision is required to modify the views when the above schema evolutions occur. Among these four schema

evolutions, the attribute removal and the relation removal led to more discarding of the views because removing attributes or relations means that the data stop providing these information. The relation removal has the highest possibility of discarding views because the process tackling relation removal discards the view as long as the removed relation is in a relational algebra operation other than union.

Although the attribute decomposition and the attribute domain change can also result in discarding of the views, there is only a small possibility of this. There are two attribute decompositions shown in Table 8-11 and one makes a view discarded. However, it does not reflect the real situation completely, because there are two possibilities when an attribute decomposition happens:

- The evolved attribute is itself a source in the projection operation of a view. This does not discard the view.
- The evolved attribute is one of the operands of an expression in the projection operation of a view. This discards the view.

The test data of the case study only design two attribute decompositions each of which cover one of the two possibilities. This is why the percentage was 50%. However, there are 52 attributes in the four source databases and only two of them are in the expression of a projection operation. Therefore, in principle, the possibility of discarding a view when attribute decomposition happens is only 4%. Although, the percentage may vary depending on different source databases, the total possibility should not be high.

In a federated system or a mediated system, the integration and the schema reconciliation of the source databases are undertaken by hard-coded programs. Hard-coded programs are more complex to understand than the structured data in the meta-database. Although some semi-automatic software tools have been produced, the wrappers of a mediated system rely largely on manual work. The schema reconciliation of a federated system is also carried out manually. In our architecture,

the meta-database enables database providers to save views as structured data in a relational database. Therefore, software components implementing SED can be provided so that most of the manual work is saved. Hypothesis D is supported by this response variable.

8.3.4 Computational Cost

It has been discussed that most of manual modification has been replaced by automatic view modification provided by SED in EA-SODIA. Therefore, it is helpful to look at the computational cost of the SED. The SED was run on each schema evolution listed in Table 8-11 and calculated the average time for each type of schema evolution. The schema evolutions which discarded the views were not considered. The results are shown in Figure 8-1.

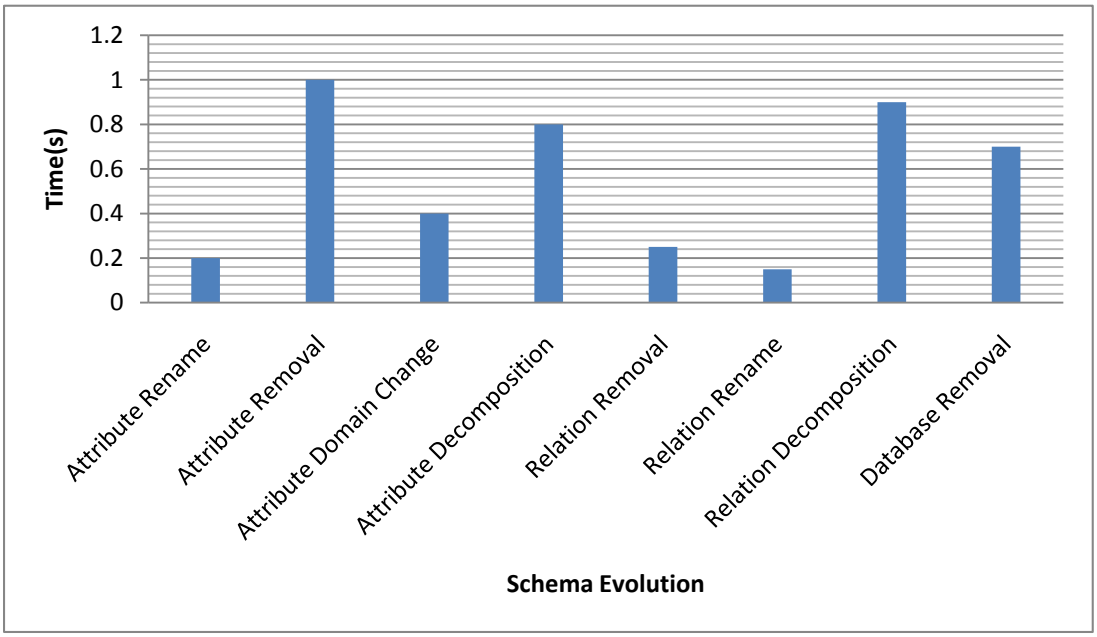

Figure 8-1 The Computational Time of SED

It can be seen from Figure 8-1 that all the average computational times of the SED were all less than or equal to one second. The most economic processes were the process tackling attribute removal and the process tackling relation removal. This is due to the constraint defined in Chapter 5 that every relation and attribute of a local schema must be renamed using a projection operation before it is taken by other

atomic views. Consequently, the SED only needs to check the atomic views which involve a projection operation. It replaces the evolved attribute or relation with the new one if the evolved relation or attribute is involved. Therefore, no other atomic views were checked so that it took the least computational time.

The attribute removal and the database removal took more time because this involved the communication cost between the DS and the DIS. The communication between the DS and the DIS was required because the importing view required modification. In addition, the relation decomposition and the attribute decomposition also took more time because they needed to create new views so that more connections with the DBMS were required.

Apparently, the automatic view modification takes much less time than the manual work, because no manual work can be finished within a few seconds. Therefore, the hypothesis D is further supported. The manual work on producing views will also be discussed later in this chapter.

8.3.5 Hypothesis E

The hypothesis is:

> SOA and web services can help reduce the cost caused by the database evolutions and system evolutions because they provide high reusability, autonomy and discoverability.

EA-SODIA is a service-based architecture. It is expected that the service-based architecture and web services technologies can provide high reusability, autonomy and discoverability. The case study examined each of the features.

Reusability

In order to examine the reusability provided by web service technologies, several new databases were added into the system to see how the DS can be reused. The DS that

exposes a source database is built using the OGSA-DAI which were developed using web service technologies. The following steps were used to publish a new source database:

- Using the tool provide by OGSA-DAI to publish a new source database. Firstly, the name of the new service was required. Secondly, the type and the name and the URI of the database were required. Finally, the password and the user name were required for the service to access the database.
- The exporting views and the importing views were built.

Each new database was deployed and published by the same steps. It can be seen from the above steps that the data required for a particular database were:

- the name of the new service.
- the type and the name and the URI of the database.
- the password and the user name for access to the database.
- the exporting views and the importing views.

Among the above data, the first three were provided easily within one minute by the investigator. The exporting views and the importing views had to be built differently each time because heterogeneity problems needed to be solved. However, no hard-coded programs were developed for publishing a new database. All the Java classes provided for the EA-SODIAS were completely reused without any modification. Also, the OGSA-DAI programs which deal with the message transfer between services and the connection with the DBMS and the data transformation between Java resultsets and the SOAP documents were also reused. As the DS of OGSA-DAI has the consistent functions and parameters for external service requestors to access, the DS can be accessed immediately after publishing. In a traditional system (federated system or mediated), hard-coded programs are required to add a new database into the system. Therefore, it concluded that SOA and web service technology provide high reusability, in that they help to reduce the cost of system maintenance.

In addition, the information such as the name and URI of the database for connection with the DBMS are stored in an XML file, rather than a hard-coded program. Therefore, the system can be easily maintained when a system evolution occurs. In the case study, the investigator used the tool of OGSA-DAI to change the name and the URI of the database. This work was finished within two minutes. Therefore, it further supports hypothesis E.

Autonomy

One of the issues of distributed databases, although not the focus of this research, is autonomy. The database providers need to have complete control over their databases and their applications must be operated independently. The case study showed that the database providers can have complete control over information exposed to external applications by building views. The external application was only able to identify and access the exporting views, rather than the local schema. Also, database providers do not provide the authority of accessing the database to external users. The password and the user name are only used by the DS so that all the external applications (DISs in this case) access the DS and do not have any authority information. The local schema can be used by other applications normally. There are no rules on the design of the local schemas and how they are managed. A database provider has the complete decision when the database is added to and removed from the integrated system. Therefore, EA-SODIA provides high autonomy due to the use of web services.

Discoverability

The case study showed that EA-SODIA provided high discoverability by using a registry service. This is one of the most important characteristics provided by SOA and web services. Although UDDI was not used, the registry service which was a DS provided similar functionality. All the DSs were registered into the registry service where the DIS can find the locations of all relevant DSs. The DISs were also registered into the registry service so that the SED of the DSs were able to find all the DISs where the importing views required modification. In addition to providing high

discoverability, the registry service provided much help with reducing the maintenance work. The registry service stores and manages the information of all the services centrally so that an application can find all the relevant services in this system by maintaining only the location information of the registry service. Consequently, when a system evolution (e.g. the name and the location of a service) occurs, only the information stored in the registry service requires modification.

In a loosely coupled federated system, each database has to maintain the information of all other databases. Consequently, all the databases require modification when a system evolution occurs on one database. In a mediated system, each mediator maintains the information of all the involved source databases. Therefore, when a source database evolves, all the mediators must be modified in order to work properly. However, the source databases do not keep the information of the mediators so that it is difficult to locate all the mediators.

To conclude, the SOA and web service technology provide considerable help in reducing the maintenance cost caused by evolutions by providing high reusability, autonomy and discoverability. Hypothesis E is therefore supported.

8.4 Scalability

One of the reasons that the traditional integrated systems (e.g. the federated system and the GAV mediated system) do not allow evolution is that the number of the source databases may become very large. As the number of source databases becomes larger, the system becomes dramatically more complex because it has to deal with all the relationships among source databases at design-time. Therefore, another important characteristic which is scalability was examined by the case study. The case study examined scalability by replicating the four source databases which were designed at the beginning. Then, increase the number of DISs. The case study still applied the same pre-defined schema evolutions in Table 8-10 on the system. The main variables

which were examined are:

- How the number of the views changes when the number of the source databases becomes larger. The results are shown in Figure 8-2.
- How the number of the affected views changes when the number of the source databases becomes larger. The results are shown in Figure 8-2.
- How the number of the discarded views changes when the number of the source databases becomes larger. The results are shown in Figure 8-2.
- How the computational cost of the SED changes when the number of the source databases becomes larger. The results are shown in Figure 8-3.
- How the computational cost of the SED changes when the number of the source databases becomes larger. The results are shown in Figure 8-4.

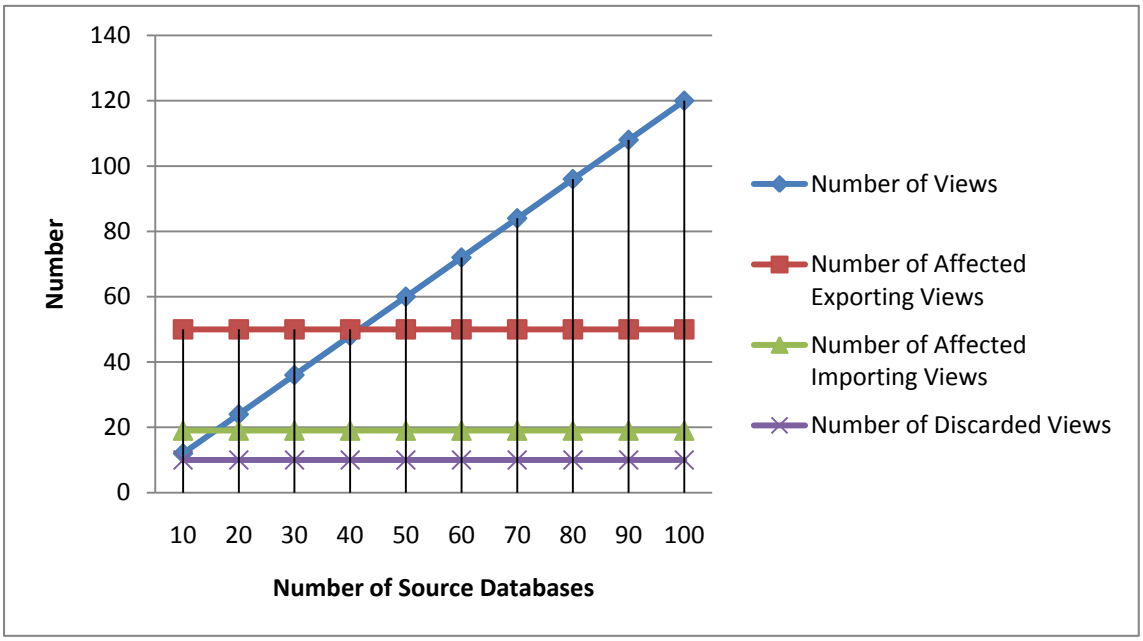

Figure 8-2 Growth of the Number of the Views and the Affected Views and the Discarded Views when More Source Databases are Added

Figure 8-2 shows that all the numbers remained unchanged when the number of source databases increased, except the number of the views which had a linear growth. The increase of the number of the views was as expected, because for each new source database, a set of views needed to be constructed. The number which generally

remained stable indicated that the components of the system did not increase when more source databases were added. Therefore, the EA-SODIAS provided good scalability at this stage.

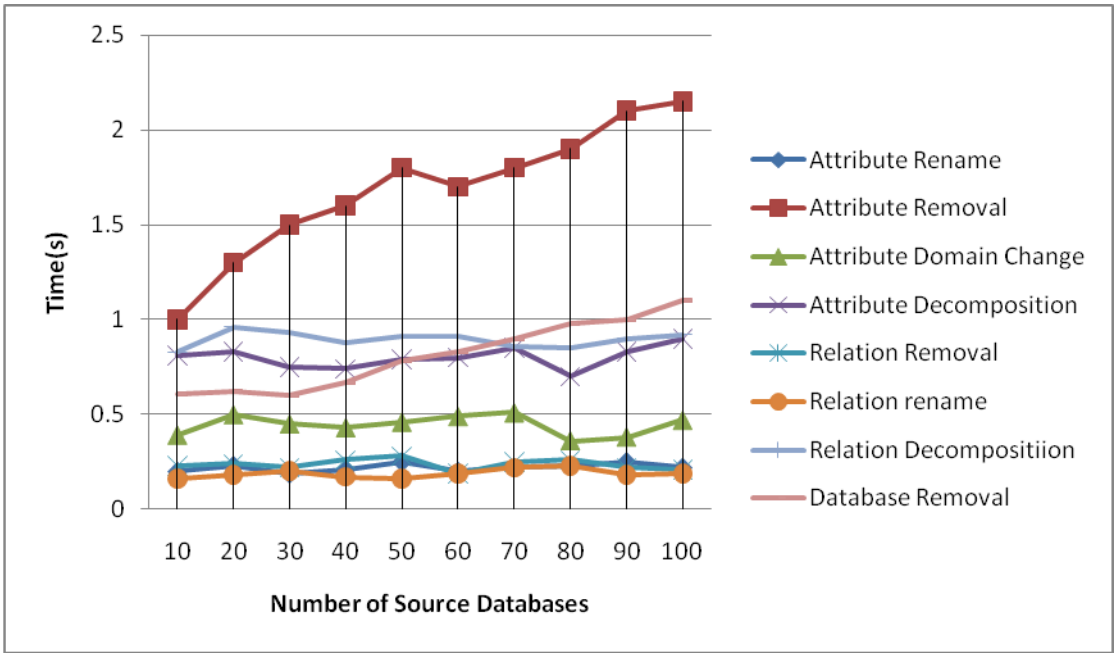

Figure 8-3 Growth of Computational Time when More Source Databases are Added

Figure 8-3 shows that the computational time of the SED for most of the schema evolutions remained similar. The computational time of the SED for the attribute removal and the database removal had a slow linear increase because these two types of schema evolution required the access to the registry service and the DIS where the importing views needed to be modified. As more source databases were added, the number of the importing views increased so that the computational time grew slightly. Generally, the EA-SODIAS provides much better scalability at this stage compared with traditional systems.

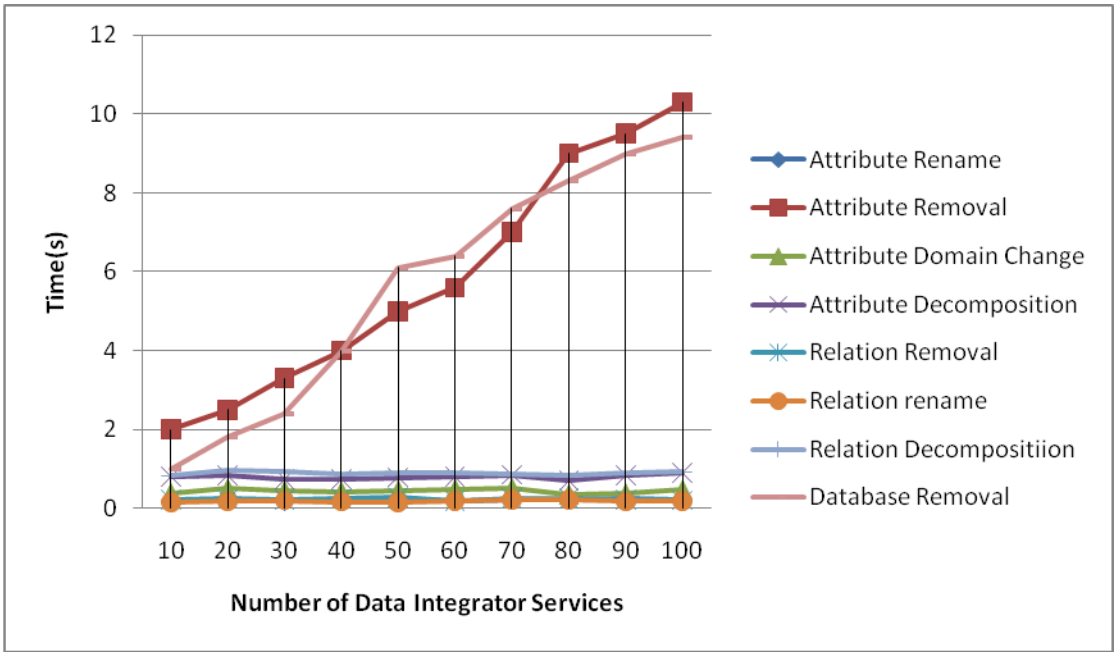

Figure 8-4 Growth of Computational Time when More Data Integrator Services are added

Figure 8-4 shows similar results to that of Figure 8-3. The computational time of SED for tackling attribute removal and database removal grew linearly, while others remained similar. It may be seen that the increase in the computational time of these types was more dramatic than that in Figure 8-3. This is because that the increased data integrated services led to more communication between the DS and DISs. When an attribute removal evolution or a database removal occurred, the DS needed to access the register service to find all the DISs and then access each of the DISs in order to modify the relevant importing views. However, the computational time was at the same level which was counted in seconds. Also, most of the computational time remained unchanged.

To sum up, the above results showed that the components of the system which required modification and the computational time of SED caused by the schema evolutions generally remained unchanged, when the number of source databases and the DISs grew. Although, there were small increases in some computational time, they stayed at a similar level. Therefore, EA-SODIA can provide good scalability.

8.5 Manual Work

Although most evolution can be automatically tackled by SED and the query processor, there is still some manual work. Therefore, it is important to examine how much work is required for integrating a source database into the system and how much work is required to identify affected views and modify them when a schema evolution occurs. As the investigator is also the designer of the system and the source databases, the work of the investigator may not be representative. Therefore, a colleague of the investigator who has knowledge in relational databases was asked to undertake this work. The colleague integrated the pre-designed four databases into the system and then modified the relevant views manually when the pre-designed schema evolutions were applied. For the work of tackling schema evolutions, the average estimated time was recorded. The results are shown in Figure 8-5 and 8-6, respectively.

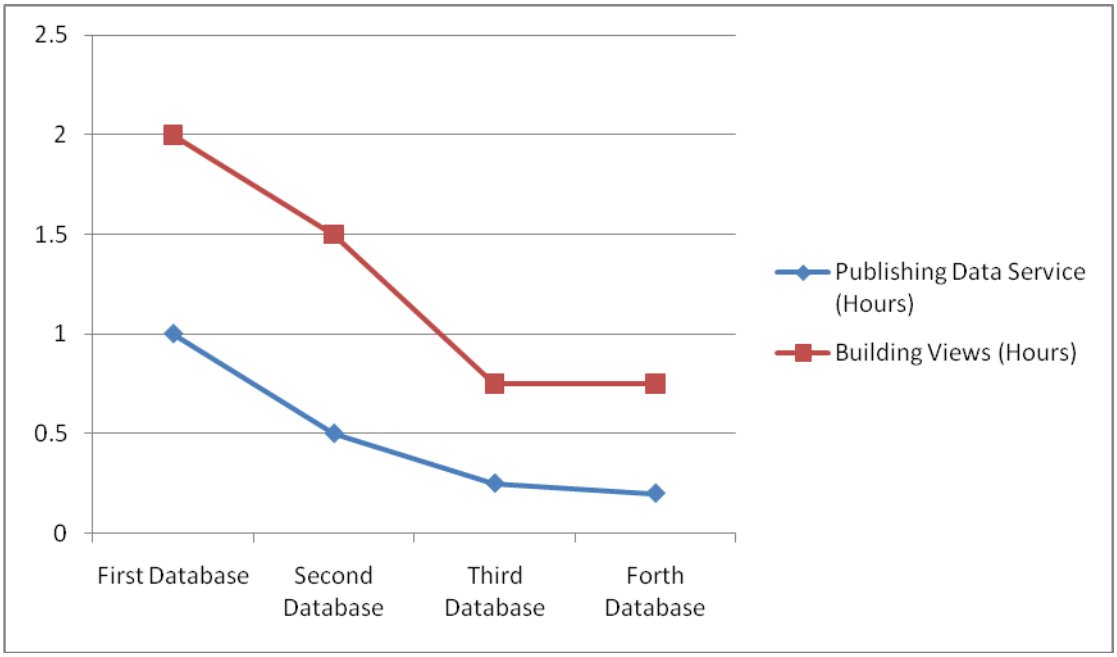

Figure 8-5 Time to Integrate New Source Databases

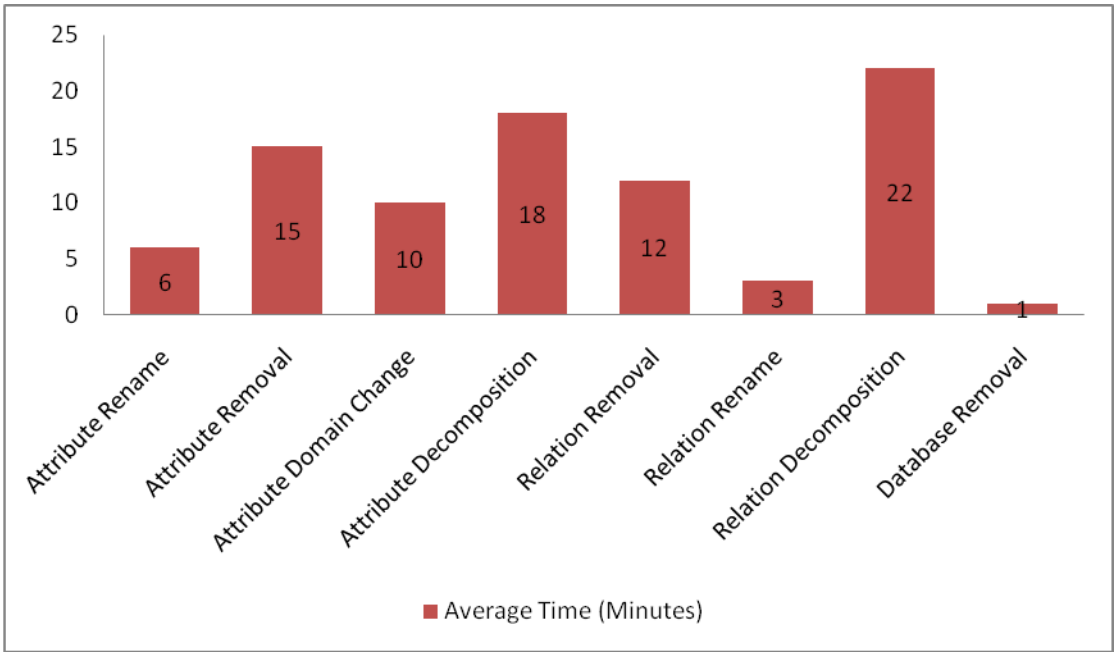

Figure 8-6 Time to Tackle Schema Evolutions

Note that the time for learning how to integrate a source database and tackle schema evolutions was not counted. Figure 8-5 shows that the time of publishing dropped from one hour to about fifteen minutes while the time of building views dropped from two hours to about forty minutes, as the investigator became familiar with the method. Therefore, it can be expected that a database administrator can integrate a database into the system within about three hours, which is acceptable.

Figure 8-6 shows that database removal and relation rename evolutions took the least time while attribute decomposition and relation decomposition evolutions took the most time. All the average times were less than an hour which is acceptable. In addition, schema evolutions which took the most maintenance work can be automatically tackled by the SED so that it further proved that the SED is effective.

8.6 Expandability

This section discusses the ability of EA-SODIA to incorporate different type of source databases.

As mentioned in Chapter 1, the source databases which can be integrated into the system must be in the relational model. In addition, the exporting views are constructed using relational algebra operations. As the relational algebra operations are designed to manipulate the data in the relational model, the data which can be directly integrated into the system are relations. Other data sources such as flat files and legacy hard-coded queries and the object-oriented databases cannot be integrated into the system by constructing exporting views. However, the system has the mechanism for translating the relational algebra into the language which can obtain actual data from a local database. Therefore, other data sources may be integrated into the system as long as the corresponding programs are provided by the data providers for obtaining the data and converting them into relations. Also, the DSs need to be extended by providing the programs for translating relational algebra into the queries which can invoke the programs provided by the data providers.

As the importing views are constructed using a conjunctive query language, they have the ability to integrate other types of source databases (e.g. objective-oriented database). Also, the OGSA-DAI support other data sources such as XML documents. Therefore, it is concluded that EA-SODIA has the potential to integrate other types of source databases by adding new components, although it does not have this ability at this stage, In addition, it requires further research to deal with the evolution problems brought by other types of data sources.

8.7 Domain Independence

The RSMV, the meta-database and SED are designed for data in the relational model without being tailored to a particular domain. As discussed in Chapter 7, the information can be modelled into data in a relational model independently of its domain. The relational data can then be integrated into the system. Although the case study is a single case study which did not provide a set of data for another domain, it can seen from the case study there are no views which rely on the information specific

for that domain. The data manipulated by the approaches in this research are only relations and attributes and local schemas. In addition, it can be seen throughout the thesis, all the examples demonstrating RSMV, the meta-database and SED use the symbols which are independent of any application domains. Therefore, it is concluded that EA-SODIA and all the algorithms in this research are domain independent.

8.8 Language Independence

This section discusses the potential for incorporating services developed in other programming languages. EA-SODIAS was designed and developed using Java. In principle, web services and SOA are completely language independent, because each service provides functions based on the same standards and the communication between services are based on SOAP messages. Therefore, the service requesters need not know the programming language of the applications exposed by the web services. Currently, web services are supported mainly by two programming languages, Java and Microsoft ASP.net. They both provide APIs and development tools and web containers for developing web services. Therefore, the web services developed in either Java or ASP.net can work together as long as the parameters can be converted into the SOAP document. Other programming languages, however, do not provide support for web services.

However, one issue was found during the case study. It was found in a discussion with one investigator's colleague who was developing a DS using ASP.net that the ASP.net is using a different format to encapsulate the relation of query results. Consequently, the query result encapsulated by ASP.net service in the SOAP document may not be correctly accepted by a service developed using Java. Another problem found in the case study is that OGSA-DAI currently relies on the Java environment and is not language independent. Thus, incorporating DSs developed by other languages was not successful in this case study. However, this may be solved in the future when programming languages implement more characteristics of web services and SOA.

8.9 Disadvantages

Some issues were also found in the case study. They indicated that under some environments the system based on EA-SODIA does not work well. The issues are as follows:

1) As mentioned previously, other types of data sources cannot be integrated into the system because the heterogeneity among them cannot be solved using relational algebra views. Although it is not a success criterion in this research, some legacy systems which involve other types of databases (e.g. flat files and objective-oriented language) do exist in practice. The traditional integration systems such as federated systems and mediated systems may have the ability to integrate other data sources, because the heterogeneity problems can be handled by hard-coded programs. This is a remaining issue of EA-SODIA and requires further research.
2) In this research, schema evolution must be entered into the system in order. It means that a maintainer has to know not only what schema evolutions occurred, but also the order. In practice, it is possible that the maintainer cannot remember the sequence of the occurrence of schema evolution. It is also possible that the complete database schema has been replaced. Consequently, SED of EA-SODIA can be successfully conducted. It also requires further research to deal with a batch schema evolution or replace an existing schema with a new schema.
3) It is assumed in this research that the data in the tuples are consistent, meaning that same terms are used. However, it is not realistic in a real-world project. This problem is usually called an ontology problem which is an important topic of other research.
4) Due to the nature of the LAV and the Bucket Algorithm adopted in this research, it was shown in the case study that the system was only able to provide incomplete result sometimes. In the environments where the completeness of the data is vital, the federated system and GAV mediated and

the Data warehousing may be suitable choices.

5) One of the main concerns of the LAV approach is the computational cost of the query containment test. The query containment test is well known to be NP-complete. Although the Bucket Algorithm is used in order to reduce the cost of the containment test, it may still be huge when a large number of source databases are integrated. Evaluation of the LAV approach is shown in [63] and [98]. However, this may be released by adding conditions into both user queries and importing views so that the number of relevant source databases can be largely reduced.

8.10 Conclusion

The conclusion reached by undertaking the case study is that EA-SODIA and all the approaches introduced in this research have the ability to deal with most of the heterogeneities defined in Chapter 1. More importantly, the goal of this research has been achieved which is to solve the evolution problems and reduce the cost of maintenance work caused by evolution. During the case study, all the questions were answered and all hypotheses were supported by analyzing various response variables.

The test data of the case study included all the heterogeneities defined in Chapter 1 and all the evolutions defined throughout the thesis were applied into EA-SODIAS. In addition, all possible conditions of SED were considered when designing the schema evolutions. Therefore, the case study can be considered as a representative case. Also, it was discussed that EA-SODIA is domain independent so that the results can be generalized to other application domains.

Other characteristics such as scalability and expandability and language independence were discussed. Although some disadvantages were still found during the case study, the results show that the success criteria defined in Chapter 1 were fulfilled.

Chapter 9 Conclusion

9.1 Introduction

Chapter 8 presented the results of the case study for evaluation. Four research questions were answered and five hypotheses were supported by analyzing the response variables. Other properties such as scalability and domain independence and language independence and expandability were also discussed followed by some disadvantages found during the evaluation.

This chapter reviews the research presented in this thesis. The work accomplished is compared to the criteria for success defined in Chapter 1, some general issues and directions for further work are discussed.

9.2 Review of Research

9.2.1 The Research Issues

This thesis investigates the integration of separate existing heterogeneous and distributed databases which, due to organisational changes, must be merged and appear as one database. The integrated system based on our architecture is referred as a virtual view approach and needs to:

- provide an integrated view of data from autonomous heterogeneous data sources.
- allow data sources to evolve independently.

Schema reconciliation and query decomposition and schema evolution detection were identified as the major research issues. Schema reconciliation deals with heterogeneity problems by building views which are stored in a meta-database. Query decomposition finds all the relevant source databases and deals with organizational evolutions. Schema evolution detection automatically modifies the views when schema evolution occurs.

9.2.2 Related Work and SOA

Chapter 2 explored several existing approaches to database integration. They are federated database systems, data warehousing, DQP and mediated systems. The comparison between those approaches was presented, discussing why they lack support for evolution. The SOA and web services were introduced together with the idea of SaaS and late-binding. These presented their potential to deal with evolution.

9.2.3 Evolution Adaptive Service-Oriented Data Integration Architecture

The architecture for database integration in this thesis which deals with the heterogeneity and evolution is called EA-SODIA. It involves three processes addressing the three major research issues reviewed in section 10.2.1. Chapter 3 presented the overview of the architecture and the components of the architecture encapsulating the three processes.

The first process is schema reconciliation (presented in Chapter 4) which reconciles the schemas and the representation of heterogeneous source databases, establishing mapping between the local schemas and the global schema in the meta-database. The algorithm for schema reconciliation is termed Relational Schema Mapping by Views (presented in Chapter 4) which is accomplished by stages at DIS and DS respectively. The RSMV eliminates heterogeneities and integrates source databases by constructing exporting and importing views. The approach, called LAV, is adopted to integrate the reconciled local schema into the global schema. Both views are represented and stored in a meta-database. The formal representations of the views were also presented.

The process of solving schema evolutions is called Schema Evolution Detection (presented in Chapter 5). Some rules were defined to identify the views affected by a schema evolution and whether a view must be discarded. Two processes were presented in order to identify affected views and automatically modify affected views. Finally and more importantly, eight processes for automatic view modification were

produced in order to tackle different types of schema evolution.

The final process is query processing (presented in Chapter 6) which basically involves four steps: Query Reformulation, Query Decomposition, Query Transformation and Result Composition. All of these steps identify the relevant source databases and transfer user queries into queries in terms of the local schema, finally obtaining the results. Among these steps, query reformulation takes the responsibility for tackling the organizational evolutions. The query decomposition step adopts the Bucket Algorithm which is a query process algorithm for LAV approach.

The EA-SODIA was compared with the related works in some aspects such as scalability and complexity of creation and complexity of maintenance. Chapter 3 presented a summary of this.

9.2.4 Service Design

As EA-SODIA is service-based and the algorithms in this research need to all be implemented by services, the design of both DIS and DS was presented in Chapter 7. Although the schema reconciliation based on RSMV is manual work, the meta-database is managed at both DIS and DS respectively. A design method which combines the service-oriented design and the object-oriented design was presented with its advantages discussed.

9.2.5 Case Study

As both heterogeneity and evolution are difficult to evaluate, a single case study was conducted to examine all typical situations. An experimental implementation (presented in Chapter 7) was produced for the case study, as the complete implementation was unrealistic. The experimental implementation reflects the query processing and the schema evolution detection, employing Java classes which are embedded in OGSA-DAI to implement various parts of the process. Although the

RSMV is mostly manual process, the meta-database was designed and created at both services.

Chapter 7 presented an introduction to the case study method and discussed why it was chosen as the evaluation method in this research. The research questions and hypotheses and some response variables of the case study were also defined. Chapter 8 presented an extensive and detailed evaluation of the EA-SODIA and all of its algorithms using the hypotheses and the response variables defined in Chapter 7. A set of local schemas in an application domain were designed in order to examine the capability of eliminating heterogeneity of the system. Although two heterogeneities were not addressed directly by building exporting views and importing views, they were addressed in the query process. The issue of an incomplete result which is a consequence of the adoption of the Bucket Algorithm and the LAV approach was discussed. The capability of handling evolution was also examined. All the hypotheses were generally supported. The examination of the computational cost and the scalability showed that most of the cost remained similar and others only had a slow linear growth. The case study also showed that the manual work of integrating a new database and modifying existing views in response to the schema evolutions were acceptable. The EA-SODIA can be applied into other application domains and other programming languages can be used as long as they provide support for web services. Finally, some environments where the EA-SODIA is not suitable were also discussed.

9.3 Evaluation of the Research

An evaluation of the research reported in this thesis is now presented in the context of the criteria for success and research aims given in Chapter 1. There are repeated here with a discussion of each.

1) The heterogeneities defined in Chapter 1 can be eliminated using the RSMV approach and query processing.

Chapter 4 defines a set of extended relational algebra operations which can deal with

most of the heterogeneities. The remaining heterogeneities such as domain conflicts and some structural conflicts are tackled during query processing presented in Chapter 6.

2) The RSMV approach and meta-database can reduce the cost of modification work caused by schema evolutions, and the query processor can reduce the number of the queries which require modification when any organizational evolution occurs.

Chapter 4 defines the formal representation of the views in the meta-database so that all views can be stored in the meta-database. Modifying the structured data in the database is simpler than modifying hard-coded programs. The RSMV ensures that there are no hard-coded programs required for schema reconciliation. Chapter 6 provides a process of identifying source databases which can tackle some organizational evolutions.

3) If any schema evolution occurs in one source database, the views of other source databases do not require modification so that the system can still work properly.

The adoption of LAV ensures that each local schema is integrated into the global schema independently of other local schemas. The relationship among local schemas is worked out by the query containment test. Therefore, the views of the source database are independent of each other.

4) The SED and Meta-database can reduce the cost of modification work caused by schema evolutions.

As all the views are stored in the meta-database presented in Chapter 4, the automatic view modification is able to be produced. Chapter 5 provides eight processes for tackling schema evolutions by automatically modifying the views in the meta-database so that most of the manual work can be saved.

5) SOA and web services can help reduce the cost caused by database evolution and system evolution because they provide high reusability, autonomy and discoverability.

Chapter 3 and 7 present the various characteristics of SOA and web services. As web services follow a consistent standard, they provides high reusability. This ensures that no modification is required on the components of the service when publishing new source databases. The registry service enables the DS to find all the DISs and the change of the information such as name of URL of the services is easier to tackle.

Is has been demonstrated that the work presented in this thesis meets the criteria for the success and research aims defined in Chapter 1. Section 9.4 discusses these accomplishments, and section 9.5 identifies areas for continuing the work and improving the capabilities of the method.

9.4 Discussion

A reflective discussion of the work accomplished in this thesis is now presented. In general, EA-SODIA is a success. It meets the requirements shown in Chapter 1.

The RSMV approach has proved to be successful. The adoption of the LAV and the Bucket Algorithm is a good choice, as they allow source databases to be integrated independently. Although the high cost of the query containment test is the major concern, the Bucket Algorithm relieves the problem to some extent. Also, adding extra conditions in both importing views and user queries is a possible solution for reducing the cost. Although the LAV and the Bucket Algorithm can only provide incomplete results in some situations, they fulfill the requirements for solving evolution problems. The use of a set of relation algebra operations has been a good idea, as the views constructed in terms of them can be well represented as structured data in the meta-database.

One of the major successes has been the meta-database. The formal representation of the data in the meta-database has proved to be effective so that all the data required for schema reconciliation can be stored. It has been the key to avoid hard-coded

programs which are the major cause of high maintenance cost of the traditional systems. The representation of the data in the meta-database also ensures that the automatic view modification can be conducted. Consequently, the manual work has been further reduced.

The schema evolution detection has proved to be effective. The rules for examining the validation and the rules for identifying the affected views and views which should be discarded have been effective. The processes for tackling eight types of schema evolution have proved to be correct and effective. It has largely reduced the manual maintenance work caused by schema evolution. Some schema evolution such as attribute removal and relation removal and attribute domain change and attribute decomposition may lead to discarding the views so that manual work is required. The major issue of the current schema evolution detection has been that it has only been able to tackle the types of schema evolution one by one and sequentially. In the case that the schema evolution cannot be recalled or the complete database schema is replaced, the schema evolution detection cannot provide help. However, it may be improved by further work which is discussed in next section.

The query process has proved to be effective in the original work which is identifying the source databases. This process ensures that the user queries do not require modification when organizational evolutions occur unless the organization property which is designated explicitly by the user queries changes. This is achieved by recording the organizational evolutions in the meta-database.

The SOA and web services have been helpful in reducing the maintenance cost. They have made the removal and addition of new source databases very simple. Also, both schema evolution detection and query process rely on the registry service which centrally manages the information such as the name and URL of all the services. This has also tackled system evolution easily.

Comparing EA-SODIA to the traditional systems such as the federated system and the

mediated system has proved an interesting study. EA-SODIA is similar to the LAV mediated system as they both use LAV approach. However, it provides a better ability to tackle evolution by not requiring hard-coded programs. The federated system can provide complete results, but it is incapable of dealing with evolution problems.

Overall, EA-SODIA has proved to a successful architecture for data integration with the capability of addressing most types of evolution.

9.5 Further Work

The work presented in this thesis could be extended in many ways and some ideas are discussed in this section.

9.5.1 Other Source Databases

Currently, other types of data sources cannot be integrated into the system because the heterogeneity among them cannot be solved using relational algebra views. However, some legacy systems which involve other types of databases (e.g. flat files and objective-oriented language) or existing queries (e.g. programs providing query results) do exist in practice. Therefore, it is useful to extend the system to have the ability of integrating other types of data sources. In fact, the system has the potential to integrate other data sources. Relational algebra operations take relations as operands and output another. These relations do not have to be the real relations in a relational database schema. They can be views or queries. Thus, other programs such as queries or the programs which obtain data from a data source also have the potential to be taken by a relational algebra operation as operands, as long as the output from them are relations.

A possible solution is that data in other data models and existing queries are translated into relations. The schemas of these relations are stored into the meta-database with additional information such as whether the relation is an existing query or data in

other data model and how it can be accessed. Exporting views and importing views can then be built as usual in terms of the relations produced above. However, this may require a program for each relation in order to obtain result from the actual database. These programs usually translate a conjunctive query received from a DIS into an equivalent query in the query language supported by the actual database. This leads to a new challenge that the programs are hard-coded and require modification when the actual database changes. Thus, we propose that the data in other data model is described as relations using some language (e.g. Description Logic) which can be structured and stored into the meta-database. It is similar to building views in terms of the local schema of the relational database schema using relational algebra operators. A component, which translates conjunctive queries into queries in other language, is added into the DS. Consequently, the only part requiring modification, when an actual database changes, is the information in the meta-database. The current schema evolution detection can then be improved to modify this information automatically.

However, it may be hard to describe some data sources such as flat files. Therefore, hard-coded programs may still be required. Further research needs to be carried out.

9.5.2 Extending the SED

As discussed in Chapter 8, schema evolution must be entered into the system in order. The schema evolution detection cannot provide help in the following cases:

- The schema evolution cannot be recalled.
- The complete database schema is replaced.

The system has the potential to deal with this situation. The former one can be solved by adding date and time of the schema evolution into the meta-database. Thus, the new process of schema evolution detection can be produced to tackle a batch of schema evolution. A software tool needs to be provided for the database administrator to apply schema evolution and store all the schema evolution in the meta-databases.

As for the latter case, a possible solution may be to provide software tools in order to find out the mapping from the new schema to the old one using some automatic schema mapping methods if the local schema is completely replaced by another one. Although software tools can be provided to help the manual work, the DS may have to be rebuilt by data provider.

9.5.3 Query Based on the Organizational Structure before Evolution

As discussed in Chapter 1, the integrated system based our architecture only answers a user query based on the current organizational structure. For example, a hospital, which used to belong to Newcastle, has become one in Durham since 2009. This evolution is tackled automatically in the integrated system. Thus, the answer to a user query asking for the number of patents in Durham will involve the patents in this hospital since the evolution. However, as the system keeps only the current version of the organizational structure, one cannot realize that the hospital used to be in Newcastle.

A possible solution, similar to that in section 9.5.2, is to add additional properties such as date and time into the organizational evolution stored in the meta-database. Thus, the system knows what and when organizational evolution occurred so that the query processing component can find the right version of the organizational structure based on the evolution. Take the previous example, the system will check the organizational evolution records in the meta-database and realize that a hospital moved from Newcastle to Durham. Therefore, the system will not access the database of this hospital if a user query asks for the number of patents in Durham in 2008.

Organizational structure evolution can have more impact on materialized systems, because both the existing materialized views and the programs extracting data from source databases require modification. We believe that the architecture in this thesis with the solution proposed in this section may also help in a materialized system if this architecture is used as a part which extracts data from the source databases.

However, it requires further research and experiments.

9.5.4 Dynamic Tackling of Schema Evolution

Currently, the schema evolution detection process is triggered manually by a data provider if the local schema changes. Consequently, the DS has to be discarded temporarily in order to conduct the schema evolution detection process until the schema evolution is tackled. Namely, the user query cannot obtain answer from this DS until the exporting views and the importing views are modified based on the schema evolution.

A possible solution may be for a DS to get the current exporting views based on the evolution history stored in the meta-database. Each schema evolution is recorded in sequence in the meta-database and the exporting views do not require changes. When a user query is received, the DS can automatically obtain the current exporting views by taking the schema evolution into account. For example, the following evolution occurs:

1) The name of the attribute A1 in relation R1 has been changed to B1.
2) The name of the relation R1 has been changed to R2.

The exporting views do not need to be changed when the above evolution occurs. When a user query involving the relation R1 and A1 is received, the DS finds the above evolution in the meta-database and then changes the user query to involve R2 and B1. In this way, the DS does not need to be suspended and is able to tackle schema evolution automatically each time when a user query comes. It may further reduce the manual maintenance.

The challenge is that for such schema evolution as Relational Removal and Attribute Removal, human intervention may still be required to modify the exporting views. Therefore, the combined approach of this solution and the schema evolution detection is preferred. It also requires further research and experiments.

9.6 Final Summary

A review of the work accomplished has been presented in this chapter. The overall success of the research has been considered in terms of the criteria shown in Chapter 1, and the two directions for further work have been established.

This thesis has examined the context, motivation, and definition of data integration, leading to the development of an architecture with some algorithms to reconcile schemas of source databases and process user queries and solve some evolution problems. The service-based architecture, called Evolution Adaptive Service-Oriented Data Integration Architecture, has been presented. Three methods, Relational Schema Mapping by Views, Query Processing and Schema Evolution Detection, have also been described and compared to similar systems. An extensive evaluation using a case study has demonstrated various characteristics of the methods by examining the response variables to support pre-defined hypotheses. Issues found in the case study were also discussed. Ideas for further work have been suggested.

EV-SODIA with RSMV and Query Process and Schema Evolution Detection is a novel and successful solution to data integration.

Appendix

A.1 Relational Algebra Operators

A set of extended algebra operators is used to construct exporting views in terms of the local schema. Those operators have been slightly modified to fulfil the needs in this work.

A.1.1 Set Operators on Relations

Three most common operations on sets are taken into account: union, intersection, and difference [84].

- The **Union** of R and S, denoted **R ∪ S**, is the set of elements that are in R or S or both.
- The **Intersection** of R and S, denoted **R ∩ S**, is the set of elements that are in both R and S.
- The **Difference** of R and S, denoted **R − S**, is the set of elements that are in R but not in S.

These operators can be applied to relations after putting some additional conditions on operand relations R and S.

1. The schema of R and the schema of S must have identical set of attributes. and the types (domains) of each attribute must be the same in R and S.
2. The attributes of R must be in the same order to the attributes of S.

It will be allowed that the number of attributes of two relations is not identical. This is explained in a later section.

A.1.2 Cartesian Product

The Cartesian Product [84] of two sets R and S is the set of pairs that can be formed by choosing the first element of the pair to be any element of R and the second any

element of S. It is denoted R × S. When R and S are relations, the product is essentially the same. The resulting tuple is a longer tuple, with one component for each of the components of constituent tuples. The resulting relation is therefore a relation with all the attributes of R followed by all the attributes of S in the same attribute order to S and R. For example, given two relations R with the schema R{a, b, c} and S with the schema S{b, c, d}, the resulting relation of R×S is P with the schema P{a, b, c, b, c ,d}. Note that the attribute order of the resulting relation schema is important. The first b is from R, while the second is from S.

A.1.3 Common Join

Common Join is a special natural join of two relations R and S, denoted as R $\bowtie_A$ S, in which we pair only those tuples from R and S that agree in a set of designated attributes which are common to the schemas of R and S. More precisely, let A {A1, A2,…, An} be a set of attributes that are in both the schema of R and the schema of S. Thus, a tuple r from R and a tuple s from S are successfully paired if and only if r and s agree on each of the attributes A1, A2,… , An. The result of the pairing is a tuple, called a joined tuple, with one component for each of the attributes in the union of the schemas of R and S. The resulting relation is the set of all joined tuples. For example, given two relations R and S with the schemas R{a, b, c} and S{b, c, d} with two common attributes b and c, the resulting relation is P with the schema P{a, b, c, d}. Note that the difference between a natural join and a common join is that the natural join agrees on all the common attributes of two relations, while common join agrees on the designated common attributes which are a subset of all the common attributes.

A.1.1 Selection

The selection operator [84], applied to a relation R, produces a new relation with a subset of R's tuples. The tuples in the resulting relation are those that satisfy some condition C that involves the attributes of R. This operation is denoted as $\sigma_C(R)$. The schema for the resulting relation is the same as R's schema, and the attributes of the

resulting relation schema are in the same order as for R.

C is a conditional expression of the type with which we are familiar from conventional programming language; for example, conditional expression follows the keyword “if” in programming languages such as C or Java. The only difference is that the operands in condition C are either constants or attributes of R. The resulting relation is derived by applying C to each tuple t of R by substituting, for each attribute A appearing in condition C, the attribute of t for attribute A. If after substituting for each attribute of C the condition C is true, then t is one of the tuples in the resulting relation.

A.1.4 Projection

The original projection operator is used to produce from a relation R a new relation with a schema that has only some attributes of the schema of R. It is denoted $\pi_L(R)$. L is a list of attributes of relation R. The result of expression $\pi_{A1,A2,\ldots,An}(R)$ is a relation that has a schema with attributes A1, A2, …, An of R.

We extend the projection operator to allow it to compute with attributes of tuples as well as choose attributes. The extended projection is also denoted $\pi_L(R)$. However, the projection lists L can have the following kinds of elements.

1. A single attribute of R
2. An expression x→y, where x and y are names for attributes. the element x→y in the list indicates that we take the attribute x of R and rename it y so that the name of this attribute in the schema of the resulting relation is y.
3. An expression E→z, where E is an expression involving attributes of R, constants, arithmetic operators, and string operators, and z is a new name for the attribute that results from the calculation implied by E. For example, a + b → x as a list element represents the sum of the attributes a and b, renamed x. Element c || d → e means concatenate the presumably string-valued or time-valued attributes c and d and call the result e.

The result is a relation whose schema is the names of the attributes on list L.

A.1.5 Grouping

The grouping operation [84] aims to consider the tuples of a relation in groups, corresponding to the values for one or more other attributes, and then aggregating or summarizing the values for one attribute within each group.

The grouping operator is denoted $\gamma_L(R)$, where L is a list of elements each of which is either:

- An attribute of the relation R to which the γ is applied; this attribute is one of the attributes by which R will be grouped. This element is called a *grouping attribute*.
- An aggregation operator applied to an attribute of the relation. To provide a name for the attribute corresponding to this aggregation in the result, an arrow and new name are appended to the aggregation. The underlying attribute is called an *aggregated attribute*.

The aggregation operators are used to summarize or aggregate the values for an attribute of a relation. The aggregation operators supported in this work are:

1. SUM produces the sum of a list of values for an attribute with numerical values.
2. AVG produces the average of an attribute with numerical values.
3. MIN and MAX, applied to an attribute with numerical values, produces the smallest or largest value, respectively.
4. COUNT produces the number of values for an attribute. Equivalently, COUNT can apply to any attribute of a relation to produce the number of tuples of that relation.

For example, given the relation R(a, b, c, d, e), the grouping operation can be written as:

$$\gamma_{a,\ MIN(b)\rightarrow g,\ COUNT(e)\rightarrow h}(R)$$

The relations taken as operands of these operators are in fact relation schemas until the operators are executed at a particular time. A relation schema can be referred to as variable of a relation which can be assigned to a value at a particular time. A relation instance with a set of tuples at a particular time is referred to as a constant or a value of a relation schema. Therefore, in this work, when we talk about a relation, we are actually talking about a relation instance. Otherwise, we always mean relation schema.

A.2 Expression Tree of a View

Writing single algebra operations on one or two relations as queries does not show the power that the relational algebra has. However, the algebra operations take relations as operands and the result of an operation is still a relation. Therefore, it is allowed to form an expression of arbitrary complexity by applying operations to the result of other operations. Consequently, more complex queries can then be constructed by forming complex expressions. An expression can be represented as an expression tree. For example, we have two relation schemas R (a, b, c, d) and S (a, e, f,), an expression may be:

$$\pi_{a,b,e} (\sigma_{b>100} (R \bowtie_a S) \cap \sigma_{e=50} (R \bowtie_a S))$$

The expression tree of the above expression is shown in Figure 4-4.

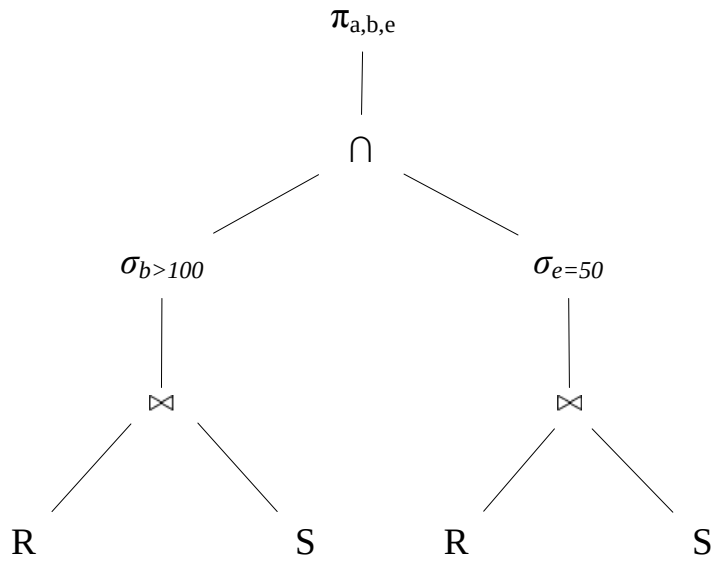

Figure 4-4 Expression tree for a relational algebra expression

It can be seen from Figure 4-4 that at an interior node an operator is applied to the arguments, which are the results of its children. In fact, there is a resulting relation, a relation or a view, which is derived at each node which is one of the operands of its parent node. Apparently, the leaves are relations or views that already exist, while the result from the root of the tree is the final result of the expression. At each interior which is neither leaf nor root, there is still a temporary view derived from the operator at that node.

In order to show clearly the temporary views, an alternative way to represent a expression is to invent names for the temporary views that correspond to the interior nodes of the tree and write a sequence of assignments that create a value for each. The notation that we use for assignment statement is:

1. A relation (view) name and parenthesized list of attributes for that relation.
2. The assignment symbol :=.
3. Any algebra expression on the right.

Thus, the expression tree shown in Figure 4-4 can be represented differently as:

V01 (a, b, c, d, e, f) := R $\bowtie_a$ S

V02 (a, b, c, d, e, f) := $\sigma_{b>100}$ (V01)

V03 (a, b, c, d, e, f) := $\sigma_{e=50}$ (V01)

V04 (a, b, c, d, e, f) := V02 ∩ V03

$V_{\textbf{Answe}r}$ (a, b, e) := $\pi_{a,b,e}$(V04)

The order of the assignment is flexible as long as the values of the views have been created before they are taken by their parents to create values for the parents themselves. V_{Answer} is the final result of the whole expression. In this research, the view on each node must only have one algebra operator within the expression on the right side of the assignment statement. This view is called the *atomic view*. The representation of an atomic view consists of two parts, the schema of the view (name and attributes) and the expression on the right side of the assignment.

References

[01] P.J.Layzell, K.H.Bennett, D.Budgen, P.Brereton, L.A.Macaulay, M.Munro, '***Service-Based Software: The Future for Flexible Software***' *Asia-Pacific Software Engineering Conference*, 5-8 December 2000, ISBN: 0-7695-0915-0 pp. 214-222

[02] K.H.Bennett, N.E.Gold, M.Munro, J.Xu, P.J.Layzell, D.Budgen, O.P.Brereton and N.Mehandjiev, '***Prototype Implementations of an Architectural Model for Service-Based Flexible Software***', *in Proceedings Thirty-Fifth Hawaii International Conference on System Sciences (HICSS-35)*, edited by Ralph H. Sprague, Jr. p.76, 2002

[03] M. Turner, D. Budgen, and P. Brereton, '***Turning software into a service***,' *IEEE Computer*, 2003, vol. 36, pp. 38-44.

[04] K. H. Bennett, N. E. Gold, P. J. Layzell, F. Zhu, O. P. Brereton, B. D., J. Keane, I. Kotsiopoulos, M. Turner, J. Xu, O. Almilaji, J. C. Chen, and A. Owrak, '***A Broker Architecture for Integrating Data using a Web Services Environment,***' presented at *First International Conference on Service-Oriented Computing* (IC-SOC 2003), Trento, Italy, 2003.

[05] I. Kotsiopoulos, J. Keane, M. Turner, P. Layzell, and F. Zhu, '***IBHIS: Integration Broker for Heterogeneous Information Sources,***' presented at *27th Annual International Computer Software and Applications Conference* (COMPSAC 2003), Dallas, 2003.

[06] F. Zhu, M. Turner, I. Kotsiopoulos, K. Bennett, M. Russell, D. Budgen, P. Brereton, J. Keane, P. Layzell, M. Rigby, and J. Xu, '***Dynamic Data Integration Using Web Services,***' presented at *2nd International Conference on Web Services* (ICWS 2004), San Diego, California, USA, 2004

[07] M. Turner, Zhu, F., Kotsiopoulos, I., Russell, M., Budgen, D., Bennett. K., Brereton, P., Keane, J., Layzell, P., and Rigby, M., '***Using Web Services Technologies to create an Information Broker: An Experience Report***,' presented at *26th International Conference on Software Engineering* (ICSE 2004), Edinburgh, Scotland, 2004.

[08] A.Gounaris, J.Smith, N.W. Paton, R.Sakellariou, A.A.A. Fernandes, P.Watson: ***"Adapting to Changing Resource Performance in Grid Query Processing".****Proc. of the 1st International Workshop on Data Management in Grids, DMG'05.*

[09] N.M. Alpdemir, A.Mukherjee, A.Gounaris, N.W. Paton, A.A.A. Fernandes, R. Sakellariou, P.Watson, P.Li: ***'Using OGSA-DQP to Support Scientific Applications for the Grid'.*** *First International Workshop on Scientific Applications in Grid Computing SAG'04, LNCS 3458,* pp. 13-24

[10] I.Gorton, J.Almquist, K.Dorow, P.Gong, D.Thurman, ***'An Architecture for Dynamic Data Source Integration',*** *Proceedings of the 38th Annual Hawaii International Conference on System Sciences* (HICSS'05) - Track 9, January 2005.

[11] '***Web Services Architecture***', W3C Working Group w3c-wsa http://www.w3.org/TR/2004/NOTE-ws-arch-20040211 Accessed in February 2005

[12] '***Service-Oriented Architecture expands the vision of Web services, Part 1***', http://www-128.ibm.com/developerworks/webservices/library/ws-soaintro.html Accessed in June 2005

[13] '***The Web Services Conceptual Architecture***', http://www-306.ibm.com/software/solutions/webservices/pdf/WSCA.pdf Accessed in February 2005

[14] ***XML Protocol Working Group*** http://www.w3.org/2000/xp/Group/ Accessed in February 2005

[15] '***Bridging the integration gap, Part 1: Federating grid data***', http://www-128.ibm.com/developerworks/grid/library/gr-feddata/ Accessed in June 2006

[17] '***Service-Oriented Architecture and Web Services: Concepts, Technologies, and Tools***', http://java.sun.com/developer/technicalArticles/WebServices/soa2/ Accessed in March 2005

[18] '***Migrating to a Service-Oriented Architecture'***, http://www-128.ibm.com/developerworks/webservices/library/ws-migratesoa/ Accessed in March 2005

[19] '***New to Service-Oriented Architectures***' http://www-128.ibm.com/developerworks/webservices/newto/ Accessed in May 2005

[20] ***SOAP 1.2 (Specification***) http://www.w3.org/TR/2003/REC-soap12-part0-20030624/ Accessed in December 2004

[21] ***Web Services Addressing*** http://www.ibm.com/developerworks/webservices/library/specification/ws-add/ Accessed in May 2007

[22] ***UDDI 3.0*** http://uddi.org/pubs/uddi_v3.htm Accessed in December 2005

[23] ***WSDL 2.0 (Working Group)*** http://www.w3.org/2002/ws/desc/ Accessed in December 2004

[24] ***Web Service Architecture from W3C*** http://www.w3.org/TR/2004/NOTE-ws-arch-20040211/ Accessed in December 2004

[25] D. Austin, A. Barbir, C. Ferris, S. Garg, '***Web Services Architecture Requirements'***, *W3C Working Group Note*, 11 February 2004. http://www.w3.org/TR/2004/NOTE-wsa-reqs-20040211 Accessed in April 2005

[26] S. C. Kendall, J. Waldo, A. Wollrath, G. Wyant, '***A Note on Distributed Computing'***, November 1994 http://research.sun.com/techrep/1994/abstract-29.html Accessed in December 2004

[27] ***'Web Services Resource Framework'***

http://www-128.ibm.com/developerworks/library/specification/ws-resource/ Accessed in January 2005

[28] '***Web Services Metadata Exchange'*** http://www-128.ibm.com/developerworks/webservices/library/specification/ws-mex/ Accessed in June 2005

[29] '***Grid in action: Monitor and discover grid services in an SOA/Web services environment'*** http://www-128.ibm.com/developerworks/grid/library/gr-gt4mds/ Accessed in July 2005

[30] '***Build grid applications based on SOA***' http://www-128.ibm.com/developerworks/grid/library/gr-soa/ Accessed in March 2006

[31] ***Data Access and Integration (DAI) Project*** http://www.ogsadai.org/ Accessed in May 2005

[32] ***UDDI v3.0*** http://uddi.org/pubs/uddi_v3.htm#_Toc12653784 Accessed in December 2005

[33] '***GT4 Primer***' http://www.globus.org/toolkit/docs/4.0/key/index.html Accessed in June 2006

[34] ***Information Services (MDS): Key Concepts*** http://www.globus.org/toolkit/docs/4.0/info/key-index.html Accessed in July 2006

[36] A. Sheth and J. Larson., '***Federated database systems for managing distributed, heterogeneous and autonomous databases***' *ACM Computing Surveys*, 1990, 22, 3, pp. 183-236.

[37] E.A. Rundensteiner, A.Koeller, X.Zhang, '***Maintaining data warehouses over changing information sources***', *Communications of the ACM***,** June 2000, Volume 43 Issue 6

[38] I. Foster et al., "***The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration,***" *tech. report*, Glous Project

[39] Foster, I., Kesselman and Tuecke, "***The Anatomy of the Grid: Enabling Scalable Virtual Organizations***". *International Journal of High Performance Computing Applications*, 2001, 15 (3). 200-222.

[40] C.Bontempo, G.Zagelow, '***The IBM data warehouse architecture***',

*Communications of the ACM***,** September 1998, Volume 41 Issue 9

[41] M.Antonioletti , M.Atkinson , R.Baxter , A.Borley , N.P.C.Hong , B.Collins , N. Hardman , A.C. Hume , A.Knox , M.Jackson , A.Krause , S.Laws , J.Magowan , N.W. Paton , D.Pearson , T.Sugden , P.Watson , M.Westhead, '***The design and implementation of Grid database services in OGSA-DAI***', *Concurrency and Computation: Practice and Experience*, Volume 17, Issue 2-4 , Pages 357 - 376

[42] ***'A new Architecture for OGSA-DAI'*** http://www.ogsadai.org.uk/docs/OtherDocs/OGSA-DAI_Architecture_AHM05.pdf Accessed in October 2006

[43] St éphane Lafortune, Eugene Wong, '***A State Transition Model for Distributed Query Processing***' August 1986, *ACM Transactions on Database Systems* (TODS), Volume 11 Issue 3

[44] Jim Smith, Anastasios Gounaris, Paul Watson, Norman W. Paton, Alvaro A.A. Fernandes, Rizos Sakellariou, '***Distributed Query Processing on the Grid***' http://www.cs.man.ac.uk/grid-db/papers/dqp.pdf Accessed in October 2006

[45] M. Nedim Alpdemir, Arijit Mukherjee, Anastasios Gounaris, Norman W.Paton, Paul Watson, Alvaro A.A. Fernandes, Jim Smith, '***OGSA-DQP A service-based distributed query processor for the Grid***' http://www.ogsadai.org.uk/docs/OtherDocs/114.pdf Accessed in March 2006

[46] Alpdemir, M.N.Mukherjee, A.Gounaris, A.Paton, N.W.Watson, P.Fernandes, A.A.A.Smith, '***Service-Based Distributed Querying on the Grid'*** In (M. E. Orlwoska, S. Weerawarana, M. P. Papazoglu, and J. Yang, eds.) *Service Oriented Computing - ICSOC 2003 First International Conference*, Trento, Italy, December 15-18, 2003, LNCS 2910, Springer-Verlag, ISBN 3-540-20681-7. pp. 467-482.

[47] V.Raman, I.Narang, C.Crone, L.Haas, S.Malaika, T.Mukai, D.Wolfson and C.Baru, '***Data Access and Management Services on Grid***' http://www.cs.man.ac.uk/grid-db/papers/dams.pdf Accessed in October 2005

[48] N.Paton, M.Atkinson, V.Dialani, D.Pearson, T.Storey and P.Watson, '***Database Access and Integration Services on the Grid***' http://www.cs.man.ac.uk/grid-db/papers/dbtf.pdf Accessed in October 2006

[49] Do.Kossmann, '***The state of the art in distributed query processing***', *ACM Computing Surveys* (CSUR), December 2000, Volume 32 Issue 4.

[50] P. Bodorik, J. S. Riordon, C. Jacob, '***Dynamic distributed query processing techniques***', *Proceedings of the 17th conference on ACM Annual Computer Science Conference*, February 1989.

[51] D.Jantz, E. A.Unger, R. McBride, J.Slonim, '***Query processing in a distributed data base***', *Proceedings of the 1983 ACM SIGSMALL symposium on Personal and small computers*, December 1983.

[52] H.Stuckenschmidt, R.Vdovjak, G.J.Houben, J.Broekstra, '***Distributed semantic query: Index structures and algorithms for querying distributed RDF repositories***', *Proceedings of the 13th international conference on World Wide Web*, May 2004.

[53] D.L.Davison, G.Graefe, '***Dynamic resource brokering for multi-user query execution***', *ACM SIGMOD Record, Proceedings of the 1995 ACM SIGMOD international conference on Management of data***,** May 1995, Volume 24 Issue 2.

[54] S.P. Bradley and K.H. Bennett, "***Mental Health Minimum Data Set*** (MHMDS)", September 2002

[55] ***IBHIS executive summary***
http://www.co.umist.ac.uk/ibhis/summary.htm Accessed in October 2005

[56] ***Database Access and Integration Services (DAIS-WG)***
http://www.gridforum.org/6_DATA/dais.htm Accessed in October 2005

[57] ***Globus Toolkit*** http://www.globus.org/toolkit/ Accessed in May 2005

[58] I.Foster, C.Kesselman, '***The Grid: Blueprint for a New Computing Infrastructure***'

[59] ***The OGSA-DAI project,*** http://www.ogsadai.org.uk/ Accessed in July 2008

[60] ***WS-I, Web Services Interoperability***, *Basic Profile 1.0*. 2004.

[61]D. Calvanese, G. D.Giacomo, M. Lenzerini, and R. Rosati. ***Logical foundations of peer-to-peer data integration.*** In *Proc. of the 23nd ACM SIGACT SIGMOD SIGART Symp. on Principles of Database Systems (PODS 2004)*, 2004.

[62]D. Calvanese, G. D. Giacomo, M. Lenzerini, R. Rosati, and G. Vetere. ***Hyper: A framework for peer-to-peer data integration on grids.*** In *Proc. of the Int. Conference on Semantics of a Networked World: Semantics for Grid Databases (ICSNW 2004)*, volume 3226 of *Lecture Notes in Computer Science*, pages 144–157, 2004.

[63]M. Lenzerini. ***Data integration: A theoretical perspective***. *In Proc. of the 21st ACM SIGACT SIGMOD SIGART Symp. on Principles of Database Systems(PODS 2002)*, pages 233-246, 2002.

[64]J. Madhavan and A. Y. Halevy. ***Composing mappings among data sources***. *In Proc. of the 29th Int. Conf. on Very Large Data Bases (VLDB 2003)*, pages 572-583, 2003.

[65]A.Cali, D.Calvanese, G.D.Giacomo, and M.Lenz-erini. ***Accessing data integration systems through conceptual schemas***. *In Proc. of the 10th Ital. Conf. on Database Systems (SEBD 2002)*, pages 161-168, 2002.

[66]A.Halevy, Z.Ives, D.Suciu, and I.Tatarinov. ***Schema mediation in peer data management systems***. *In Proc. of the 19th IEEE Int. Conf. on Data Engineering (ICDE 2003)*, pages 505{516, 2003.

[67]T.Catarci and M.Lenzerini. ***Representing and using interschema knowledge in cooperative information systems***. J. of *Intelligent and Cooperative Information Systems*, 2(4):375{398, 1993.

[68]A.Y.Halevy. ***Answering queries using views: A survey***. *Very Large Database J.*, 10(4):270–294, 2001.

[69]M.Friedman, A.Levy, and T.Millstein. ***Navigational plans for data integration***. In *Proc. of the 16th Nat. Conf. on Artificial Intelligence (AAAI'99)*, pages 67–73. AAAI Press/The MIT Press, 1999.

[70]L.M.Haas,E.T.Lin, and M.A.Roth, ***Data integration through database federation***. *IBM Systems Journal*, Dec 2002.

[71] B.Lerner, ***A model for compound type changes encountered in schema evolution***. *ACM TODS*, 25:1, 83-127, 2000

[72] E. Rahm,and P. A.Bernstein, ***A survey of approaches to automatic schema matching***. *VLDB*, 334-350, 2001

[73] P.Shvaiko, ***A Survey of Schema-based Matching Approaches***. *Journal on Data Semantics*, 146-171, Vol. 4, 2005

[74] A.Y.Levy, D.Srivastava, T.Kirk, ***Data Model and Query Evaluation in Global Information Systems***. *Journal of Intelligent Information Systems*, 121-143, Vol. 5, 1991

[75] D.Budgen, M.Turner, I.Kotsiopoulos, F.Zhu, M.Rigby, K.Bennett, P.Brereton, J.Keane, P.Layzell, ***Managing healthcare information: the role of the broker.***

Studies in Health Technology and Informatics, 3-16, Vol. 112, 2005

[76] M.Antonioletti, A.Krause, N.W. Paton, A.Eisenberg, S.Laws, S.Malaika, J.Melton, D.Pearson, ***The WS-DAI Family of Specifications for Web Service Data Access and Integration***. *ACM SIGMOD Record*, 48-55, Vol 35, Issue 1, 2006.

[77] A.Gounaris, C.Comito, R.Sakellariou, D.Talia, ***A Service-Oriented System to Support Data Integration on Data Grids***. *Proceedings of the Seventh IEEE International Symposium on Cluster Computing and the Grid*, 627-635, 2007

[78] Y.Arens , C.Y. Chee , C.N.Hsu , C.A. Knoblock, ***Retrieving and Integrating Data from Multiple Information Sources***. *International Journal of Intelligent and Cooperative Information Systems*, 127-158, Vol. 2, 1993.

[79] D.Talia, A.Bilas and M.D. Dikaiakos, ***Service Choreography for Data Integration on the Grid. Knowledge and Data Management in GRIDs***, 19-33, Vol 1, 2007

[80] J.Widom, ***Research Problems in Data Warehousing***. *Conference on Information and Knowledge Management, Proceedings of the fourth international conference on Information and knowledge management*, 25-30, 1995.

[81] C.Comito, D.Talia, ***Grid Data Integration Based on Schema-Mapping***. *Lecture Notes in Computer Science, Applied Parallel Computing. State of the Art in Scientific Computing*, 319-328, Vol 4699, 2009.

[82] R. Hull. ***Managing Semantic Heterogeneity in Databases: A Theoretical Perspective***. In *Proc. of the 16th ACM SIGACT SIGMOD SIGART Symp. on Principles of Database Systems (PODS'97)*, 1997.

[83] G. Wiederhold. ***Mediators in The Architecture of Future Information Systems***. *IEEE Computer, 25(3)*:38-49, March 1992.

[84] H. Garcia-Molina, J. D. Ullman, and J. Widom. ***Database System Implementation, chapter 11: Information Integration***. Prentice Hall, 2000.

[85] S. Chaudhuri and U. Dayal. ***An Overview of Data Warehousing and OLAP Technology***. *SIGMOD Record*, 26(1):65{74, 1997.

[86] M. Jarke, M. Lenzerini, and P. Vassiliadis Y. Vassiliou. ***Fundamentals of Data Warehouses***. Springer Verlag, 2000.

[87] M. K. Mohania and G. Dong. ***Algorithms for Adapting Materialised Views in***

Data Warehouses. In CODAS, pages 309--316, December 1996.

[88] E.A.Rundensteiner, A.J.Lee, and A.Nica, ***On Preserving views in evolving environments.*** *In Proceedings of 4th Int. Workshop on Knowledge Representation Meets Databases (KRDB '97): Intelligent Access to Heterogeneous Information.* Athens, Greece (Aug. 1997), pp. 13.1--13.11.

[89] M. Bonjour and G. Falquet. ***Concept Bases: A Support to Information Systems Integration***. *Proceedings of CAiSE94 Conference, Utrecht, 1994*.

[90] T.Erl, ***Service-Oriented Architecture: Concepts, Technology, and Design***. Prentice Hall PTR (August 12, 2005) ISBN-10: 0131858580

[91] C.J. Date, ***An Introduction to Database System Addison Wesley****;* 8 edition (August 1, 2003) ISBN-10: 0321197844.

[92] J.Gehrke, R.Ramakrishnan, ***Database Management Systems***. McGraw Hill Higher Education; 3rd edition (November 1, 2002) ISBN-10: 0071230572.

[93] B.Kitchenham, L.Pickard, S.L. Pfleeger, ***Case Studies for Method and Tool Evaluation***. *IEEE Softw*., Vol. 12, No. 4. (July 1995), pp. 52-62.

[94] R.K. Yin, ***Case Study Research Design and Methods***, *Sage Publications,* Beverley Hills, Calif., 1984.

[95] R.K. Yin, ***Applications of Case Study Research***. *2nd edition, Sage Publications,* Beverley Hills, Calif., (December 4, 2002).

[96] S.Chawathe, H.Garcia-Molina, J.Hammer, K.Ireland, Y.Papakonstantinou, J.Ullman, and J.Widom. ***The TSIMMIS Project: Integration of Heterogeneous Information Sources.*** In *10th Meeting of the Information Processing Society of Japan (IPSJ)*, pages 7{18, Tokyo, Japan, October 1994.

[97] T.Kirk, A.Y.Levy, Y.Sagiv, and D.Srivastava. ***The Information Manifold***. In *AAAI Symposium on Information Gathering in Distributed Heterogeneous Environments*, 1995.

[98] J.D.Ullman. ***Information Integration using Logical Views***. *In International Conference on Database Theory (ICDT),* pages 19-40, Delphi, Greece, January 1997.

[99] C.Ghezzi, "***Ubiquitous, Decentralized, and Evolving Software: Challenges for Software Engineering***", *In Proc. Of 1st International Conference on Graph Transformation (ICGT'02), Lecture Notes in Computer Science,* Vol. 2502, 1-5, Springer, Barcelona, Spain, Oct. 2002.

[100] ***IEEE Standard Glossary of Software Engineering Terminology***. IEEE Std 610.12-1990

[101] B.P. Lientz, E.B. Swanson, ***Software Maintenance Management***, Addiso-Wesley Publishing Company, 1980, ISBN 0201042053.

[102] IEEE Computer Society, **IEEE Standard for Software Maintenance** (IEEE Std 1219-1998), *Institute of Electrical and Electronics Engineers, 1998, ISBN 0738103365, in IEEE Standards, Software Engineering Volume 2 Process Standards, 1999 Edition,* Institute of Electrical and Electronics Engineers, 1999, ISBN 0738115606.

[103] A. Y. Levy. ***Combining Artificial Intelligence and Databases for Data Integration***. In *Special issue of LNAI: Artificial Intelligence Today; Recent Trends and Developments*. Springer Verlag, 1999.

[104] R. Pottinger and A. Levy. ***A Scalable Algorithm for Answering Queries Using Views***. In International Conference on Very Large Data Bases (VLDB), pages 484-495, Cario, Egypt, September 2000.

[105] P.Mitra. ***Algorithms for Answering Queries Efficiently Using Views***. *Technical report,* Infolab, Stanford University, September 1999.